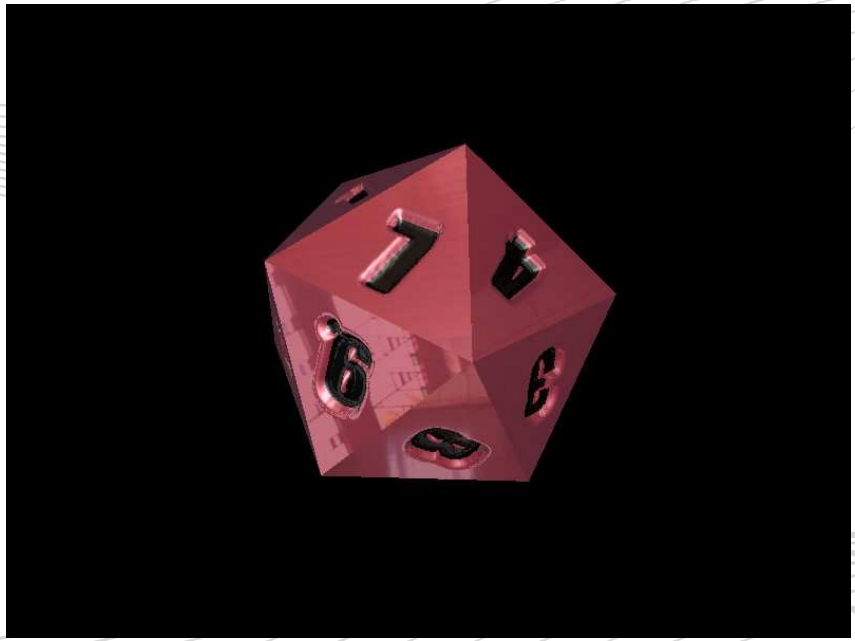
Textures and mapping techniques

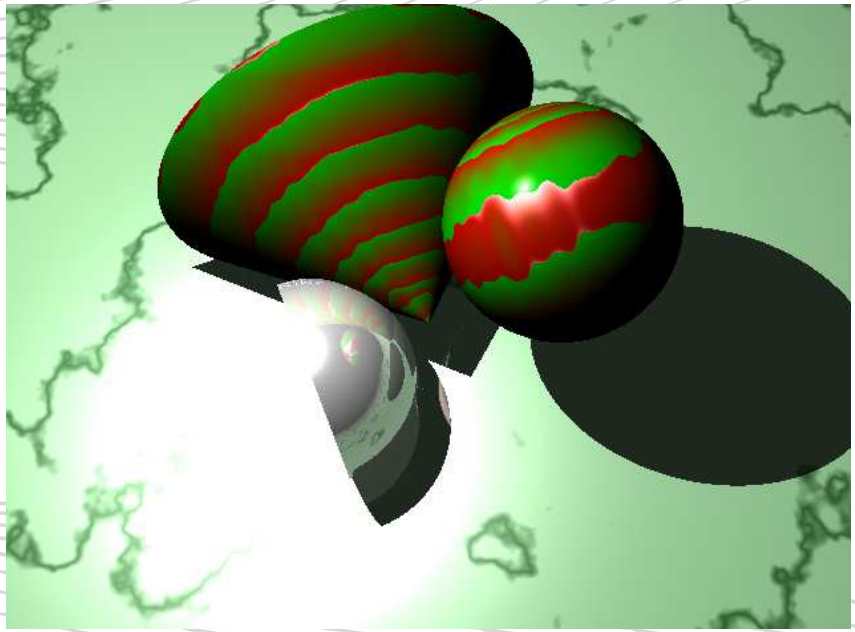
or, tricks you can do with images

Computer Graphics (DT3025)

Martin Magnusson

November 8, 2016



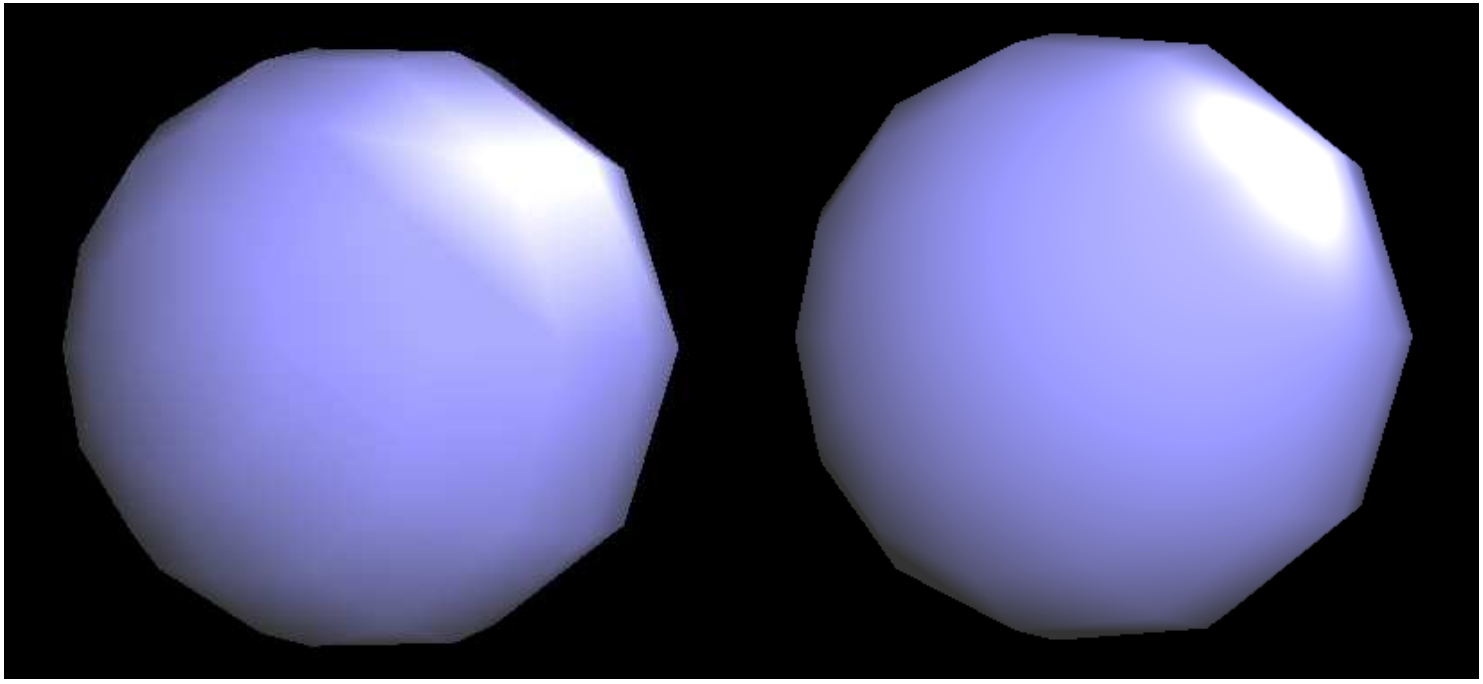


Last time

- Diffuse vs glossy reflections (Lambertian, specular)
- Phong's model (and Blinn–Phong)
- BRDFs
  - What *fraction* of light from direction ω_i goes to ω_o?
  - (Output intensity also depends on cosine with surface normal.)

(Gouraud and Phong shading in a shader context)

- Gouraud shading: compute BRDF in vertex shader, fragment shader uses interpolated colours.
- Phong shading: fragment shader gets interpolated positions and normals from vertex shader, compute BRDF per pixel.



Mathias Broxvall

Gouraud Phong

Today

- *Spatially varying* BRDFs (a k a textures).
- Other mapping techniques: normal mapping, displacement mapping, environment mapping, light mapping.
- Sampling and aliasing.

Reading material

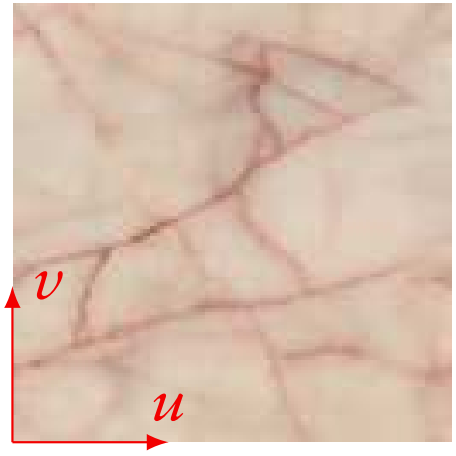
- Hughes et al.,
  - 7.9–7.9.1,
  - 9.6,
  - 20.1–20.8.2.

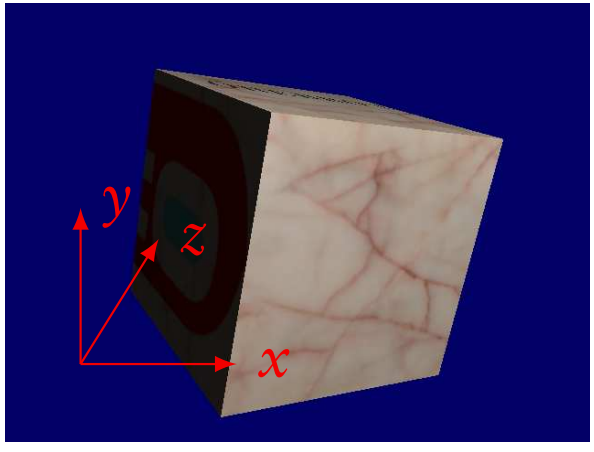
Where's the typo in Chp 9.6?

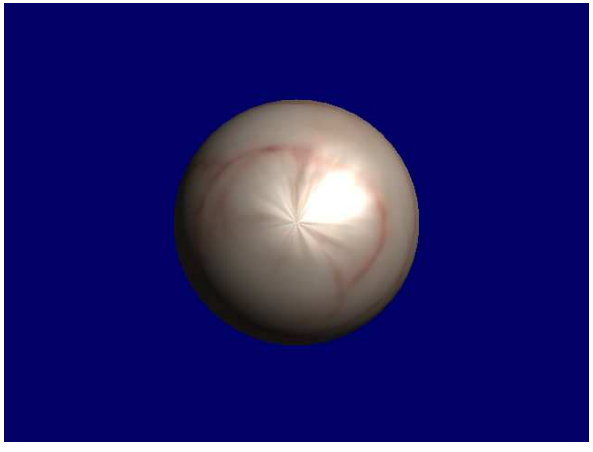
$$P = \alpha A + \beta B + \gamma \cancel{B} C \qquad (9.24)$$

Texture mapping

- Pulling a 2D texture image onto a 3D object.
  - (In other words: spatial variation of BRDF parameters.
- How to map 2D to 3D?
  - 2D texture: (u, v) on $[(0, 0), (1, 1)]$ vs
  - 3D object: (x, y, z) on infinite interval.
  - Sometimes referred to as uv-mapping.







