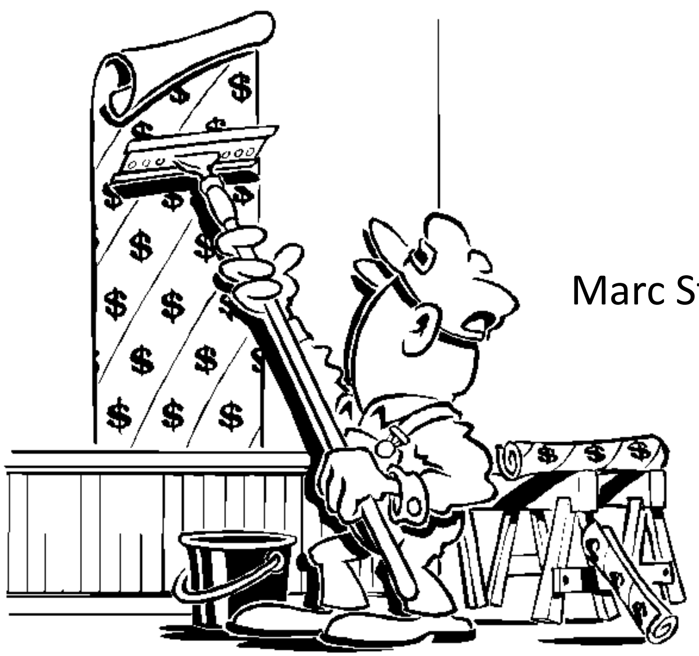
Lecture #11

Texture Mapping

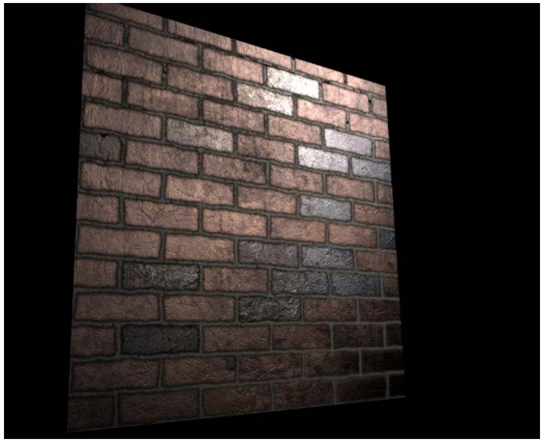


Computer Graphics
Winter Term 2016/17

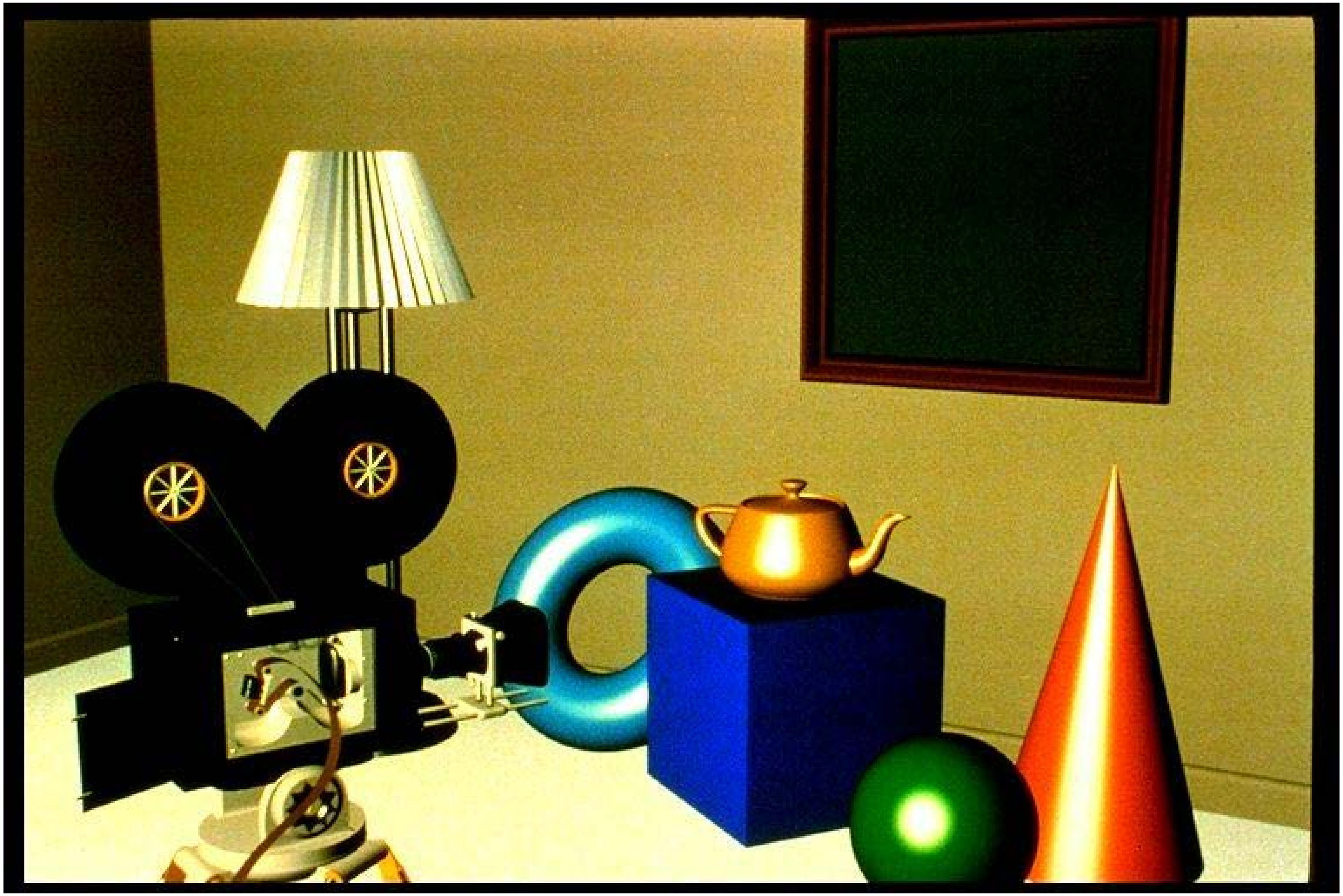
Marc Stamminger / Roberto Grosso

Texture Mapping - Introduction

- so far: detail through polygons & materials
- example: (large) brick wall
  - many polygons & materials needed for bricks
    $\rightarrow$ inefficient for memory and processing
- alternative: **Textures**
  introduced by Ed Catmull (1974)
  extended by Jim Blinn (1976)



Texture Mapping - Introduction



Foley, van Dam, Feiner, Hughes

Texture Mapping - Introduction

Foley, van Dam, Feiner, Hughes

Texture Mapping - Introduction

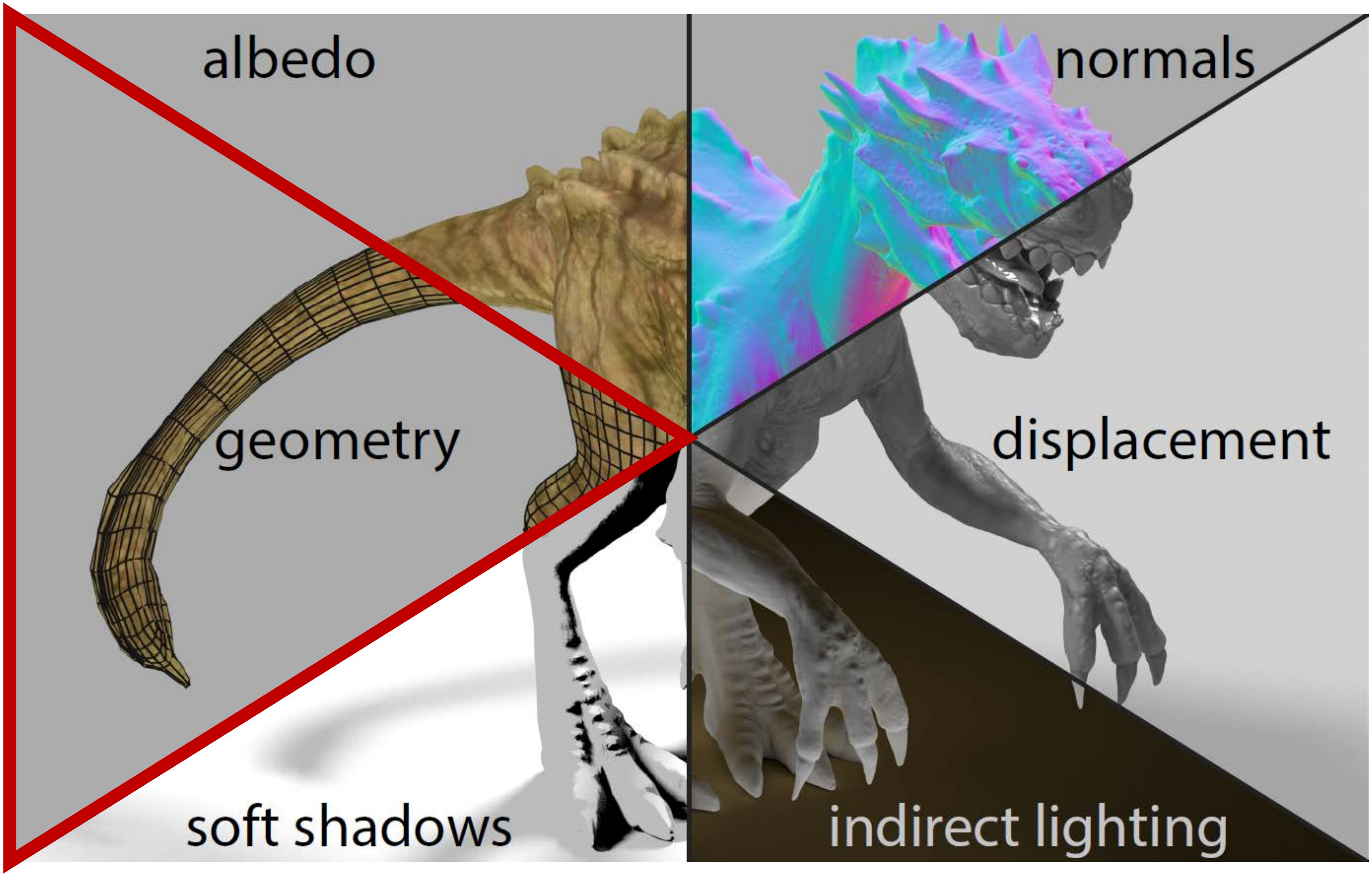
- What are textures or texture maps?
  - Functions or images that change the appearance of an object, typically its color
    → Coarse geometry (i.e. fast rendering), fine texture (i.e. fine visual detail)
  - Great performance gain compared to using huge triangle meshes with different materials
  - Can be 1D
    → heat map: maps the “temperature” of an object to color(cold=blue, warm=red)
  - or 2D
    → images to mapped onto the object like wall paper
  - or 3D
    → volumetric objects such as clouds
    → or solid objects such as wood