How to use textures

- Can have different textures for colour, roughness, *or any other attribute*.
  - Example: diffuse colour

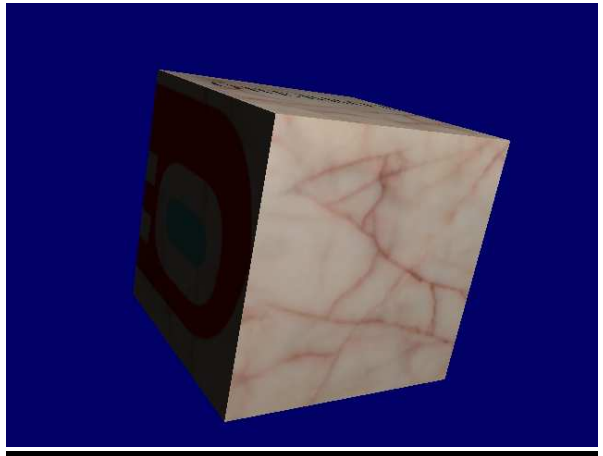
$$C = C_a I_a + \sum_I C_d I_d \cos\theta$$

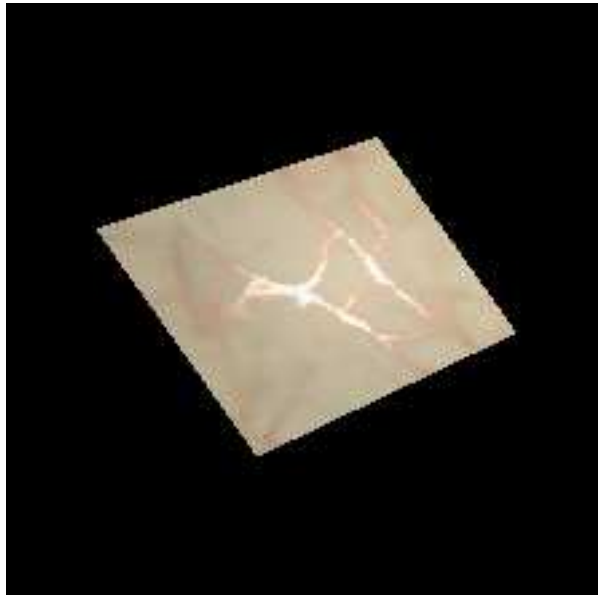
$$C = C_a I_a + \sum_I \text{texture}_d[u][v] I_d \cos\theta$$

  - Example: specular exponent

$$\sum_I C_s I_s (\omega_{r,I} \cdot v)^f$$

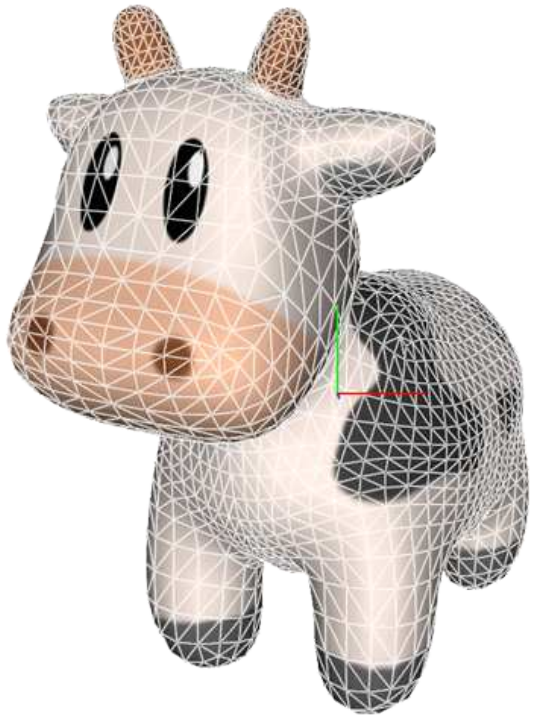
$$\sum_I C_s I_s (\omega_{r,I} \cdot v)^{\text{texture}_f[u][v]}$$



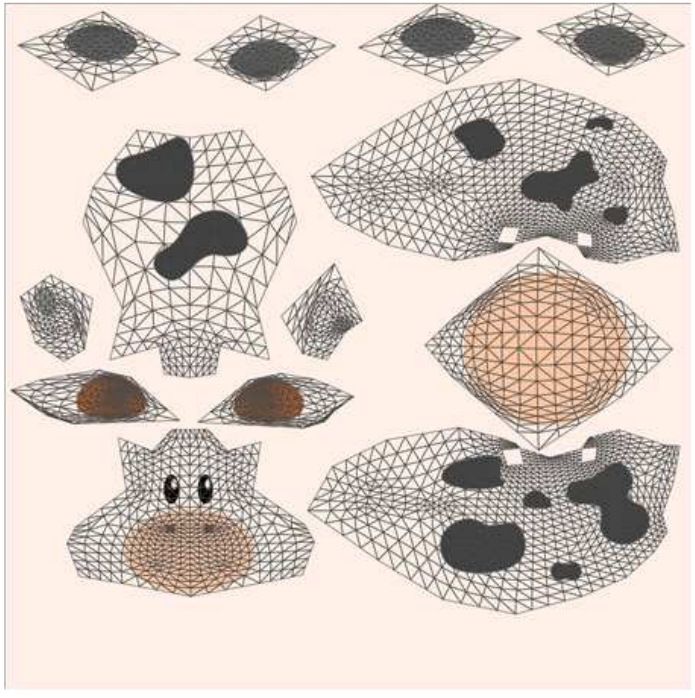


Mapping from texture to object

- We need a function that maps each 2D coordinate in the texture to a 3D surface point.



⇐



Warren Moore

Mapping vs sampling

- A mapping function: pull texture coordinates onto vertices of an object.

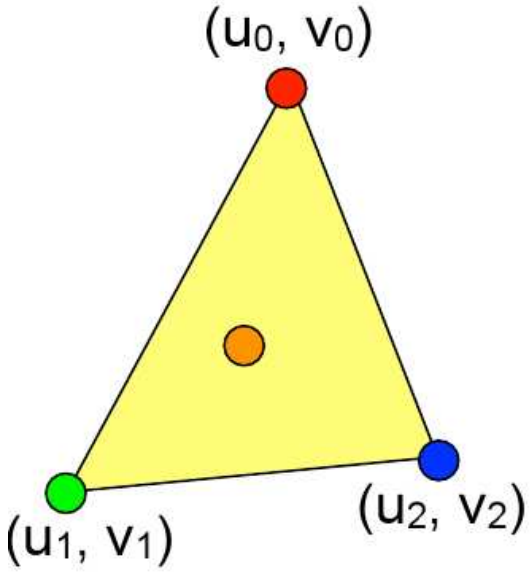
$$[u, v] = f(x, y, z)$$

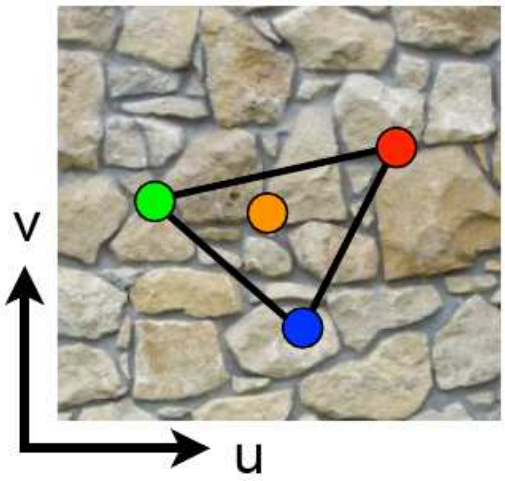
- A sampling function: find/compute/guess actual color of pixel using *texel* coordinates:

$$[r, g, b] = f(u, v)$$

uv coordinates

- Add vertex attribute: *uv* texture coordinates.
  - Determines the 2D location in the texture.
- Interpolate to find correct *texel* for each *fragment*.





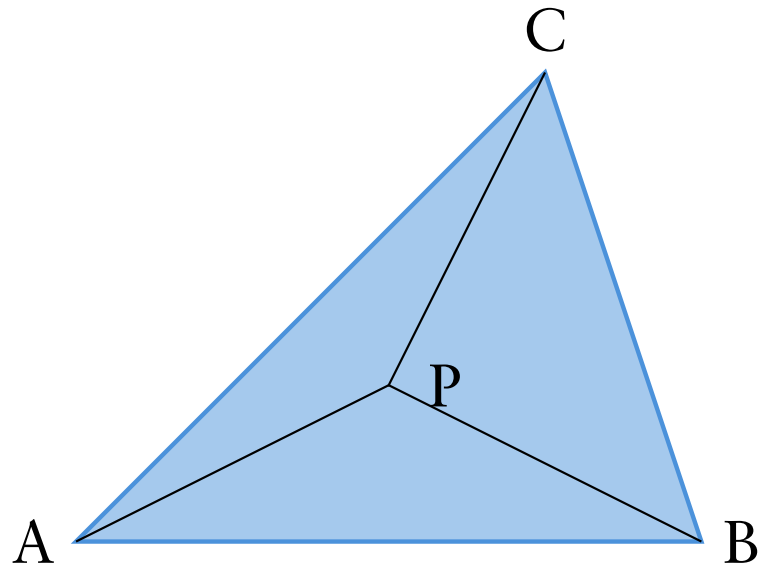
Jaakko Lehtinen

Barycentric coordinates

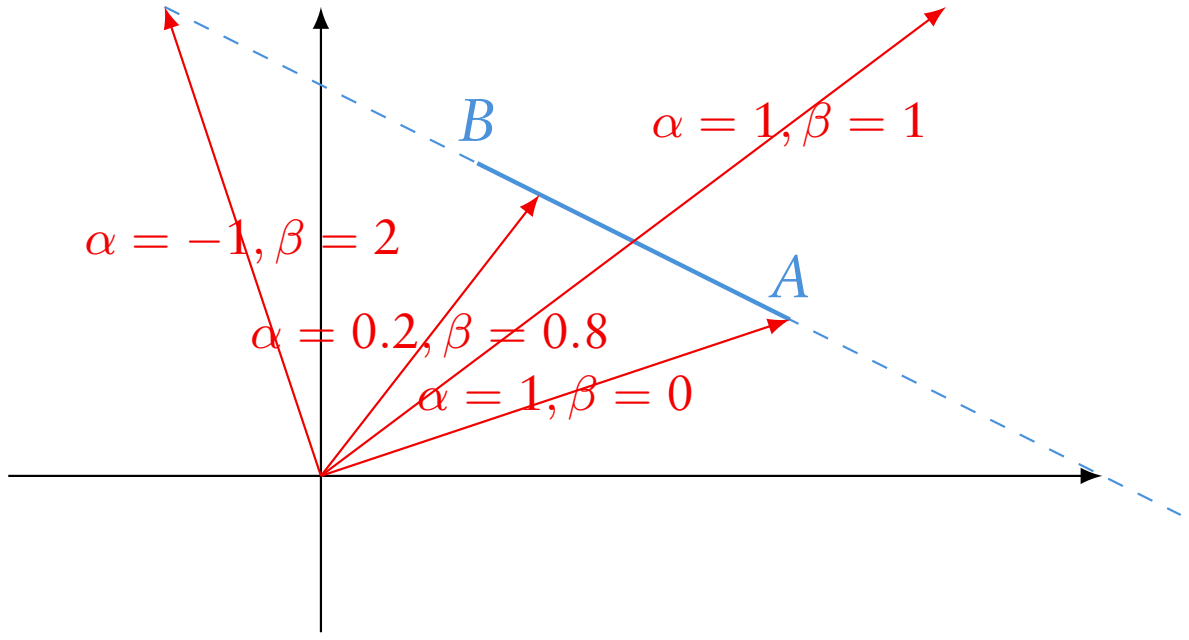
- Given vertex values A, B, C, what is the interpolated value of an interior point P?
- Linear combination:

$$P = \alpha A + \beta B + \gamma C.$$

- α, β, γ are the barycentric coordinates of P w r t the triangle ABC.
- If P is inside triangle, we require
  1. $\alpha + \beta + \gamma = 1$
  2. $\alpha, \beta, \gamma \geq 0$



Barycentric coordinate constraints illustrated



- $P = \alpha A + \beta B$

1. $\alpha + \beta = 1$: we are in the triangle's plane
2. $\alpha, \beta \geq 0$: we are inside the triangle

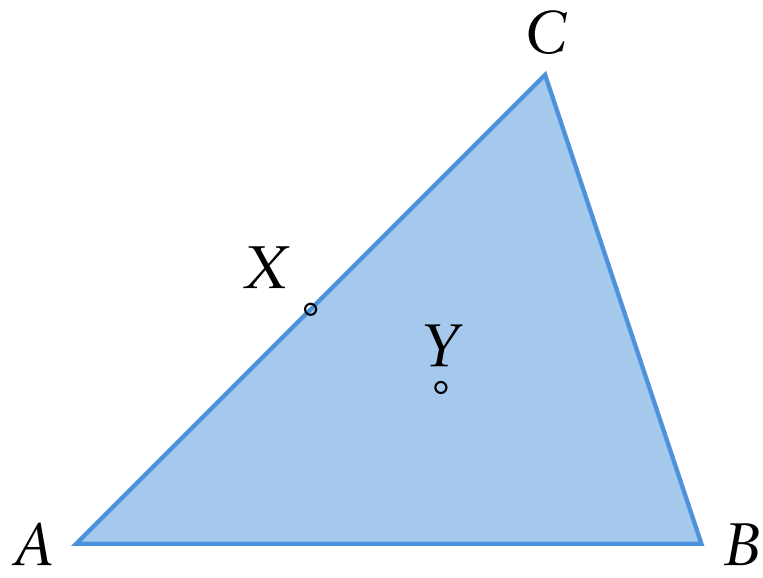
What are the barycentric coordinates of X and Y?

- X is the midpoint on the line AC.
- Y is the triangle's centroid.

$$X = \alpha_X A + \beta_X B + \gamma_X C$$

$$Y = \alpha_Y A + \beta_Y B + \gamma_Y C$$

What are $\alpha_{\{X,Y\}}$, $\beta_{\{X,Y\}}$, $\gamma_{\{X,Y\}}$?



How to compute α, β, γ?

- Given 3D *Cartesian* coordinates of P, and a triangle ABC to which it belongs, what are P's *barycentric* coordinates?
- System of linear equations:

$$\begin{bmatrix} A_x & B_x & C_x \\ A_y & B_y & C_y \\ A_z & B_z & C_z \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} = \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix}$$

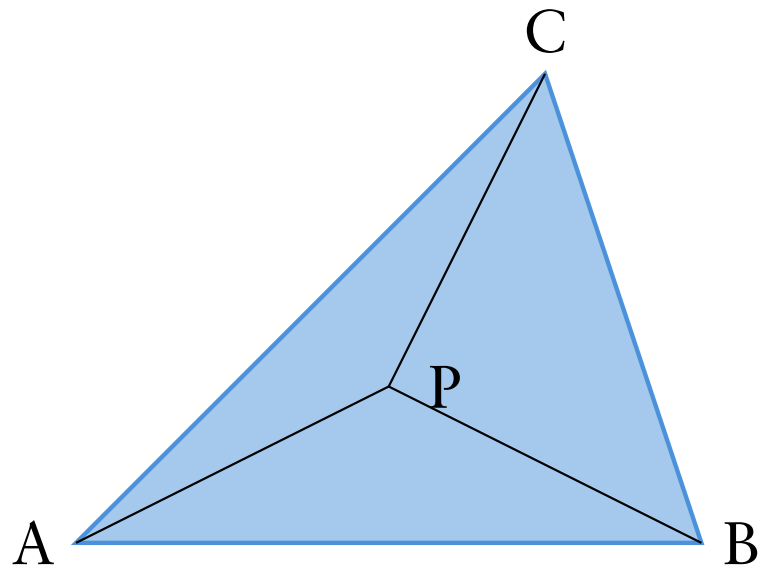
- . . . but we also require that $\alpha + \beta + \gamma = 1$:

$$\begin{bmatrix} A_x & B_x & C_x \\ A_y & B_y & C_y \\ A_z & B_z & C_z \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} = \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix}$$

- Overdetermined system! (4 equations, 3 unknowns.)
- But since we *know* that P is in the plane of ABC, last line is redundant (theoretically).

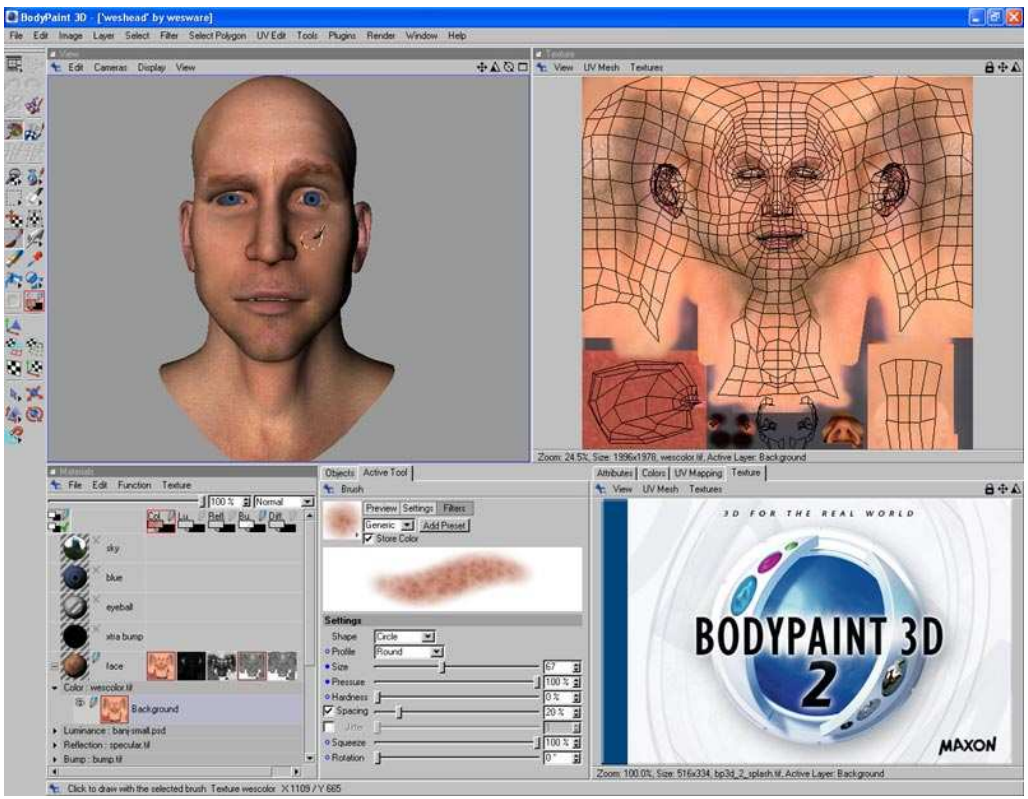
What can we interpolate?

- *xyz* spatial coordinates
- *rgb* colours
- *uv* texture coordinates
- normals
- *any vertex attribute*



How to assign vertex attributes?

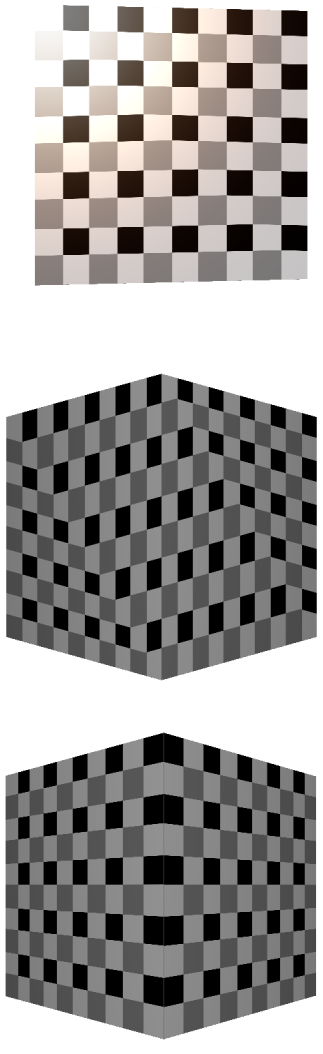
- OK, so we can interpolate between vertex values over the triangle using barycentrics.
  - And these interpolated values are given by OpenGL to your fragment shader.
- But how do we get the vertex values to begin with?



Maxon BodyPaint 3D

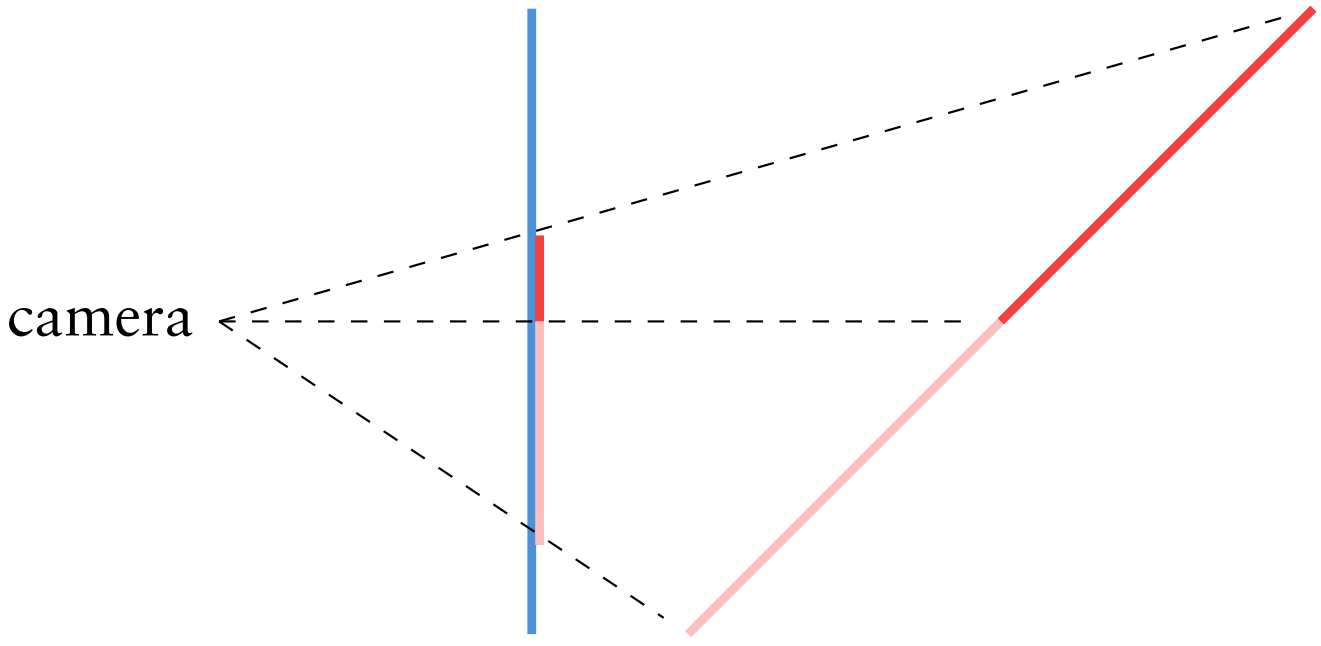
Perspective correction

- We've done *linear interpolation* using barycentric coordinates.
- But perspective transformation is not a linear transformation.
- Linear interpolation ("Gouraud shading") of *uv* coordinates leads to artifacts.



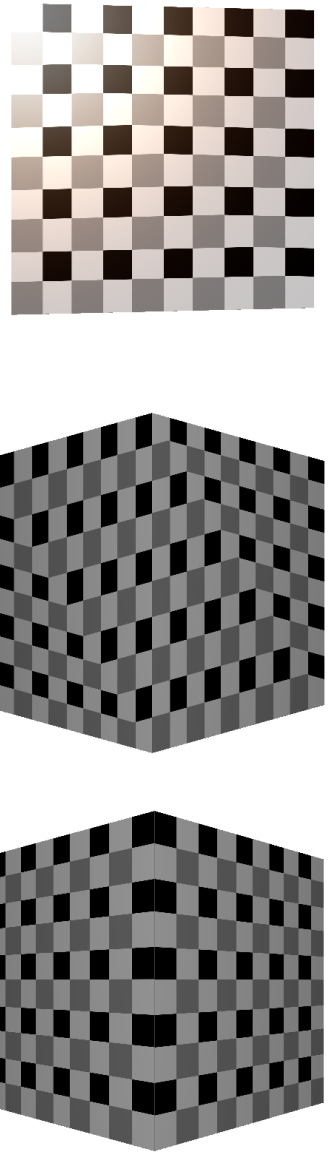
Perspective correction

- Linear variation in world space is not linear in screen space!