  - **for now, we only look at 2D textures**

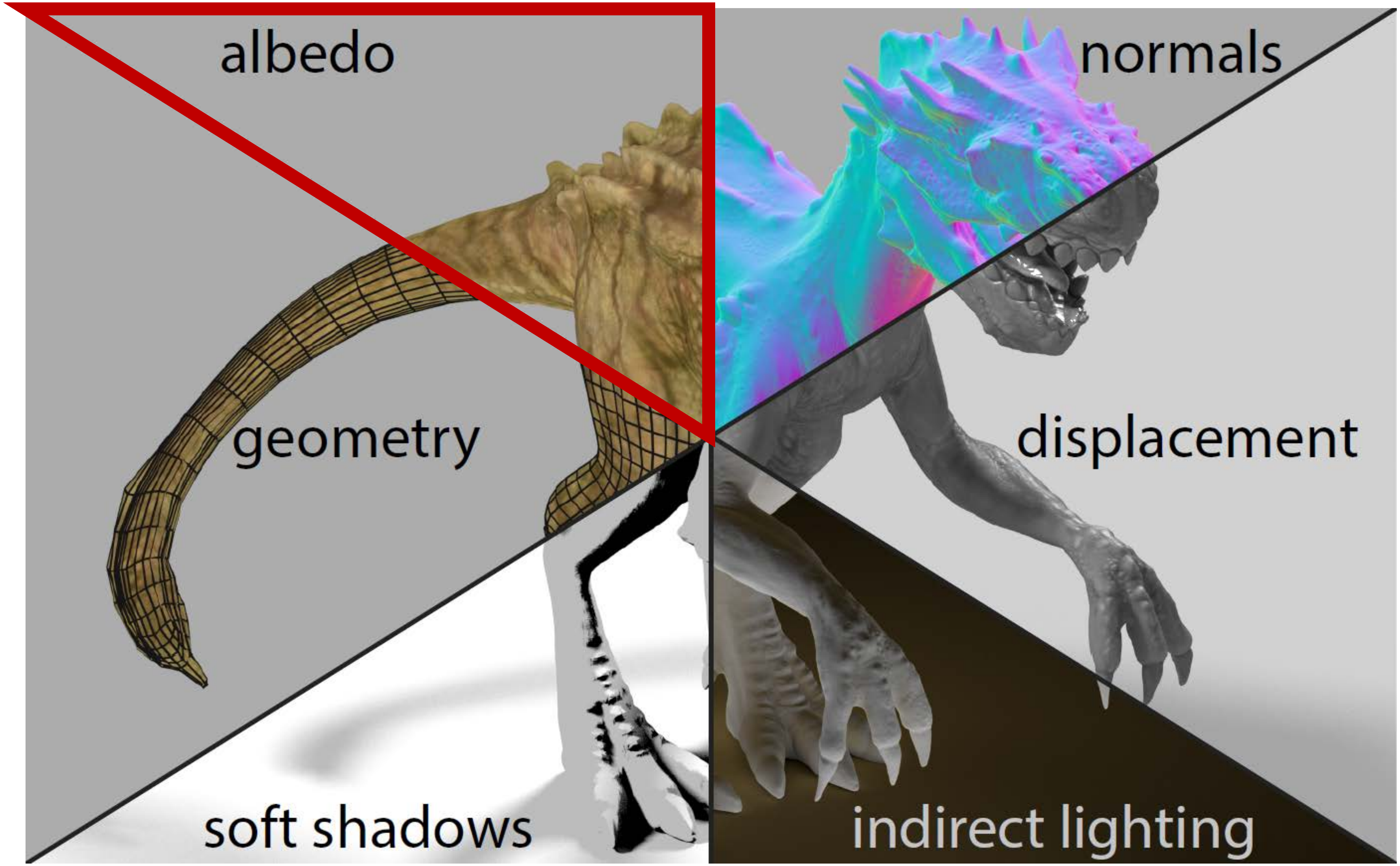
Texture Mapping - Introduction

- Textures usually contain color, e.g. the diffuse component of the Phong model
- But they can also contain specular color, ambient color or other material parameters
- And even much more!