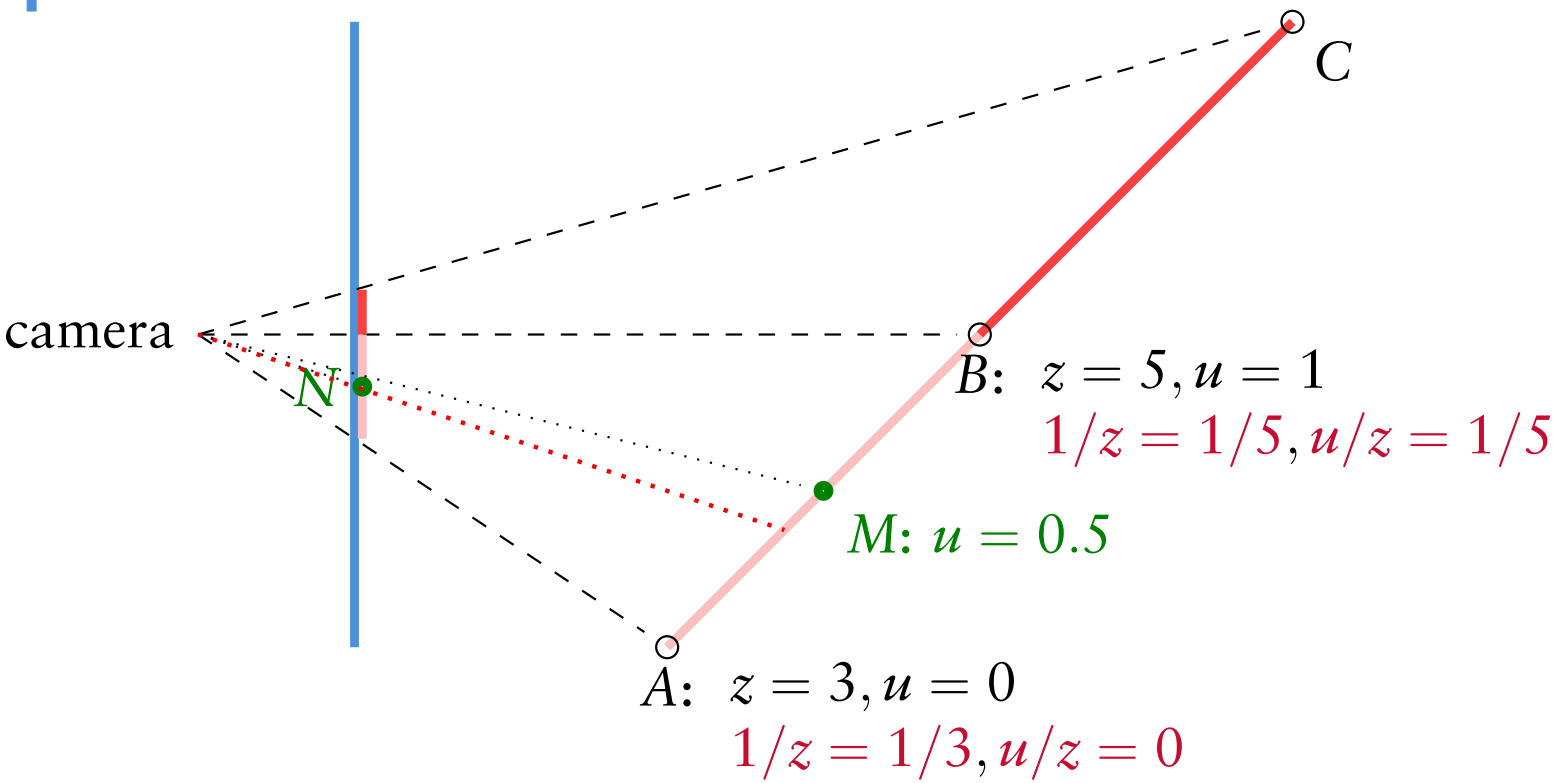


Perspective correction

- However:
  - $1/z$ is linear in screen space.
  - uv/z is linear in screen space.
  - $uv = \frac{uv/z}{1/z}$
- Interpolate the three values $u/z, v/z, 1/z$.
- For each pixel: compute $[u, v]$ from interpolated version of them.
- (OpenGL does this for you by default, for all `in` variables to fragment shader.)



Perspective correction



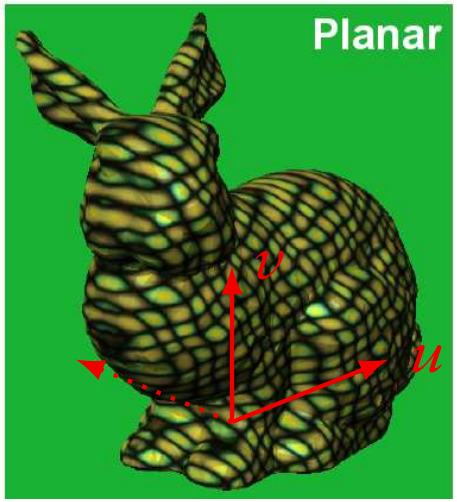
- M: midpoint of A and B. N: midpoint of projections.
- Interpolated u at N is not 0.5.
- Interpolated u/z: $\frac{0+1/5}{2} = 1/10$.
- Interpolated $1/z$: $\frac{1/3+1/5}{2} = 8/30$.
- Interpolated u at N: $\frac{1/10}{8/30} = 3/8$.

Parametric texture mapping

- What if we have non-mesh geometry?
- Solid objects: spheres etc.
- No vertices, so can't specify *uv* coordinates that way.
- Apply another, parametric, mapping instead.

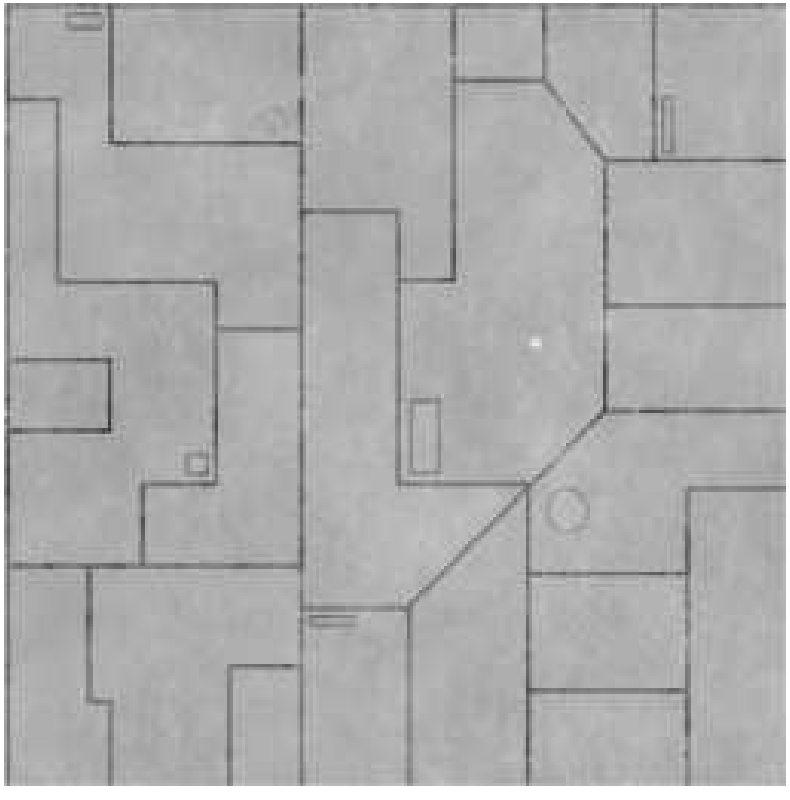
Example of other parametric mappings

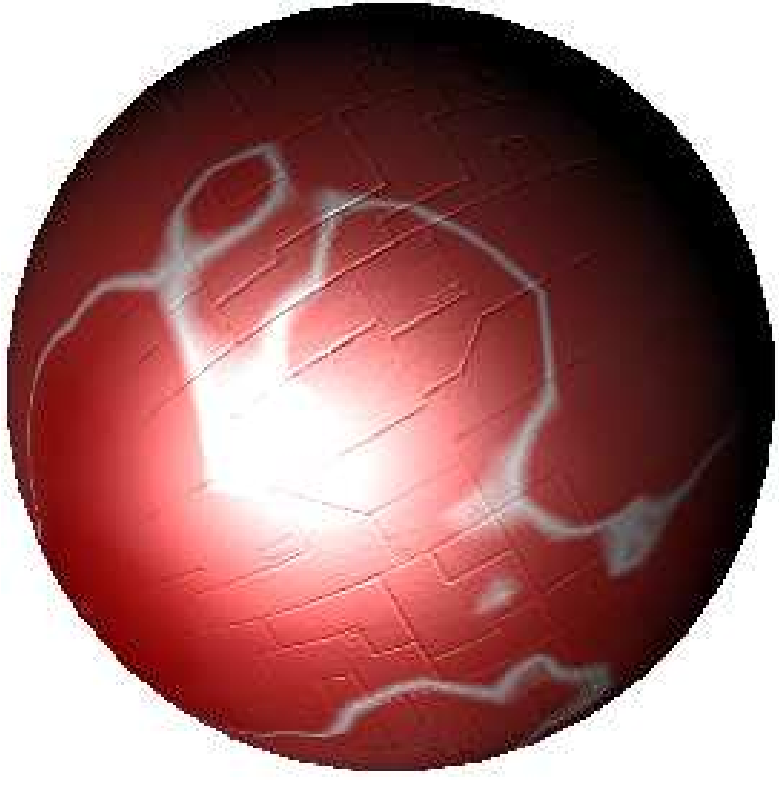
- Cylindrical
  - Assign main axis, project from axis through object.
  - 2D cylindrical coordinates map to texture coordinates.
- Spherical
  - Similarly: pick central point, project (2D spherical coordinates) from this point.
- Planar
  - Map *xy* coordinates to *uv*: object is "carved" from "texture volume".



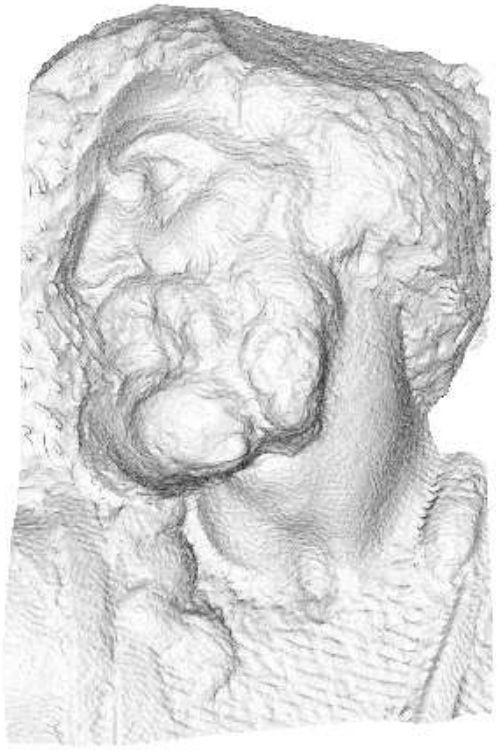
Bump maps

- Bumps maps / normal maps.
- Create surface detail without adding more polygons.

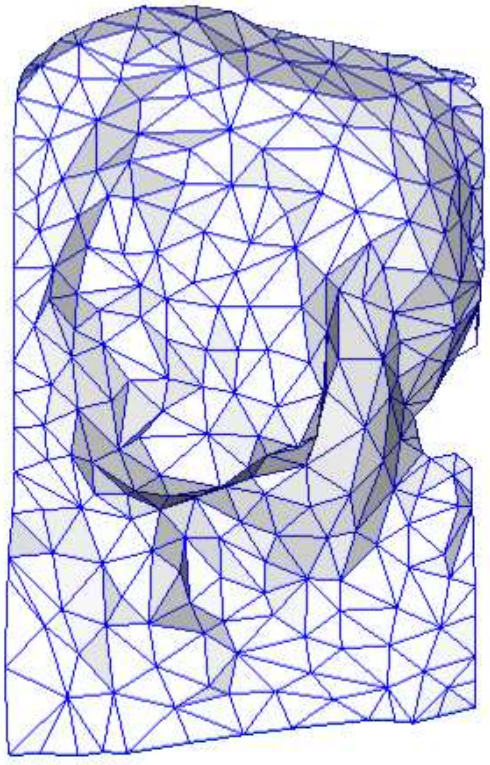




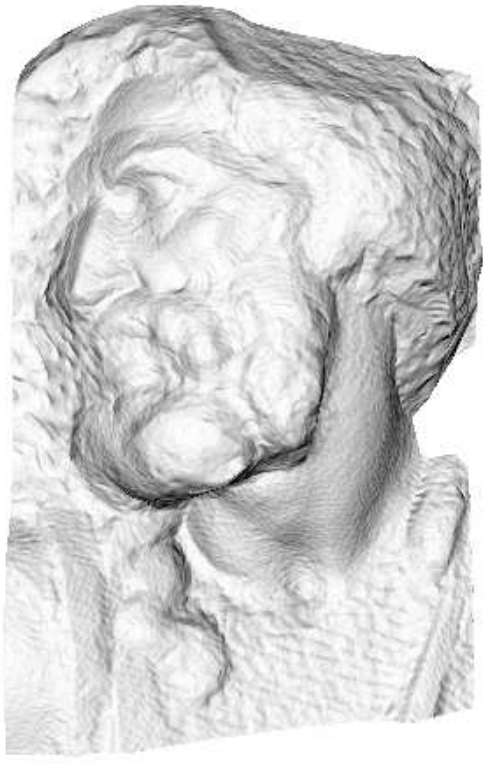
Example



original mesh
4M triangles



simplified mesh
500 triangles



simplified mesh
and normal mapping
500 triangles

Paulo Cignoni, via Wikipedia

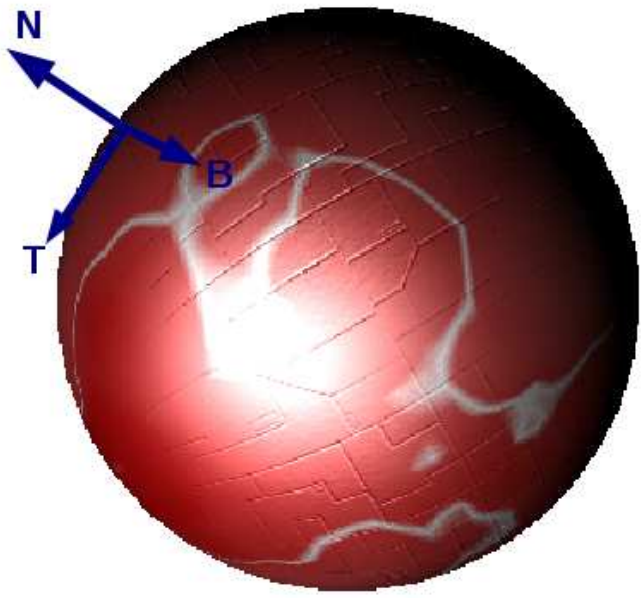
How to get bumps from texture

1. Store normal (in object space) explicitly as 3D coordinates (using *rgb* for *xyz*).
   - Just make sure that it points where you want (and is normalised).
2. Store *tilt offset* of normal: e g, using r and g component of texture.
   - $r > 0.5$: tilt normal vector in $+u$ direction,
   - $g < 0.5$: tilt normal vector in $-v$ direction.
   - Just need two values per texel.
   - Never invalid normal.
3. Store the *height* of each texel.
   - Use texture's $\frac{\mathrm{d}}{\mathrm{d}u}$ and $\frac{\mathrm{d}}{\mathrm{d}v}$ to tilt normal.
   - Just need one value per texel (saves memory),
   - but need to compute the normal (by comparing neighbours) rather than just looking it up.

Tangent/binormal

If you don't store explicit normals, then how to compute the final normal in view space?

- Two vectors, **t** and **b**, orthogonal to **n**.
- Aligned in same direction as bumpmap texture.
- **t**: "points along texture *u*"
- **b**: "points along texture *v*"

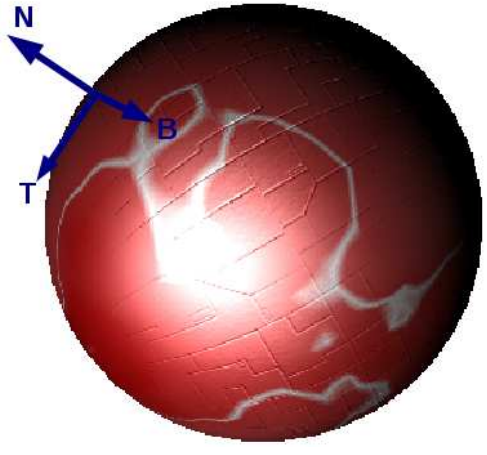


Computing the final normal

1. Transform $\mathbf{t}$ and $\mathbf{b}$ to view coordinates (vertex shader).
2. Compute bumpmap derivative (if using flavour 3) or get it from texture (if using flavour 2). (Now in fragment shader).
3. Normal offset

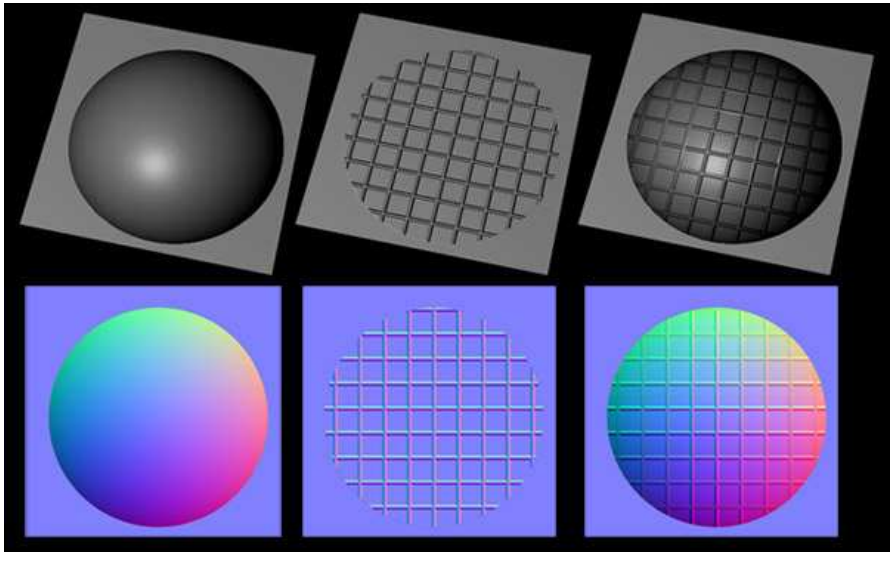
$$\mathbf{n}' = \mathbf{n} + \frac{\mathrm{d}\,\mathrm{texture}}{\mathrm{d}u} \cdot \mathbf{b} + \frac{\mathrm{d}\,\mathrm{texture}}{\mathrm{d}v} \cdot \mathbf{t}$$

4. Normalise: $\mathbf{n} \leftarrow \mathbf{n}'/\|\mathbf{n}'\|$.
5. Apply lighting model.

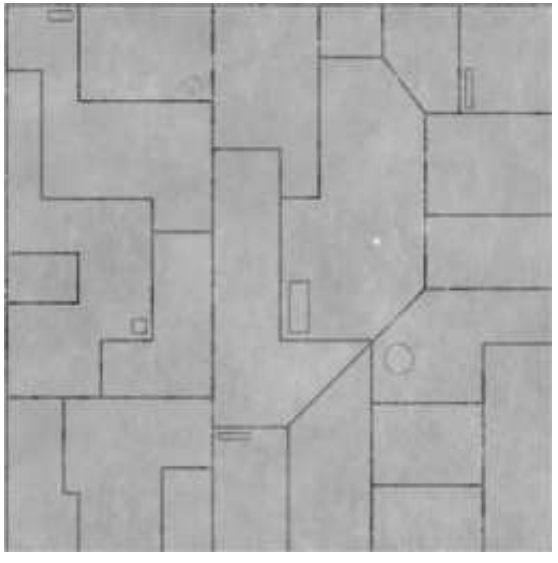


What kind of bump map is this?

1. Explicit 3D normal vector.
2. Tilt offset in r, g coordinates.
3. Pixel heights: tilt normal by $\frac{\mathrm{d}}{\mathrm{d}u}$, $\frac{\mathrm{d}}{\mathrm{d}v}$.



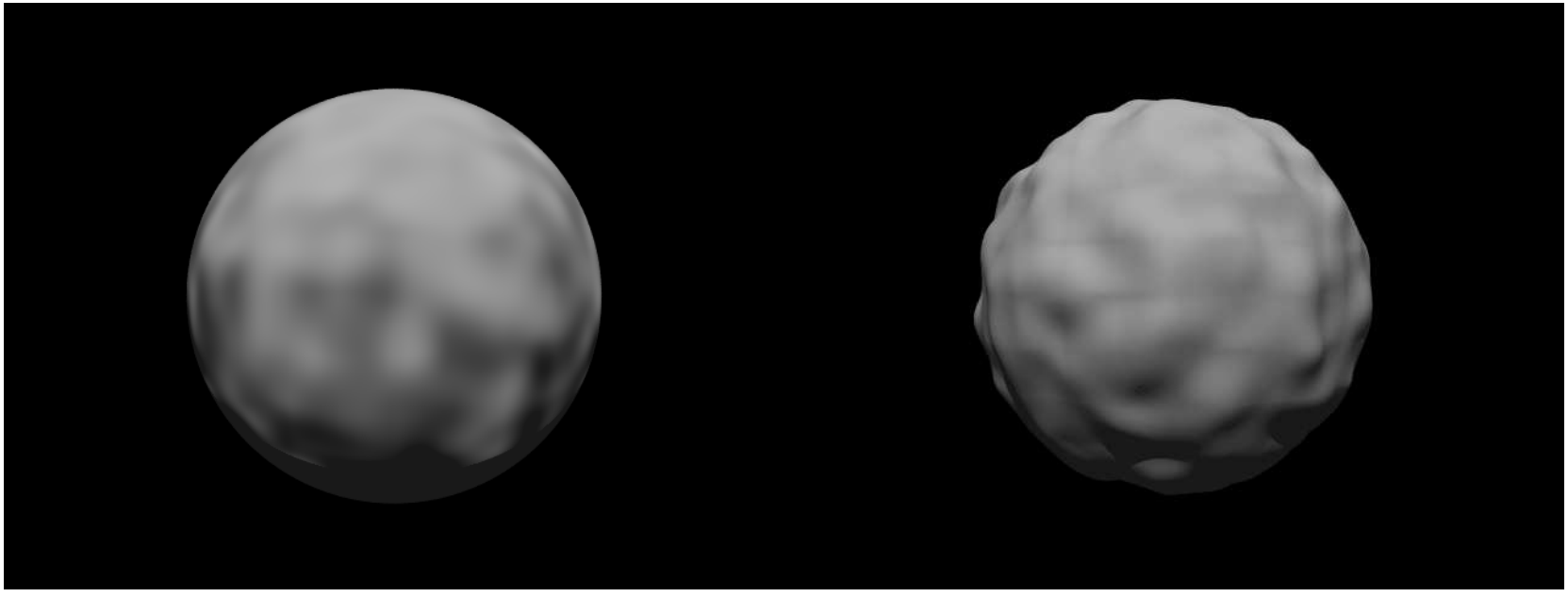
Paul Tosca



Mathias Broxvall

Displacement mapping

- Texture encodes *displacement distance* (measured along normal).
- Input triangles are tessellated (tessellation shaders) and vertices are displaced.
- This changes the geometry of the object.



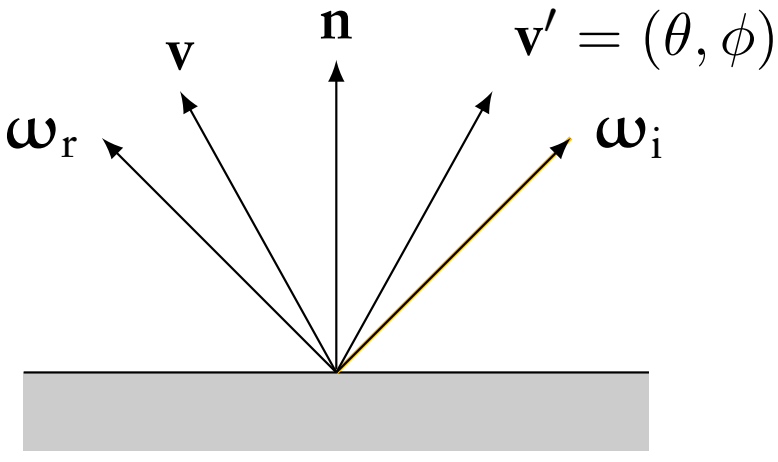
How to render chrome?

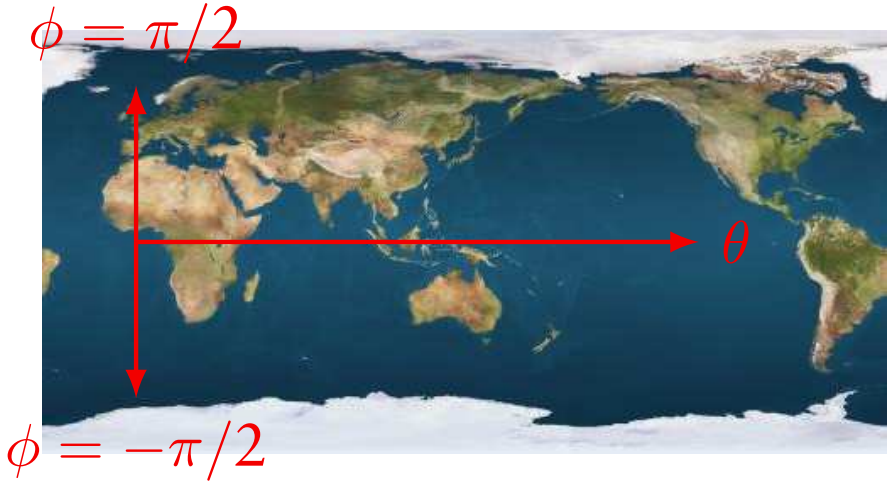
- Why can't we render this object with the reflectance models from yesterday?