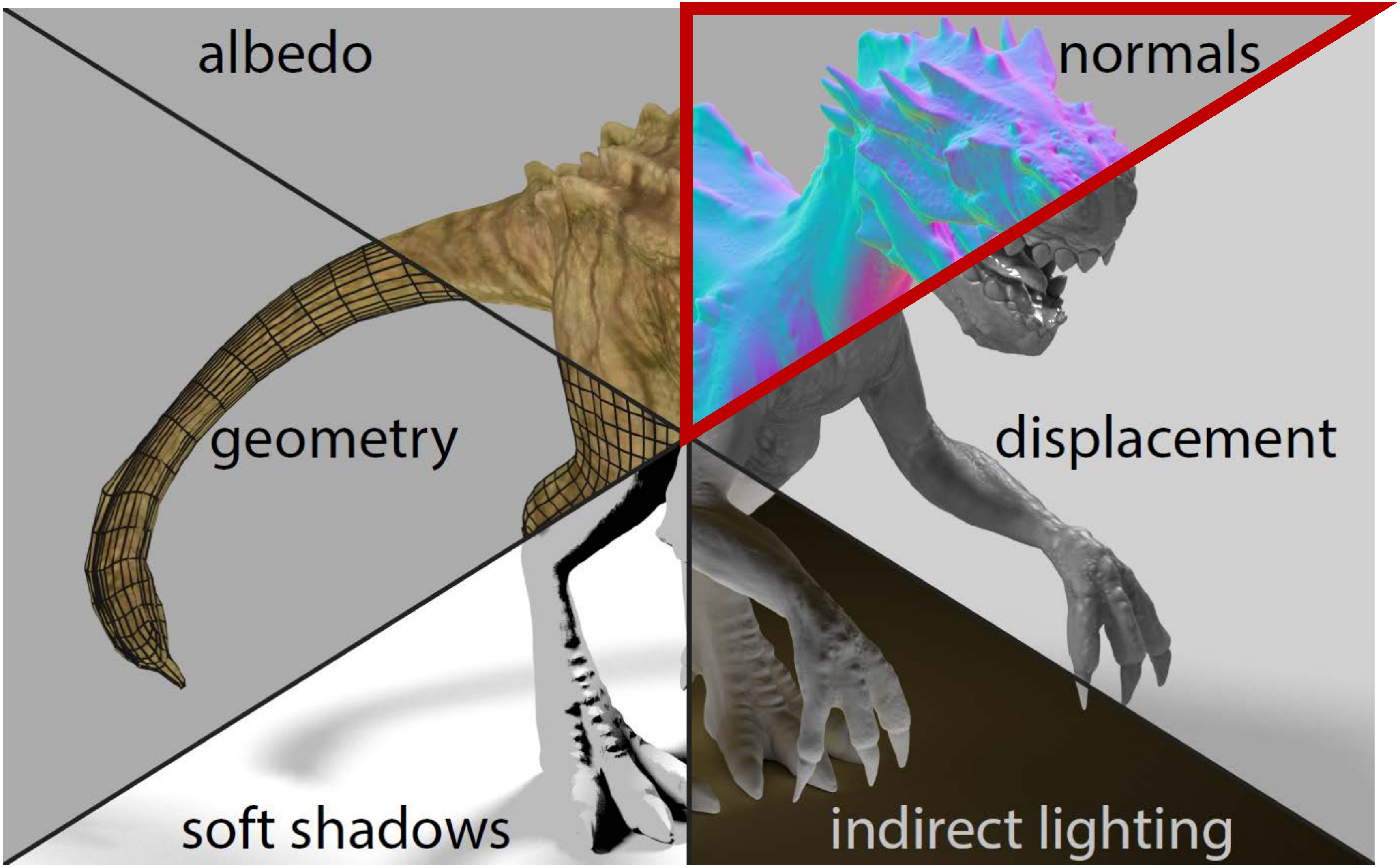
Texture Mapping - Introduction



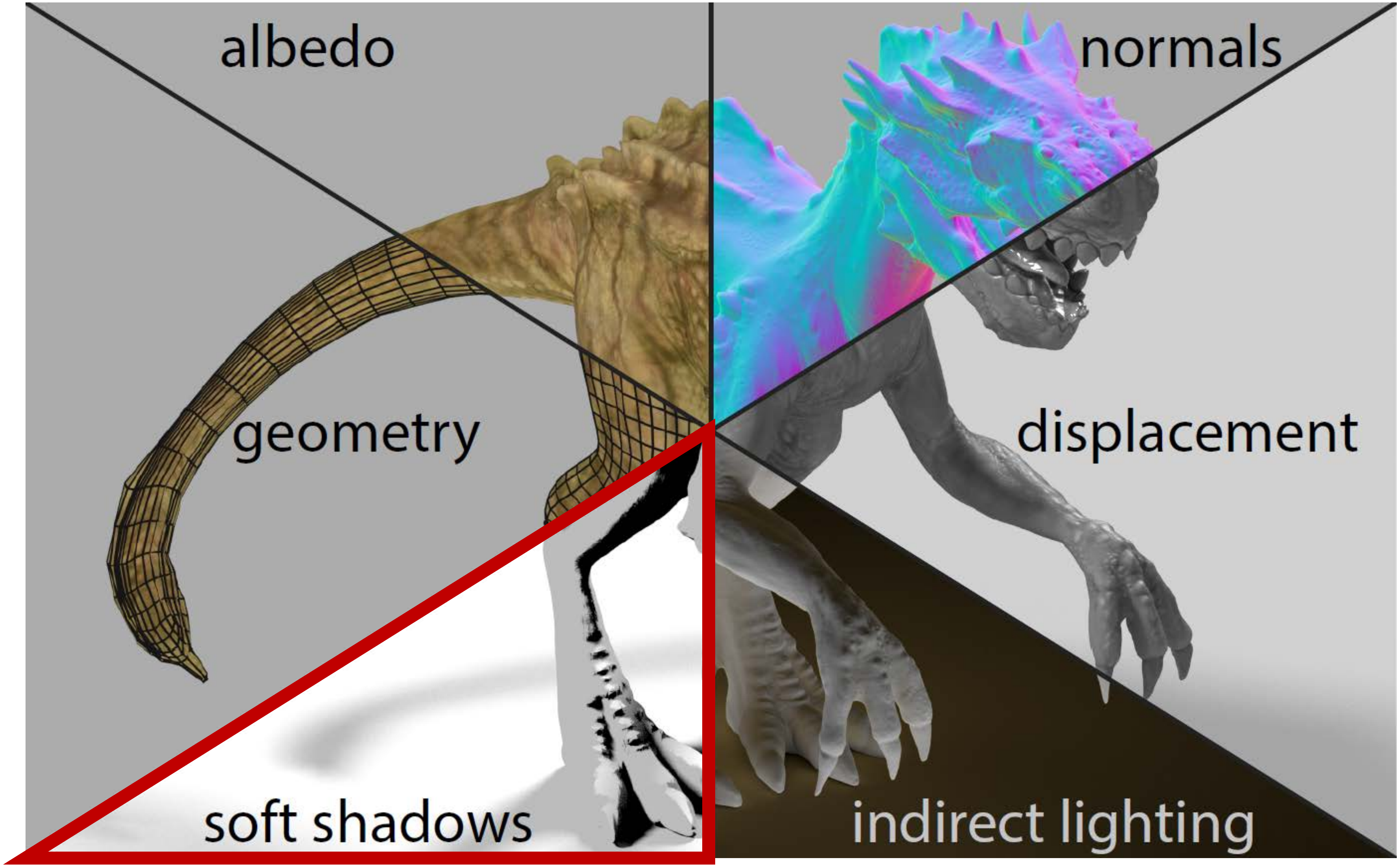
Texture Mapping - Introduction



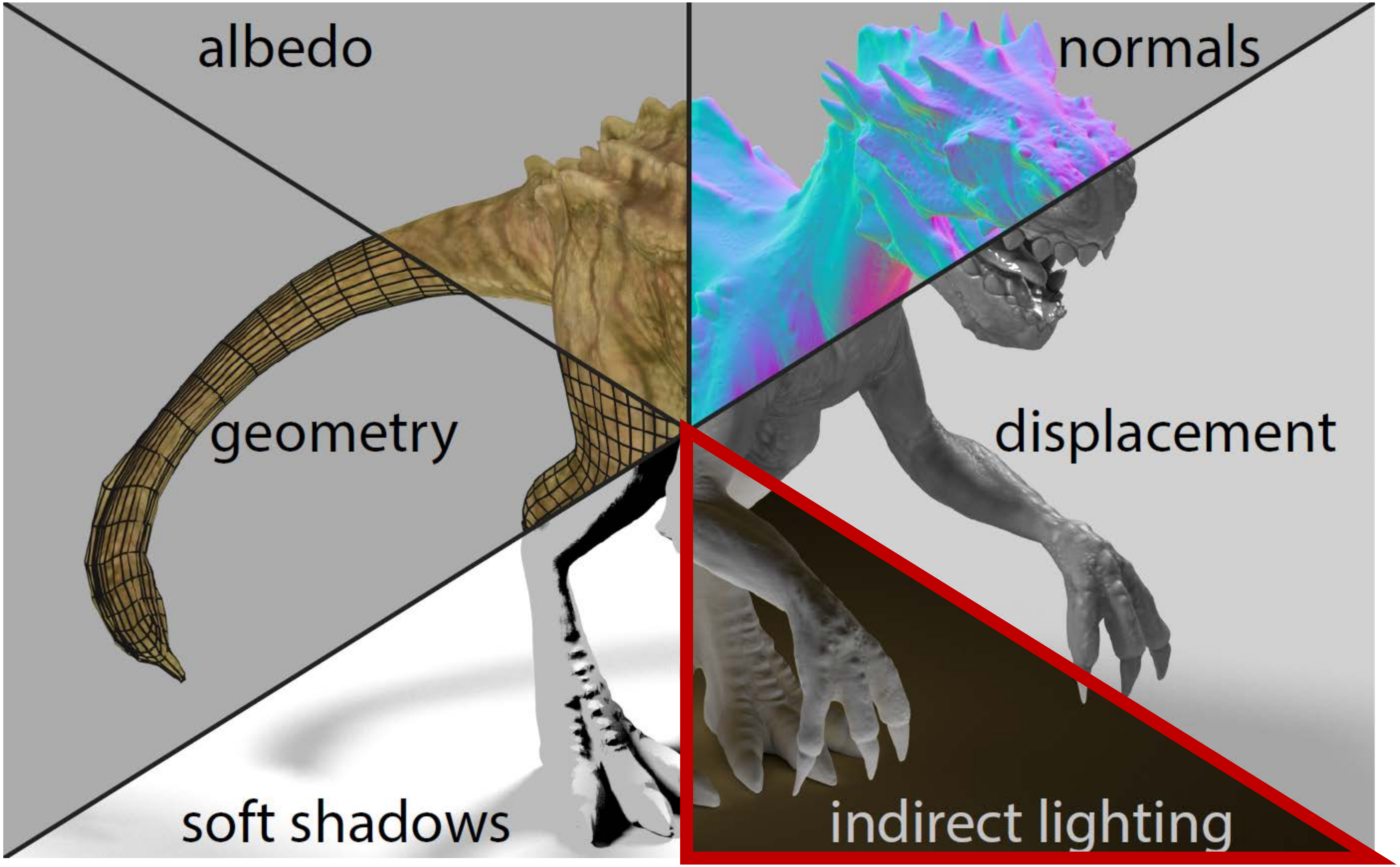
Texture Mapping - Introduction



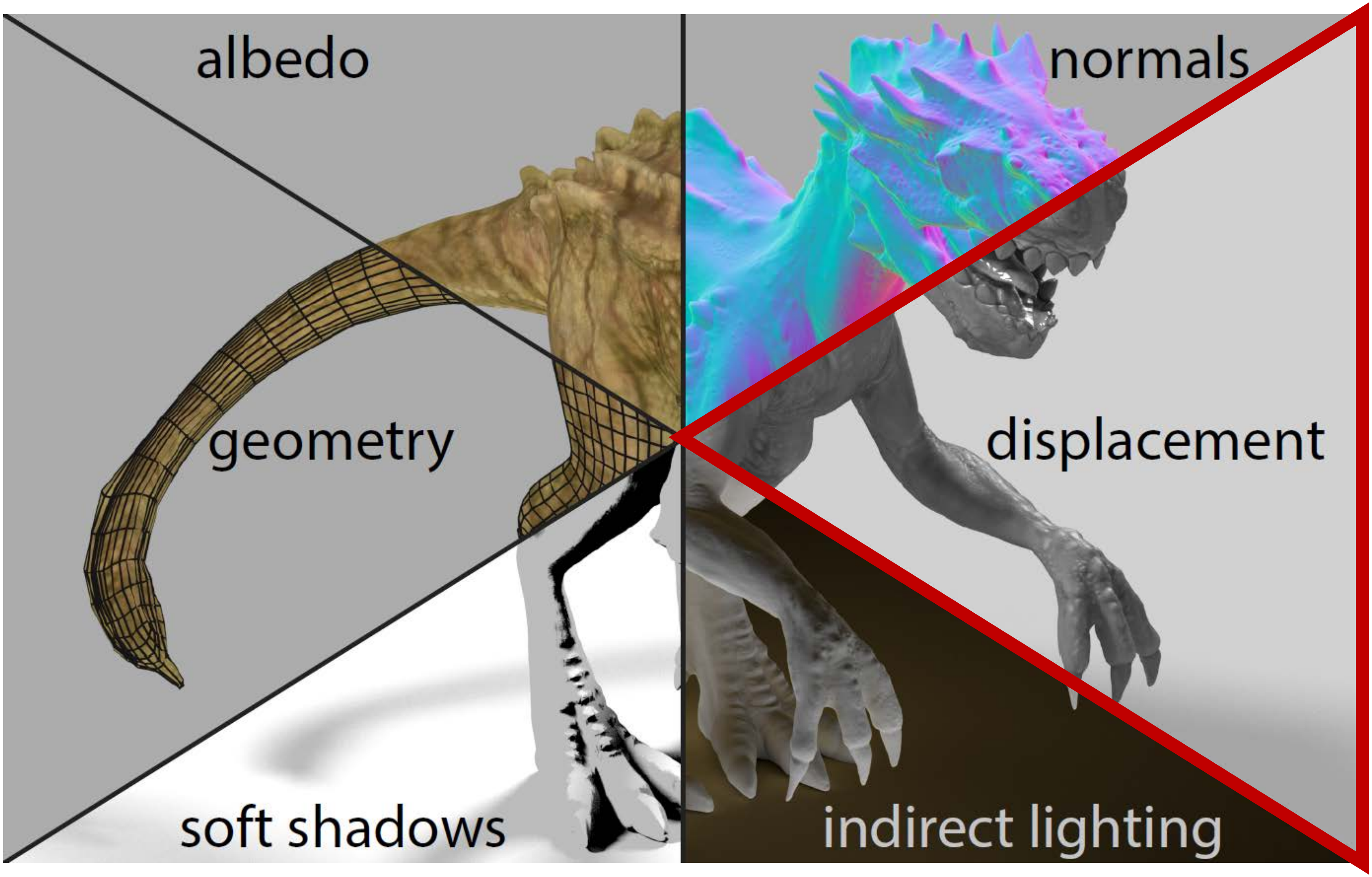
Texture Mapping - Introduction



Texture Mapping - Introduction

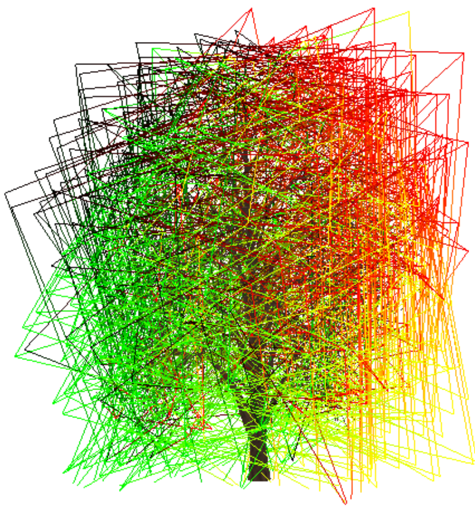


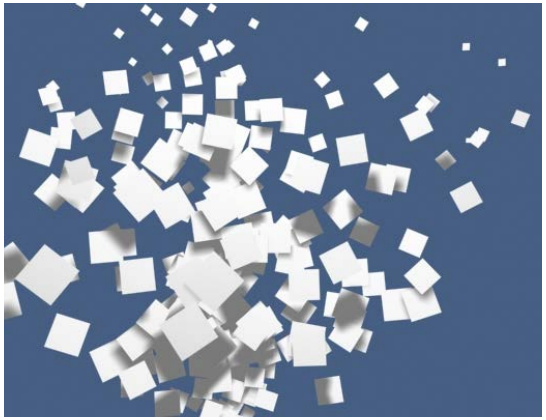
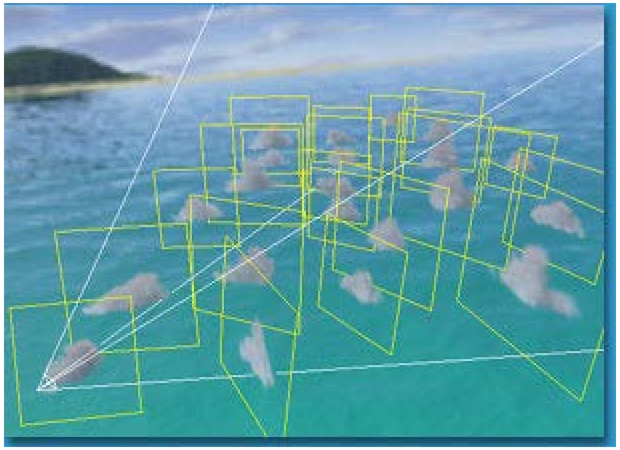
Texture Mapping - Introduction



Texture Mapping - Introduction

- many other applications for textures



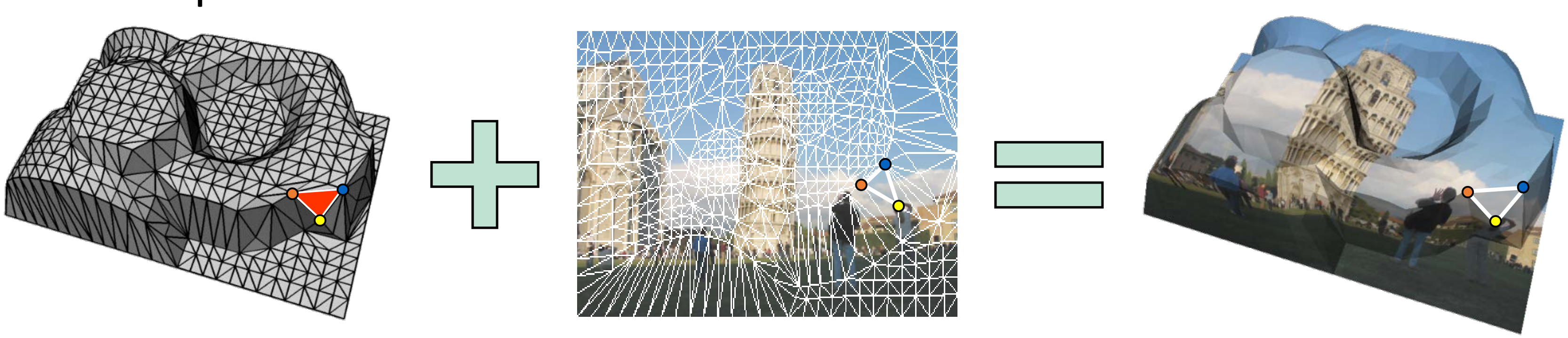




Parameterization

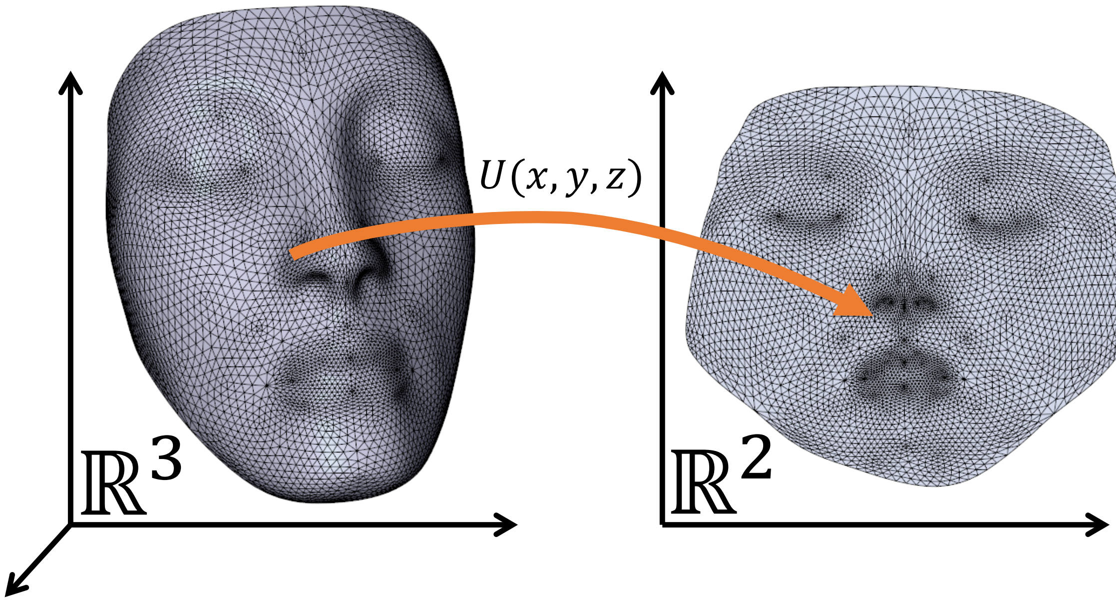