- They only take into account *light sources*, not surrounding objects.
- (And point light sources also wouldn't show up in reflection — probability of measuring the impulse reflection is zero.)
- Solution: change light model to use texture instead of (point) light sources.

Reflections with (spherical) environment mapping

- Treat *reflected view vector* $\mathbf{v}'$ as point on unit sphere.
- Have a texture that can be projected to sphere.
- Two coordinates: longitude θ and latitude ϕ (in world coordinates).
- Get reflected colour from $u = \theta/(2\pi)$, $v = \phi/\pi + 1/2$.



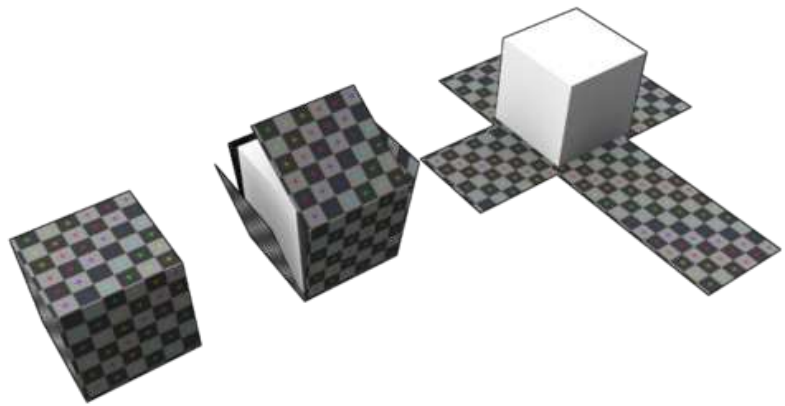


Which container?

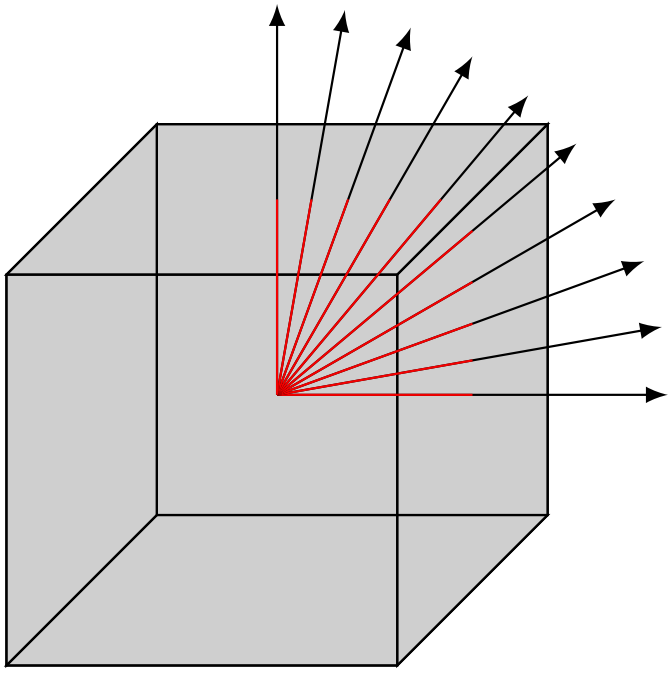
- Last slide used spherical projection of the environment map.
- Often: use box instead.
  - Snap 6 photos — or render 6 images — and send them to OpenGL as a `samplerCube`.

Cube map parametrisation

- 3D texture coordinates (u, v, w).
- Let $t = \max(|u|, |v|, |w|)$.
- Compute $(u/t, v/t, w/t)$: one coordinate will be $\pm\frac{1}{2}$, and the other to $\in [-\frac{1}{2}, \frac{1}{2}]$.
  - That's on a cube!
- Avoids distortions at poles.



Cube map parametrisation illustrated



→ reflected view vector (x, y, z)

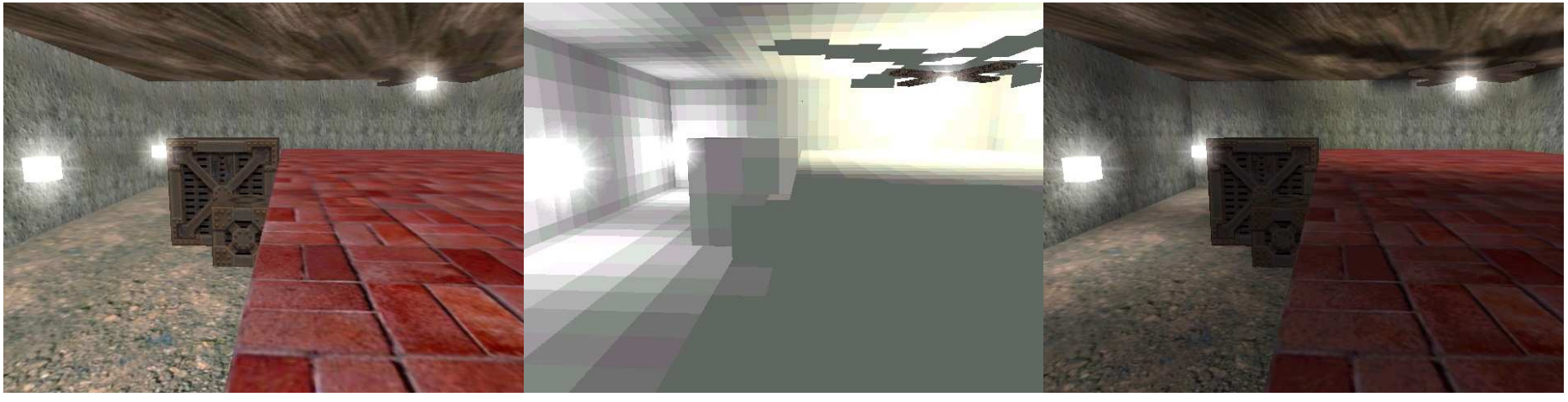
— projection to cube map $\frac{(x,y,z)}{2\max(|x|,|y|,|z|)}$

Environment mapping pros and cons

+ Fast (compared to ray tracing).
– Must be recalculated if scene changes.
– No parallax: reflection doesn't change when translating the object.

Light maps

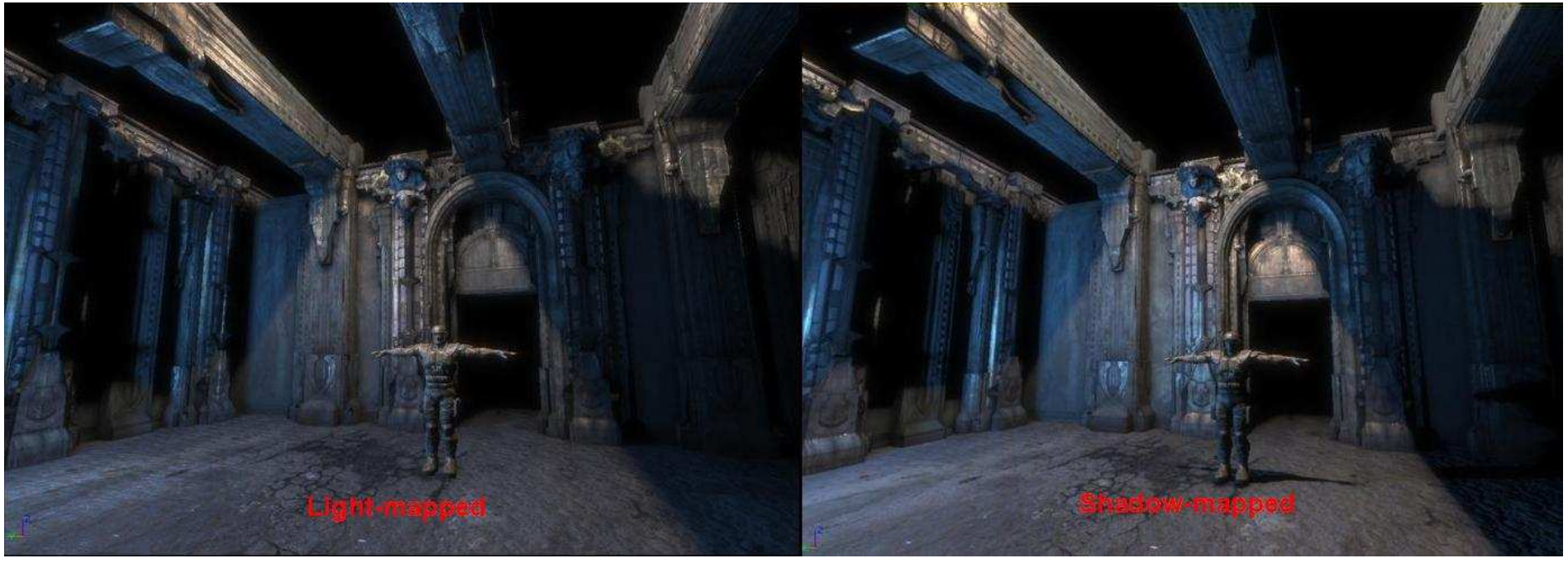
- *Precompute* brightness (e.g. using indirect illumination techniques — see lecture 8)
  - Can handle shadows, bouncing light.
  - Any complex algorithm can be used...
- Represent result as texture.
- Multiply lightmap with rest.



Quake, id software

Light mapping in Mirror's Edge

Shadow mapping



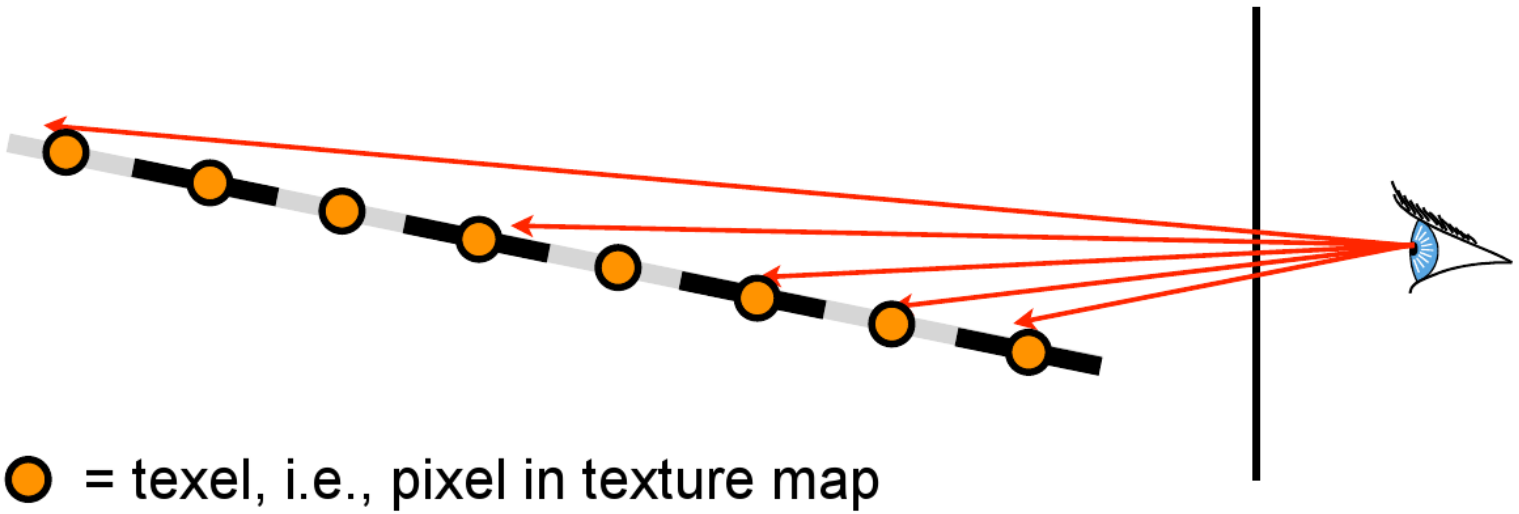
Epic Games

- Light mapping is precomputed: moving objects don't cast shadows.
- Shadow mapping to the rescue.
- More about this next week.