- Map 2D texture to surface in 3D → parameterization problem

- Simple parameterization

- difficult parameterization

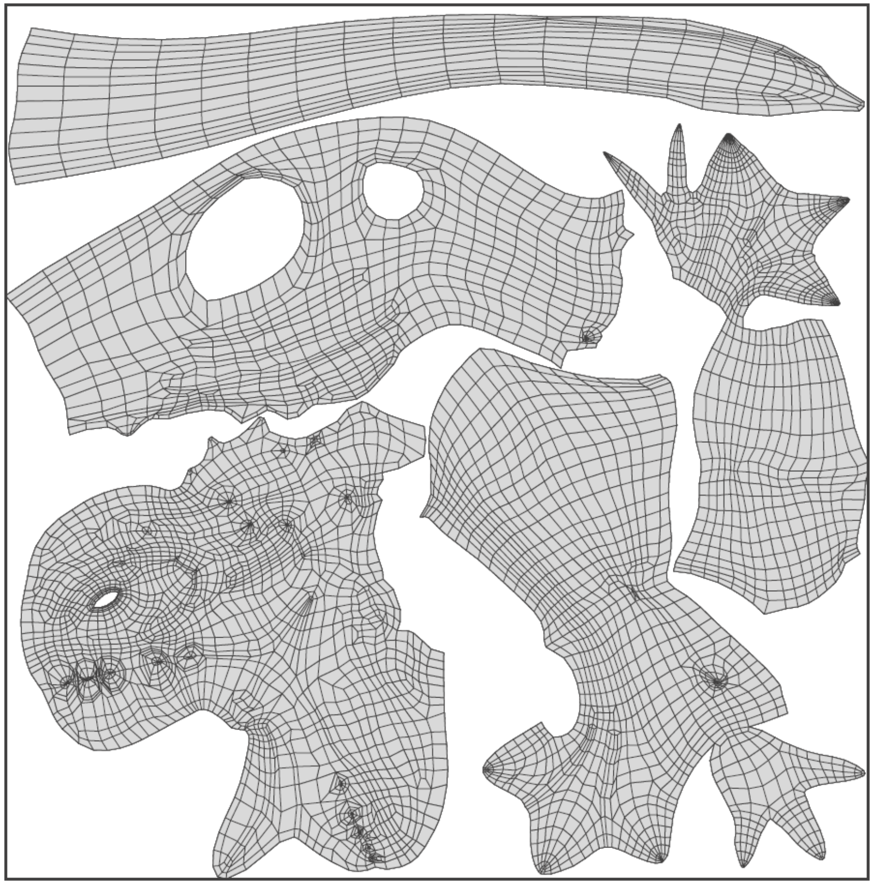


Parameterization



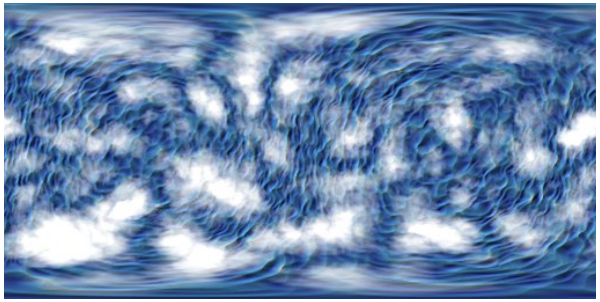
Parameterization

- **Texture Atlas**:
  not one single texture, but fragmented textures for object parts

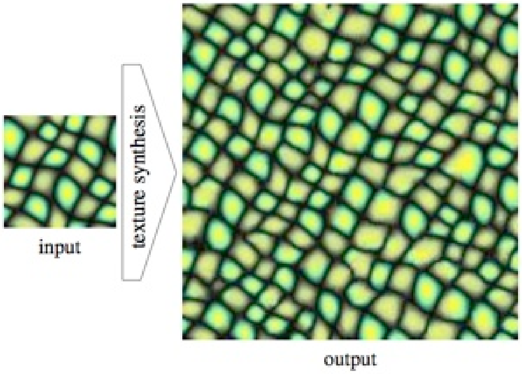


Texture Generation

- Textures can come from an image file, e.g. jpg
- or can be generated by a procedure
  → on the fly in a shader
  - often based on fractal noise or turbulence functions (see later)

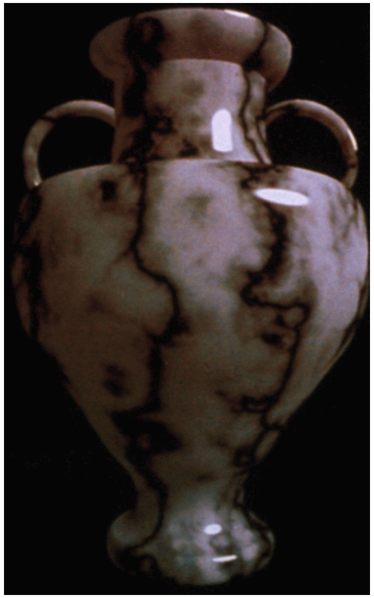


- → Texture synthesis: generate arbitrarily large high-quality texture from a small input sample.