Sampling function

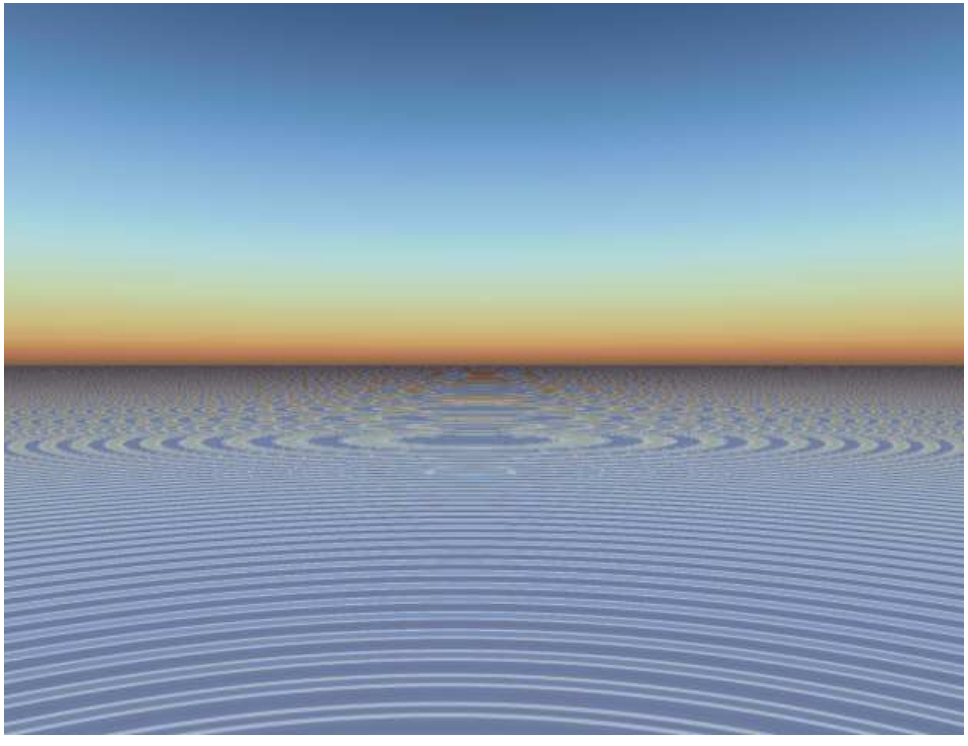
- Usually, we know $[u, v]$ texel coordinates for vertices (especially, if we specified them by hand).
- For all other pixels we have to find texel coordinates according to vertices and get the $[r, g, b, a]$ values of corresponding texel — that's a sampling function task.
- Problem: most likely texture will not fit our surface 1 to 1, and we will get non-integer $[u, v]$ coordinates $\Longrightarrow$ *texture aliasing*.

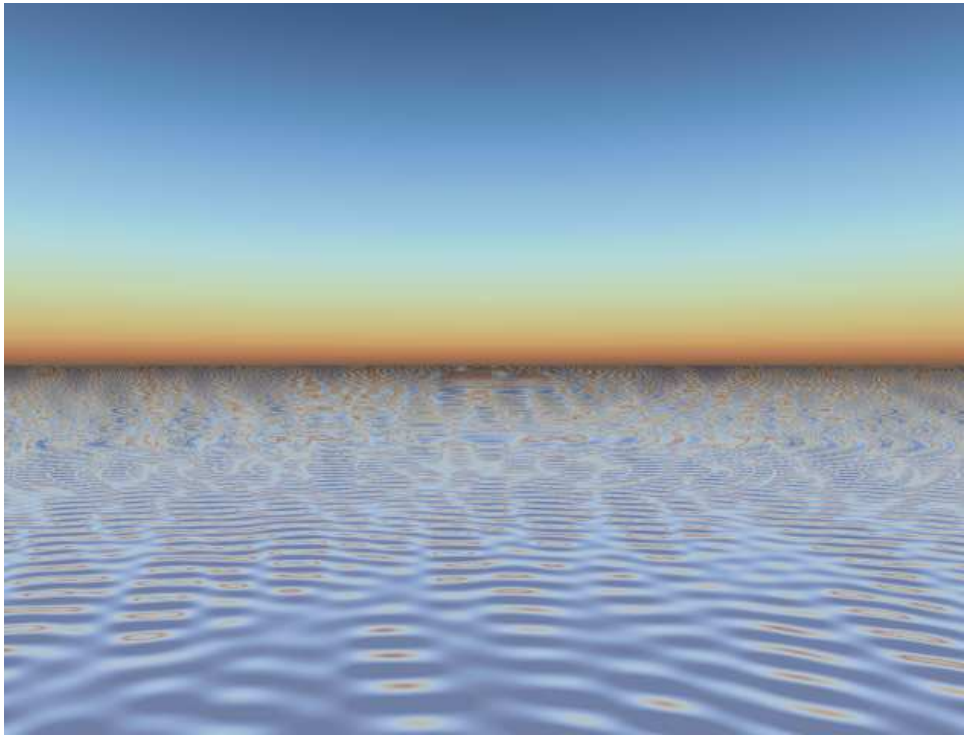
Aliasing



Jaakko Lehtinen

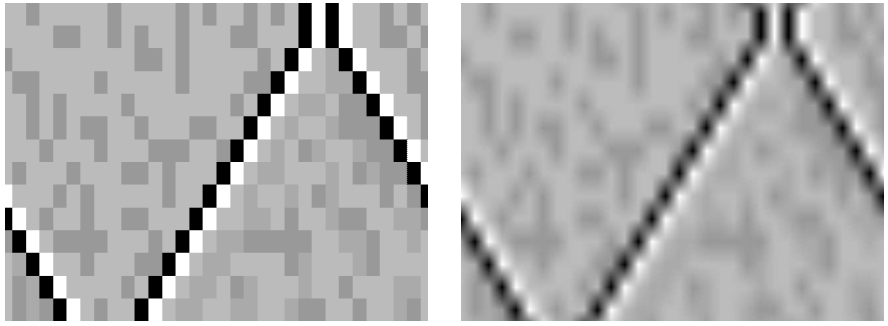
Aliasing, example





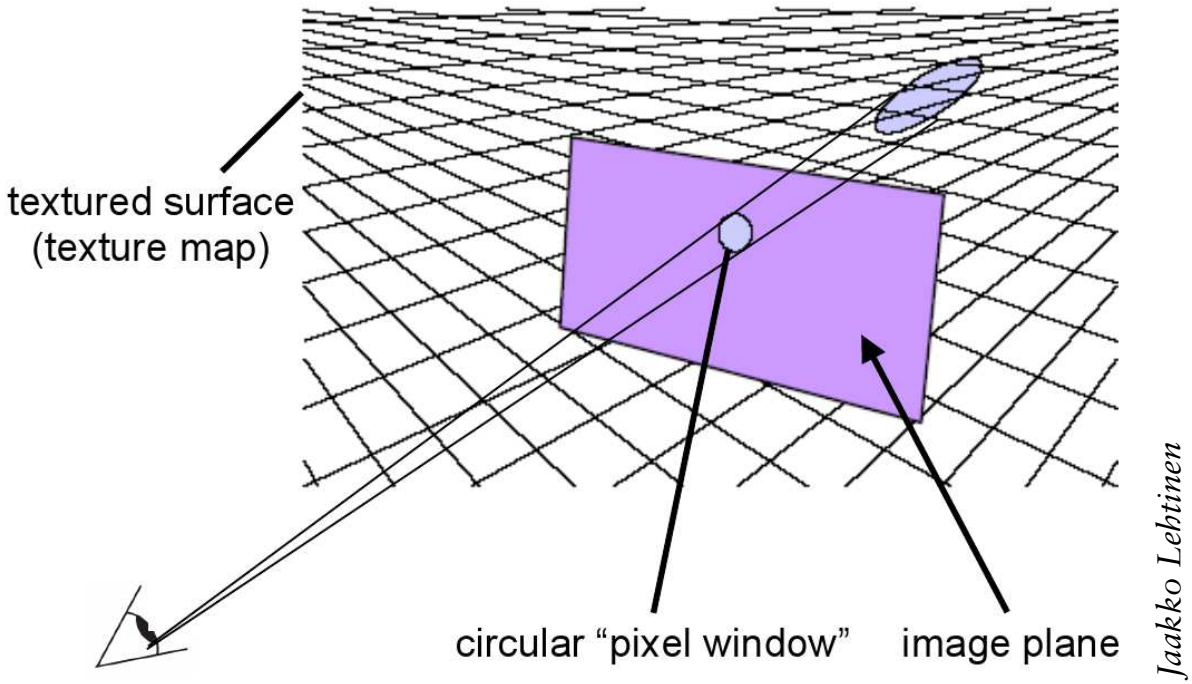
Interpolation

- Nearest-neighbour
  - Pixellated results for near objects.
  - Heavy aliasing for far objects.
- Bilinear
  - Smooths/blurs.
  - But still cannot reconstruct the true texture if we have too sparse sampling.



Sampling texture maps

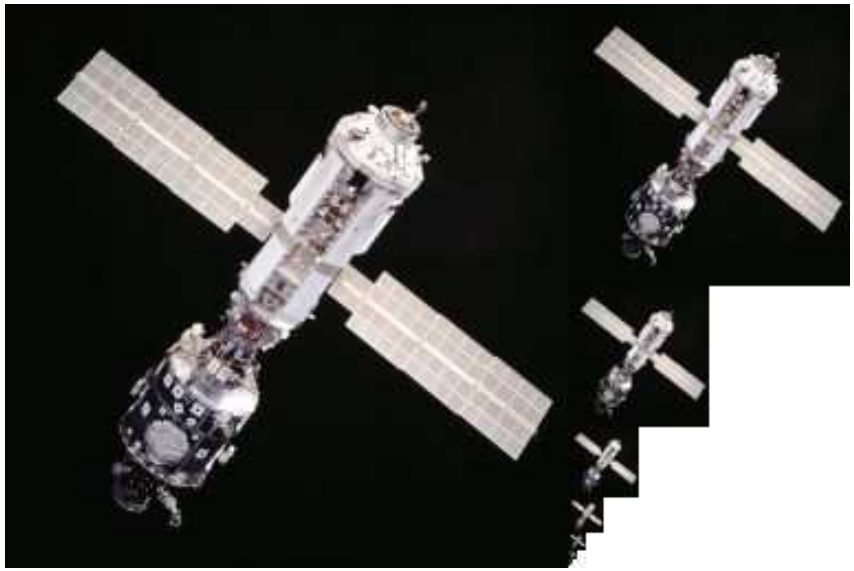
- Interpolation is not enough.
- We need to collect *several texture samples per pixel* (anti-aliasing).
- How many samples are needed?
  - 2× the frequency of the texture.



Jaakko Lehtinen

MIP mapping

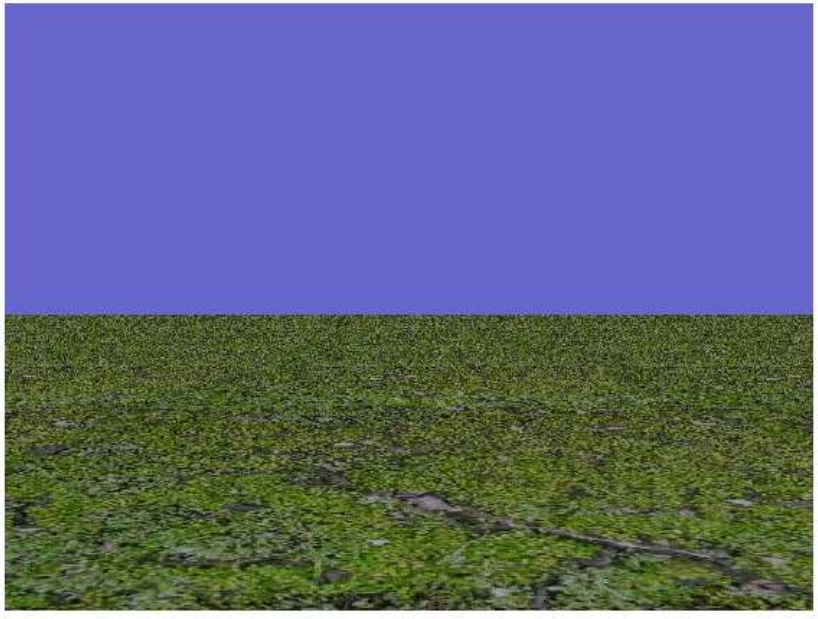
- Keep interpolation calculations outside the pipeline.
- Precompute sampling for far surfaces:
- Take the texture at highest resolution and downscale it by factor 2 until you get 1×1-pixel texture.
- Textures together form a MIP map ("multim in parvo").
- Before sampling, choose subtexture and use (nearest-neighbor) interpolation on that.
- Slightly more memory required, but less calculations.