Texture Generation

- Procedural texture generation
  - Computer generated texture image (1D, 2D or 3D) created using an algorithm.
  - Natural appearance through fractal noise, coherence and multi-scale representations, e.g. turbulence functions.



```
/* Copyrighted Pixar 1988 */
/* From the RenderMan Companion p. 355 */
/* Listing 16.19  Blue marble surface shader*/

/*
 * blue_marble(): a marble stone texture in shades of blue
 * surface
 */

blue_marble(
        float   Ks    = .4,
                Kd    = .6,
                Ka    = .1,
                roughness = .1,
                txtscale = 1;
        color   specularcolor = 1)
{
   point PP;            /* scaled point in shader space */
   float csp;           /* color spline parameter */
   point Nf;            /* forward-facing normal */
   point V;             /* for specular() */
   float pixelsize, twice, scale, weight, turbulence;

   /* Obtain a forward-facing normal for lighting calculations. */
   Nf = faceforward( normalize(N), I);
   V = normalize(-I);

   /*
    * Compute "turbulence" a la [PERLIN85]. Turbulence is a sum of
    * "noise" components with a "fractal" 1/f power spectrum. It gives the
    * visual impression of turbulent fluid flow (for example, as in the
    * formation of blue_marble from molten color splines!). Use the
    * surface element area in texture space to control the number of
    * noise components so that the frequency content is appropriate
    * to the scale. This prevents aliasing of the texture.
    */
   PP = transform("shader", P) * txtscale;
   pixelsize = sqrt(area(PP));
   twice = 2 * pixelsize;
   turbulence = 0;
   for (scale = 1; scale > twice; scale /= 2)
       turbulence += scale * noise(PP/scale);

   /* Gradual fade out of highest-frequency component near limit */
   if (scale > pixelsize) {
       weight = (scale / pixelsize) - 1;
       weight = clamp(weight, 0, 1);
       turbulence += weight * scale * noise(PP/scale);
   }

   /*
    * Magnify the upper part of the turbulence range 0.75:1
    * to fill the range 0:1 and use it as the parameter of
    * a color spline through various shades of blue.
    */
   csp = clamp(4 * turbulence - 3, 0, 1);
   Ci = color spline(csp,
   color (0.25, 0.25, 0.35),      /* pale blue        */
       color (0.25, 0.25, 0.35),  /* pale blue        */
       color (0.20, 0.20, 0.30),  /* medium blue      */
       color (0.20, 0.20, 0.30),  /* medium blue      */
       color (0.20, 0.20, 0.30),  /* medium blue      */
       color (0.25, 0.25, 0.35),  /* pale blue        */
       color (0.25, 0.25, 0.35),  /* pale blue        */
       color (0.15, 0.15, 0.26),  /* medium dark blue */
       color (0.15, 0.15, 0.26),  /* medium dark blue */
       color (0.10, 0.10, 0.20),  /* dark blue        */
       color (0.10, 0.10, 0.20),  /* dark blue        */
       color (0.25, 0.25, 0.35),  /* pale blue        */
       color (0.10, 0.10, 0.20)   /* dark blue        */
       );

   /* Multiply this color by the diffusely reflected light. */
   Ci *= Ka*ambient() + Kd*diffuse(Nf);

   /* Adjust for opacity. */
   Oi = Os;
   Ci = Ci * Oi;

   /* Add in specular highlights. */
   Ci += specularcolor * Ks * specular(Nf,V,roughness);
}
```

Textures

```
/* Copyrighted Pixar 1988 */
/* From the RenderMan Companion p. 355 */
/* Listing 16.19  Blue marble surface shader*/

/*
 * blue_marble(): a marble stone texture in shades of blue
 * surface
 */

blue_marble(
```

```
PP = transform("shader", P) * txtscale;
pixelsize = sqrt(area(PP));
twice = 2 * pixelsize;
turbulence = 0;
for (scale = 1; scale > twice; scale /= 2)
    turbulence += scale * noise(PP/scale);

/* Gradual fade out of highest-frequency component near limit */
if (scale > pixelsize) {
    weight = (scale / pixelsize) - 1;
    weight = clamp(weight, 0, 1);
    turbulence += weight * scale * noise(PP/scale);
}
```

```
        color (0.15, 0.15, 0.26),  /* medium dark blue */
        color (0.15, 0.15, 0.26),  /* medium dark blue */
        color (0.10, 0.10, 0.20),  /* dark blue        */
        color (0.10, 0.10, 0.20),  /* dark blue        */
        color (0.25, 0.25, 0.35),  /* pale blue        */
        color (0.10, 0.10, 0.20)   /* dark blue        */
        );

    /* Multiply this color by the diffusely reflected light. */
    Ci *= Ka*ambient() + Kd*diffuse(Nf);

    /* Adjust for opacity. */
    Oi = Os;
    Ci = Ci * Oi;

    /* Add in specular highlights. */
    Ci += specularcolor * Ks * specular(Nf,V,roughness);
}
```

Textures

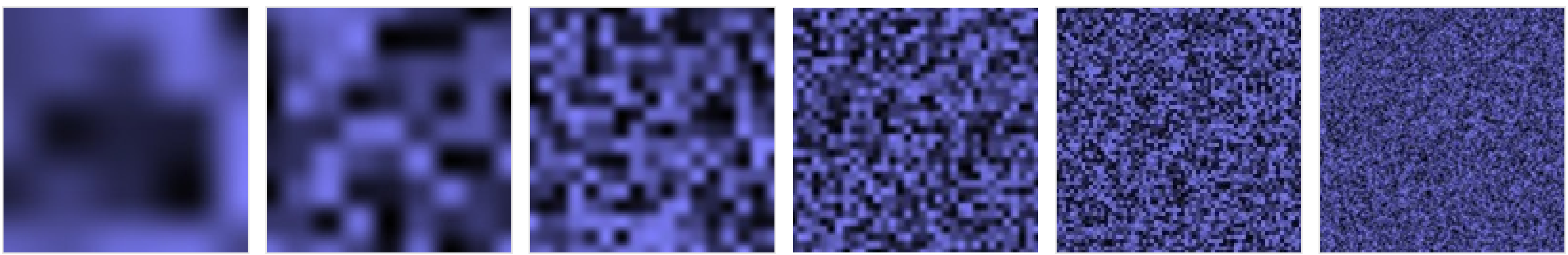
```
/* Copyrighted Pixar 1988 */
/* From the RenderMan Companion p. 355 */
/* Listing 16.19  Blue marble surface shader*/

/*
 * blue_marble(): a marble stone texture in shades of blue
 * surface
 */

blue_marble(
```

```
PP = transform("shader", P) * txtscale;
pixelsize = sqrt(area(PP));
twice = 2 * pixelsize;
turbulence = 0;
for (scale = 1; scale > twice; scale /= 2)
    turbulence += scale * noise(PP/scale);

/* Gradual fade out of highest-frequency component near limit */
if (scale > pixelsize) {
    weight = (scale / pixelsize) - 1;
    weight = clamp(weight, 0, 1);
    turbulence += weight * scale * noise(PP/scale);
}
```

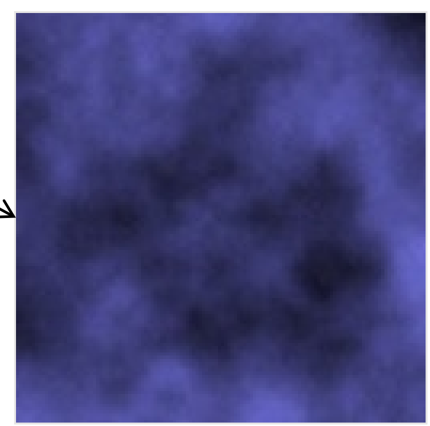
```
        color (0.15, 0.15, 0.26),  /* medium dark blue */
        color (0.15, 0.15, 0.26),  /* medium dark blue */
        color (0.10, 0.10, 0.20),  /* dark blue        */
        color (0.10, 0.10, 0.20),  /* dark blue        */
        color (0.25, 0.25, 0.35),  /* pale blue        */
        color (0.10, 0.10, 0.20)   /* dark blue        */
        );

    /* Multiply this color by the diffusely reflected light. */
    Ci *= Ka*ambient() + Kd*diffuse(Nf);

    /* Adjust for opacity. */
    Oi = Os;
    Ci = Ci * Oi;

    /* Add in specular highlights. */
    Ci += specularcolor * Ks * specular(Nf,V,roughness);
}
```

Procedural texture functions - example

- Noise and turbulence (Perlin noise with different frequency resolution “layers”)

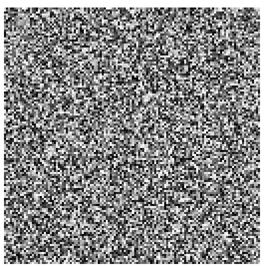


Sum of all layers

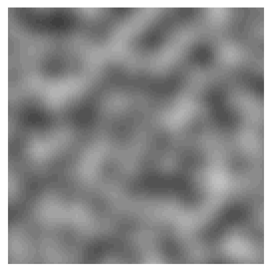


Texture functions

- Noise - “White noise”: Assign random color for every point
- “Perlin noise”
  - Method for generating coherent noise over space.
  - Coherent means: the function values change smoothly.



Non coherent

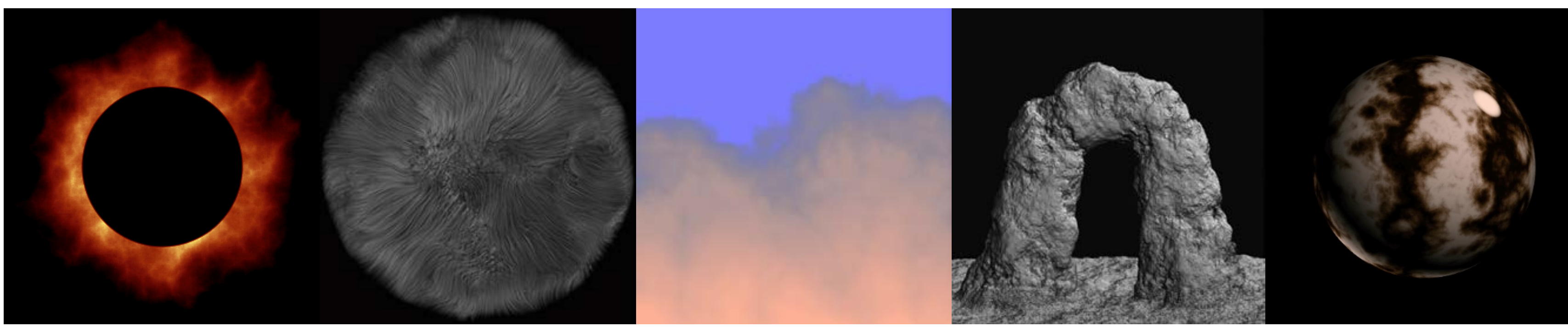


Coherent

Images by Matt Zucker

Texture functions

- Perlin noise
  - Solid texture
  - Based on gradient noise
    - Generate an n-dimensional lattice of random gradients
    - The noise value is interpolated in the lattice cells, e.g. using linear or cosine interpolation.
  - Gradient noise is conceptually different than value or wavelet noise.



http://www.noisemachine.com/talk1/

Texture functions

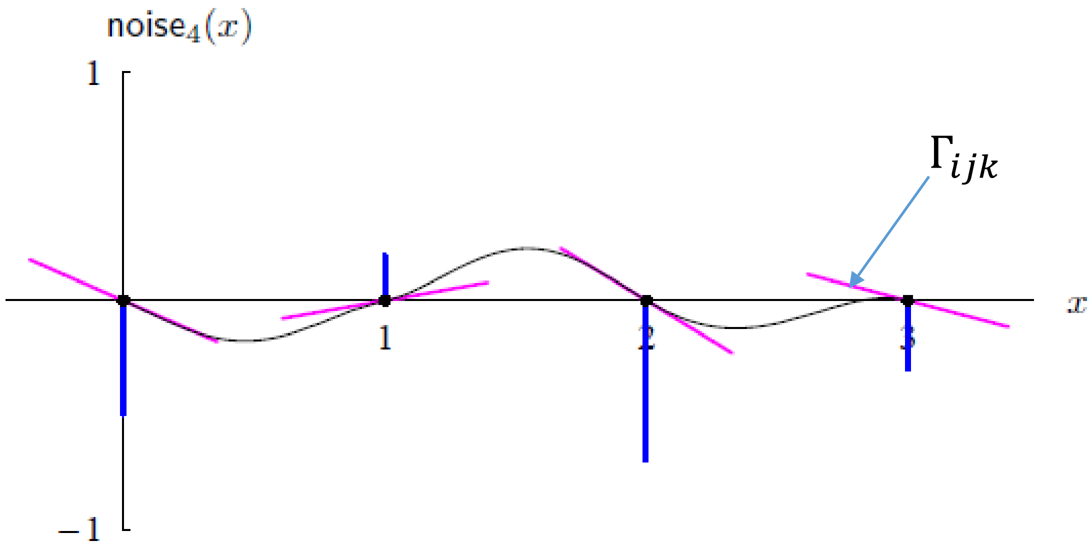
- Perlin Noise
  - At grid point (i, j, k) the gradient is Γ_{ijk}
  - Γ_{ijk} is determined from (i, j, k) using an array of precomputed random gradient values $G[]$ and a hash function $\phi()$ as:

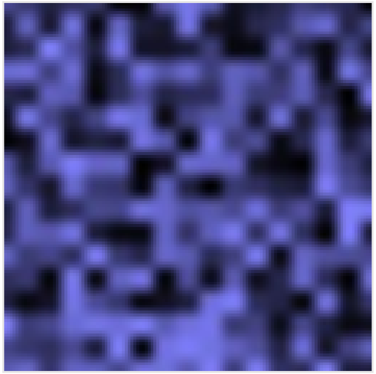
$$\Gamma_{ijk} = G\left(\phi\left(\mathrm{i} + \phi\big(j + \phi(k)\big)\right)\right)$$

  → „pseudorandom“ gradient values, fast to compute
  - Then, these grid point gradients are interpolated

Texture functions

- Perlin noise: at the grid points
  - the gradient is selected randomly
  - the function has value zero





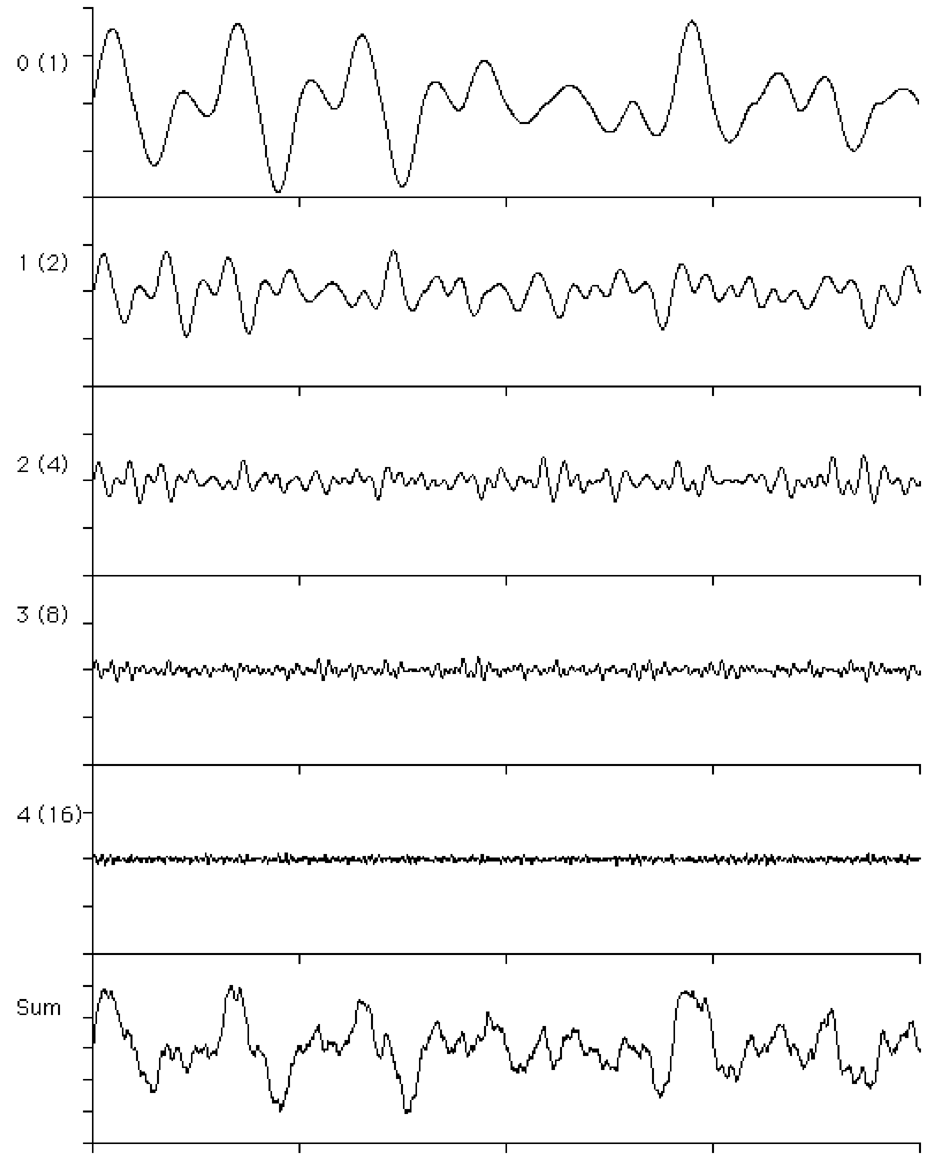
Texture functions

- Turbulence
  - Many natural textures contain repeating features of different sizes
  - Perlin pseudo fractal “turbulence” function
  - Effectively adds scaled copies of noise function on top of itself

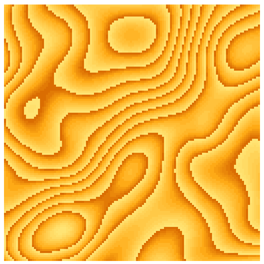
$$n_t(x) = \sum_i \frac{\left|n(2^i x)\right|}{2^i}$$

Texture functions

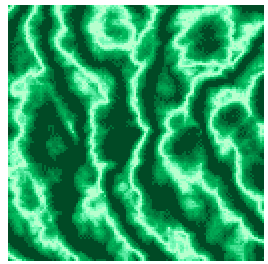
$$n_t(x) = \sum_i \frac{|n(2^i x)|}{2^i}$$



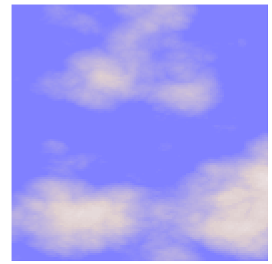
Texture functions: Perlin noise



wood



marble



clouds

Images by Matt Zucker

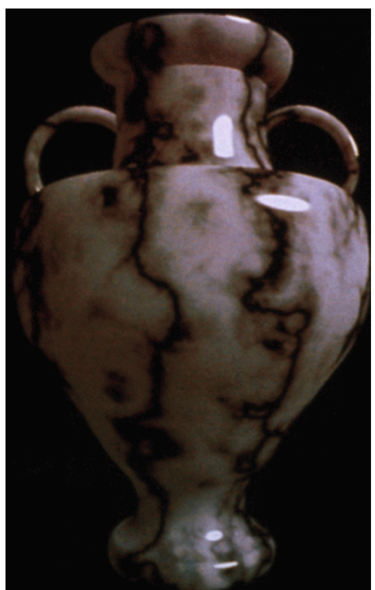


Image by Ken Perlin

Texture Functions

- Very recent paper: wood shader

http://flycooler.com/

bertramguitars.com

Texture Mapping

- Mapping in 2D:
  - Texture image of size (n_x, n_y)
  - Constraints on some architectures (powers of 2)
  - Texture coordinates “s” and “t” for accessing texture images
    → (s, t, r) in 3D and
    → (s, t, r, q) homogeneous texture coordinates

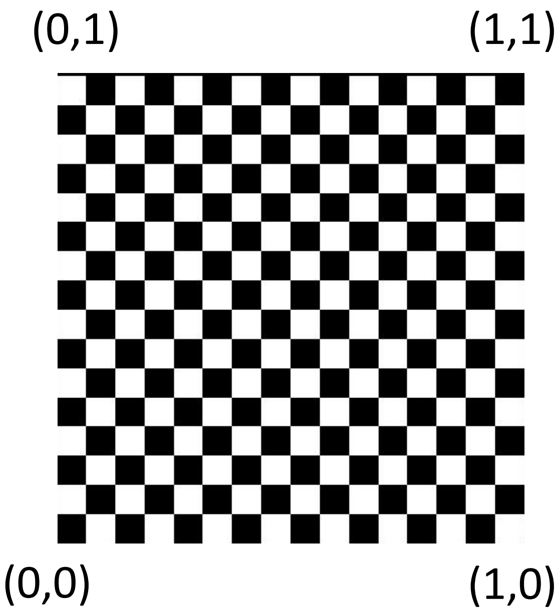
  - Assign to every geometric point (x, y, z) on the polygon $\mathbf{P}$ a texture coordinate (s, t):

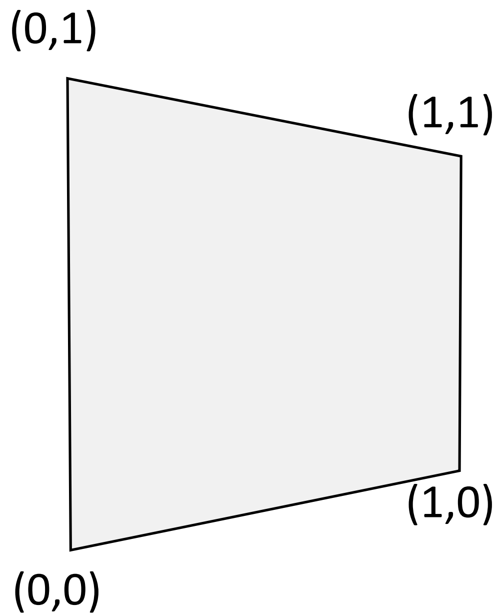
  → $F: P \in \mathbb{R}^3 \rightarrow [0,1]^2 \in \mathbb{R}^2$

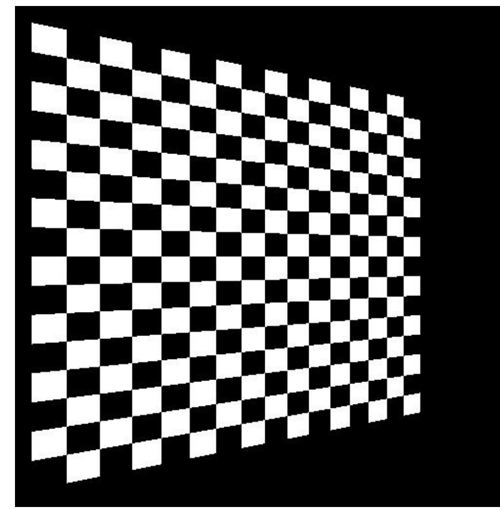
- Simple procedure:
  1. for every vertex assign (s, t).
  2. For interior points assign (s, t) by interpolation.

Texture Mapping

- Texture + Quad = Image



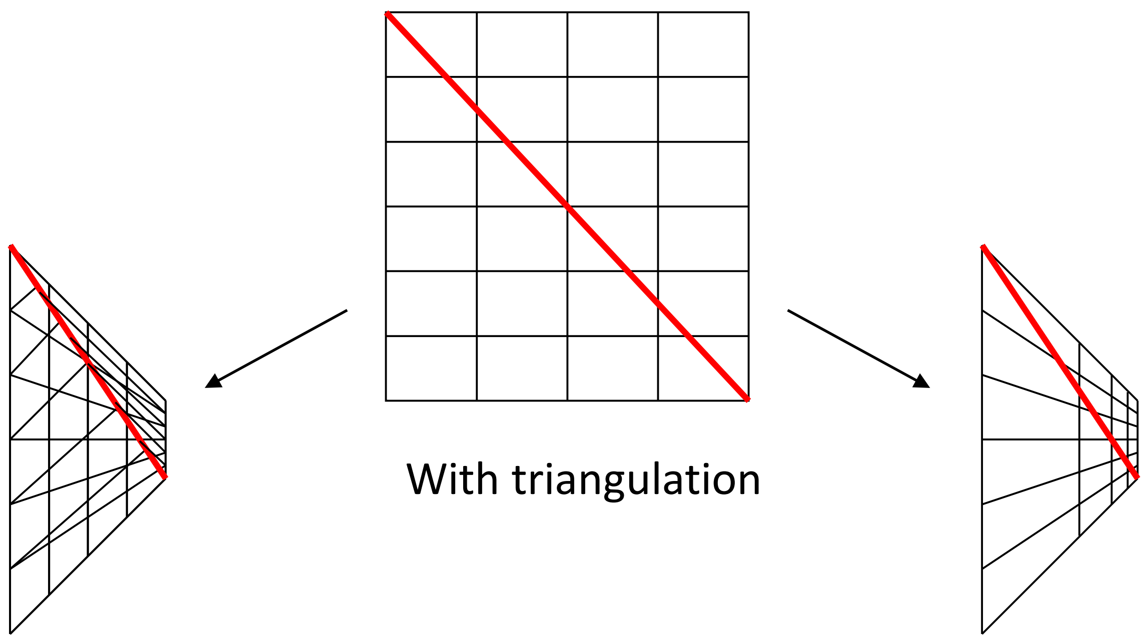




Texture Mapping for Rasterized Triangles

- Interpolation Problem
  - Standard interpolation method at rasterization stage (linear interpolation) results in distorted images!
  - Reason: Does not consider the distortion of the perspective transformation!

Original texture



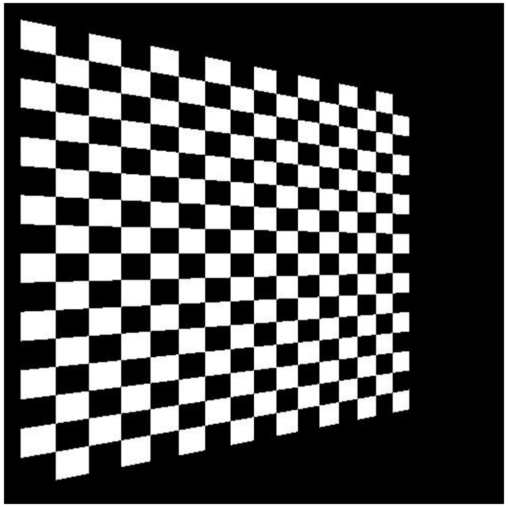
With triangulation

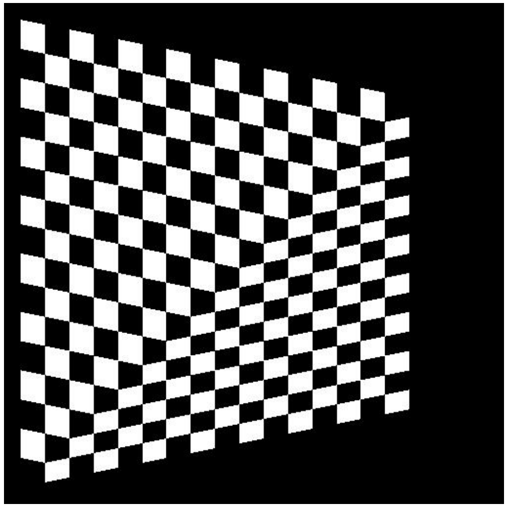
Interpolation using
barycentric coordinates

Perspective
interpolation

Texture Mapping for Rasterized Triangles

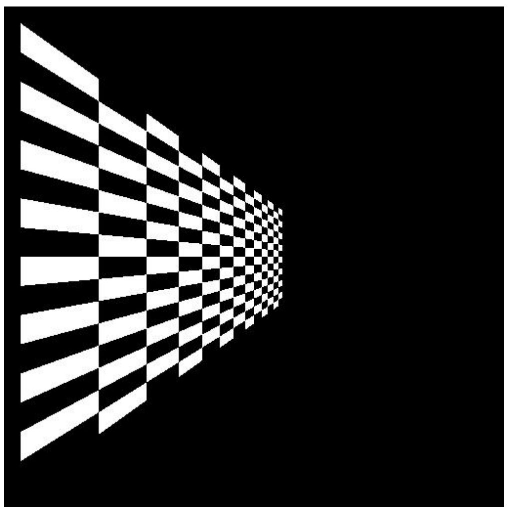
- Correct wrong

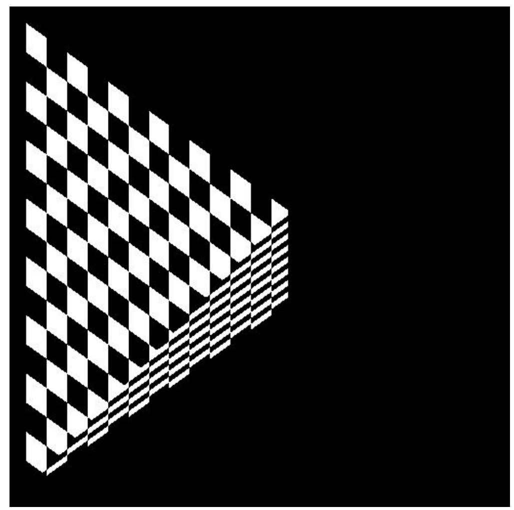




Texture Mapping for Rasterized Triangles

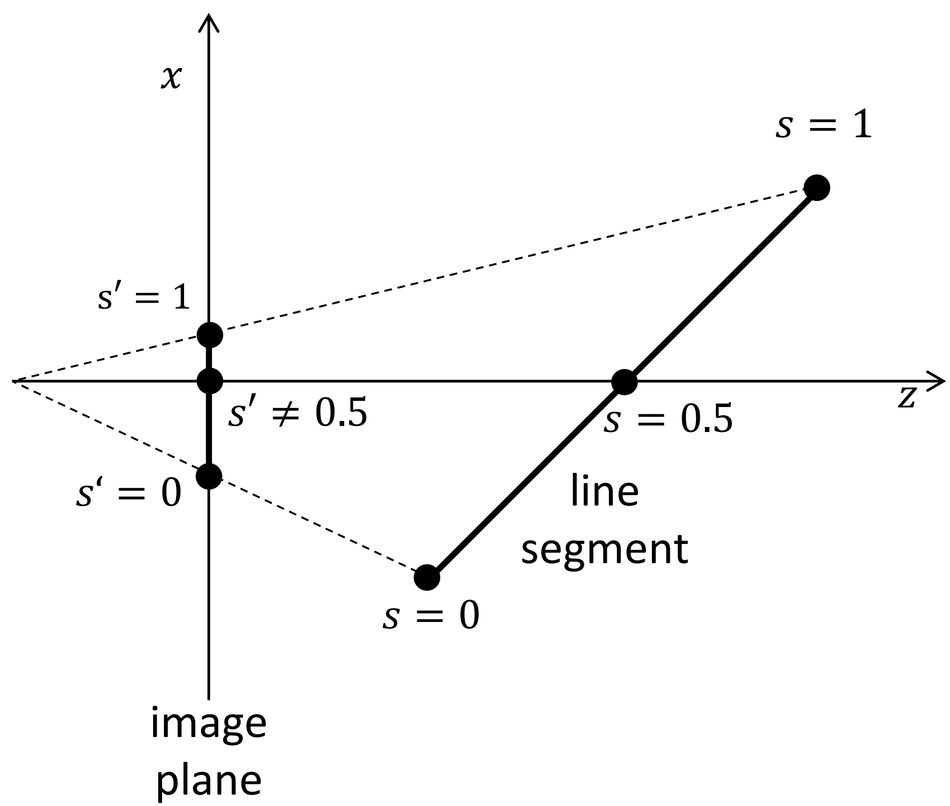
- Correct very wrong





Texture Mapping for Rasterized Triangles

- Perspective interpolation – problem statement
  - Example: line segment not parallel to image plane:
  - s: texture coordinate in world space, s’: texture coordinate in screen space
  - Linear interpolation of s’ in screen space does not match interpolation of s in worlds coordinates.



Texture Mapping for Rasterized Triangles

- Perspective Interpolation
  - Needed: Mapping $s' \rightarrow s$ that implements perspective correct linear interpolation in screen space
  - Solution: consider the division by z!
  - following derivation from http://www.comp.nus.edu.sg/~lowkl/publications/lowk_persp_interp_techrep.pdf

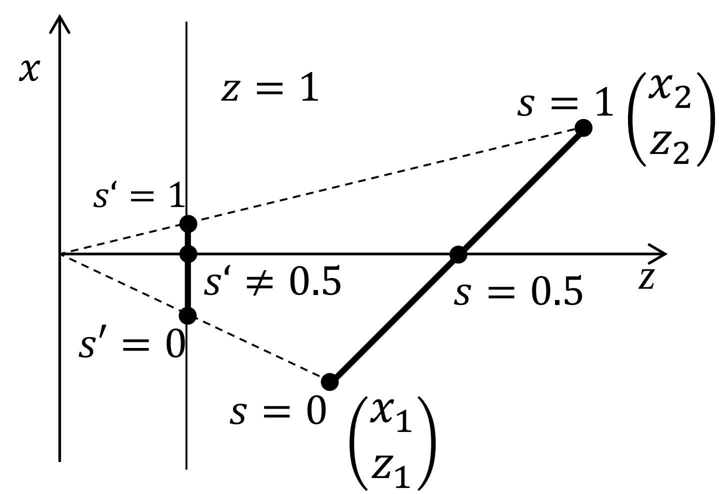
Texture Mapping for Rasterized Triangles

- s: relative position in world space, s‘ in image space
- In world space, we describe the line segment as:

$$\begin{pmatrix} x \\ z \end{pmatrix} = \begin{pmatrix} x_1 \\ z_1 \end{pmatrix} + s \begin{pmatrix} x_2 - x_1 \\ z_2 - z_1 \end{pmatrix}$$

- in image space:

$$x' = \frac{x_1}{z_1} + s' \left(\frac{x_2}{z_2} - \frac{x_1}{z_1} \right)$$

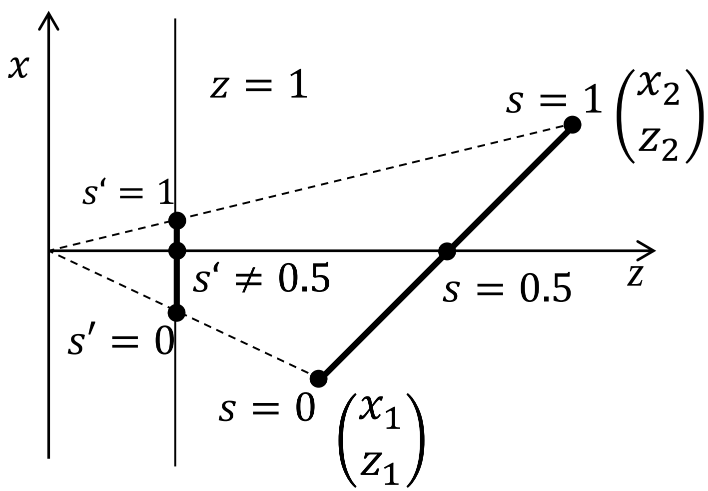


- Obviously s’ is not the same as s!

Texture Mapping for Rasterized Triangles

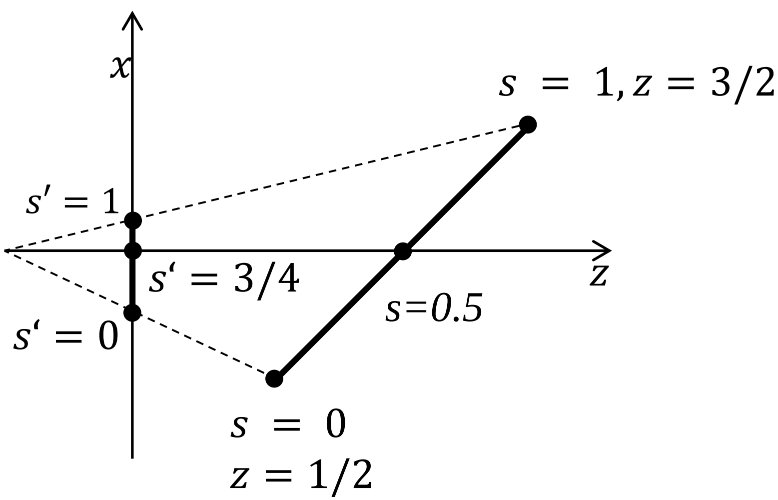
- During rasterization, we know s‘, and need to derive s from s‘
- with some arithmetics, we find

$$s = \frac{s' z_1}{s' z_1 + (1 - s') z_2}$$



Texture Mapping for Rasterized Triangles

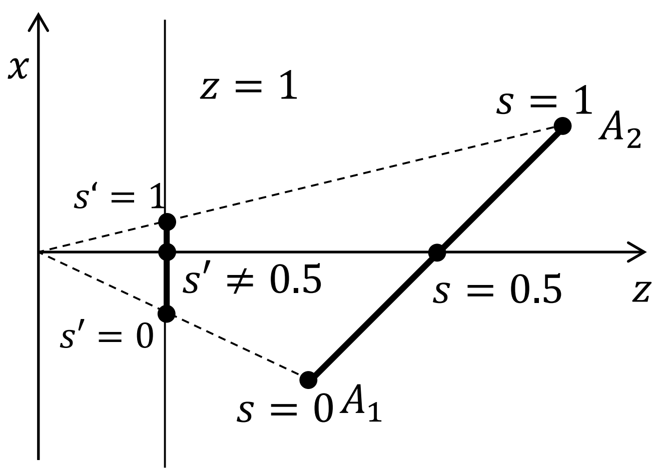
- Example
- $s' = \frac{3}{4} \to s = \frac{\frac{3}{4}z_1}{\frac{3}{4}z_1 + \frac{1}{4}z_2} = \frac{1}{2}$



Texture Mapping for Rasterized Triangles

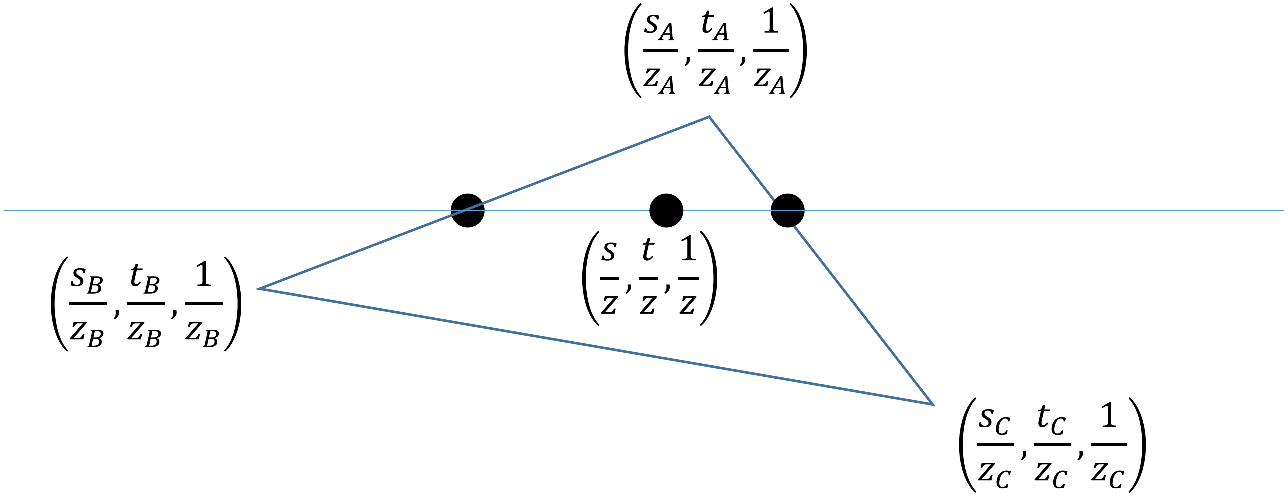
- so if we want to interpolate an attribute A along a line
- with z-values z_1 and z_2
- and attribute values A_1 and A_2
- using the image space relative position s‘
- we have to compute

$$A(t) = A_1 + s(A_2 - A_1) = \frac{\frac{A_1}{z_1} + s'\left(\frac{A_2}{z_2} - \frac{A_1}{z_1}\right)}{\frac{1}{z_1} + s'(\frac{1}{z_2} - \frac{1}{z_1})}$$



Texture Mapping for Rasterized Triangles

- New approach for interpolating texture coordinates for rasterization
  - interpolate s/z, t/z, and $1/z$ during rasterization
  - Per pixel: $(s/z)/(1/z), (t/z)/(1/z) \rightarrow (s, t)$



Texture Mapping

- In OpenGL:
  - 1D, 2D and 3D textures
  - textures can have luminance only (grey value), color, or color plus alpha (see next week)
  - 8bit per channel, 16bit per channel or float values
  - are sampled in a shader using a **sampler** object
  - homogeneous texture coordinates (s, t, r, q)

Next Lecture

- How to interpolate textures
- Texture Aliasing