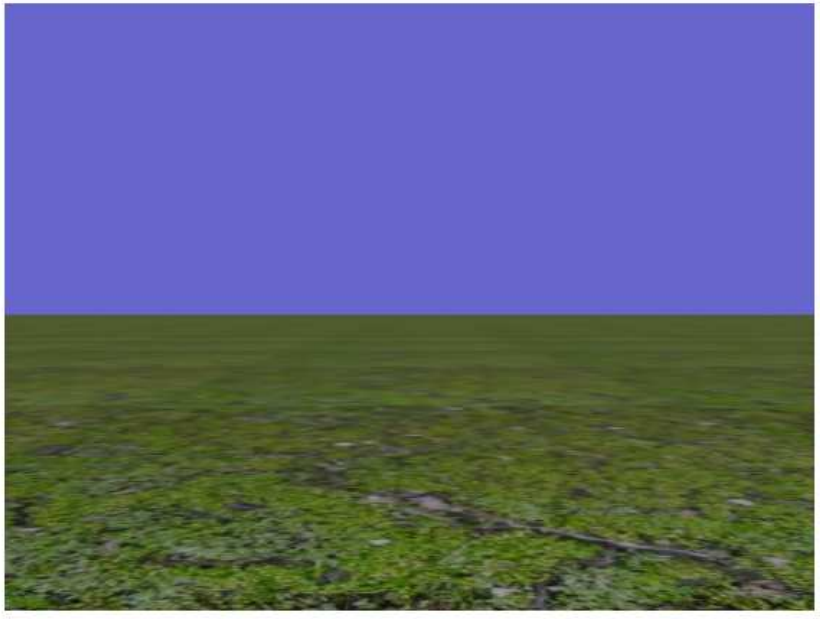
Mike Hicks, via Wikimedia Commons

Limitations of isotropic MIP mapping

- Isotropic: blurs equally in all directions. (OK for moving object, less OK for infinite plane.)



Without mipmapping



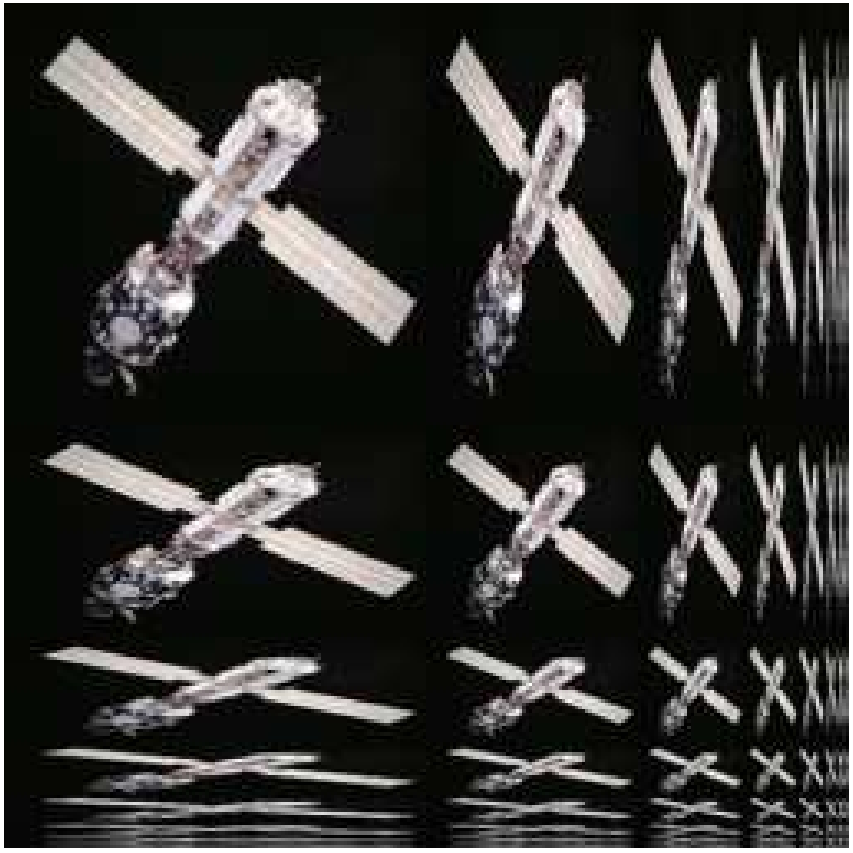
With (isotropic) mipmapping

Mathias Broxvall

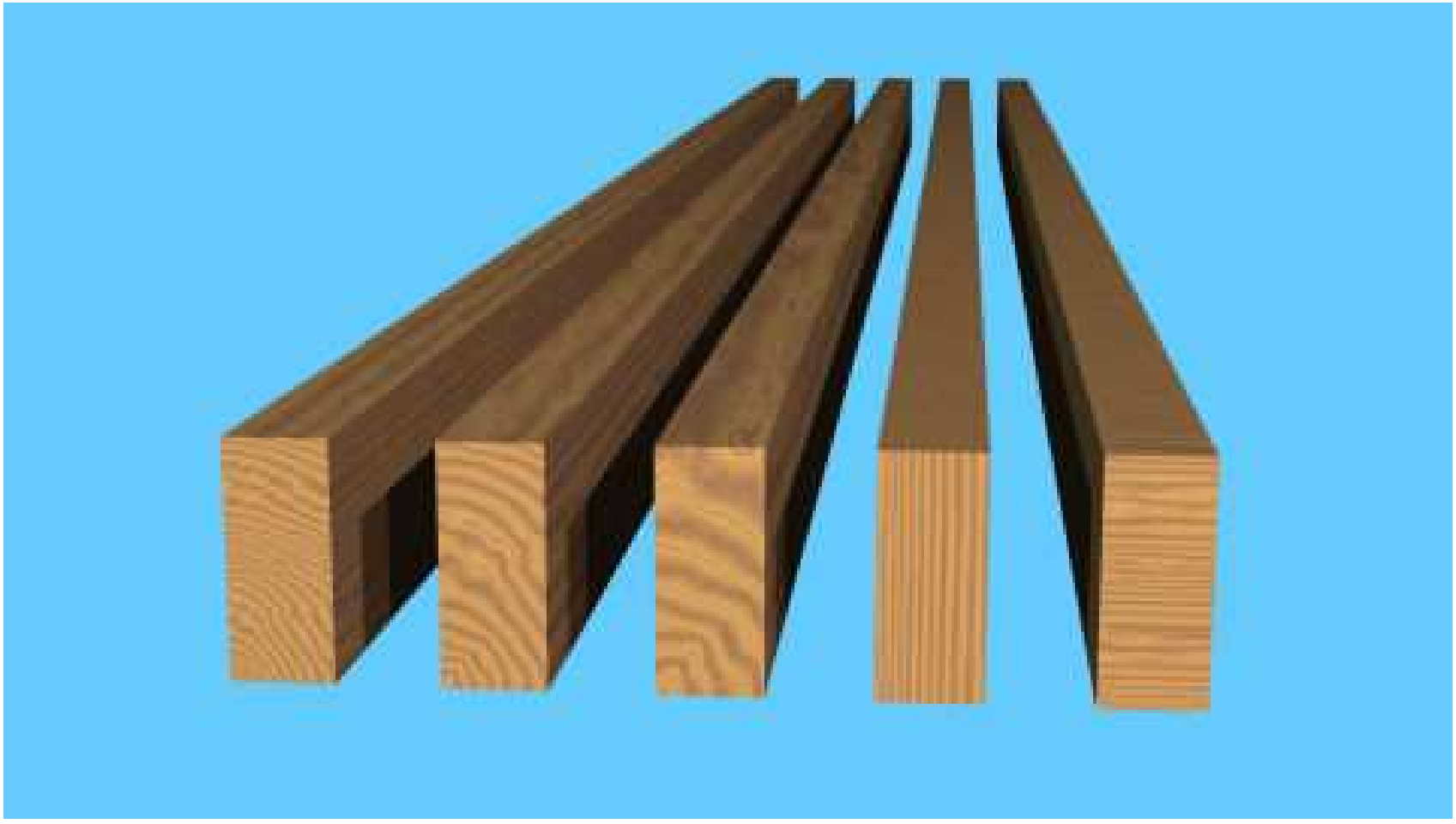
RIP mapping

Anisotropic MIP mapping

- Downscale textures non-uniformly = anisotropic.
- More memory required.

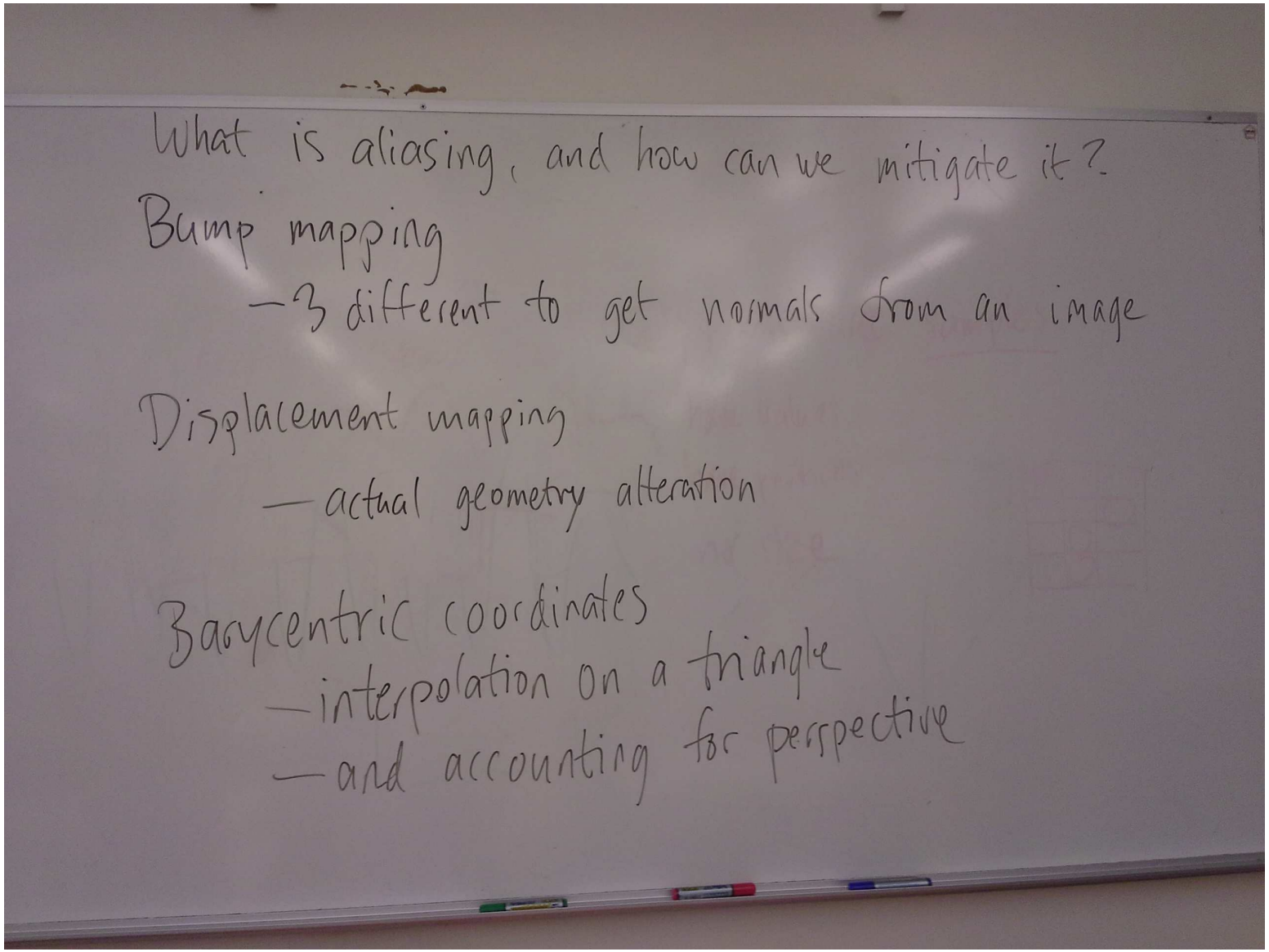


Procedural texture examples



Procedural texture examples

Summary


What is aliasing, and how can we mitigate it?
Bump mapping
— 3 different to get normals from an image
Displacement mapping
— actual geometry alteration
Barycentric coordinates
— interpolation on a triangle
— and accounting for perspective

Next lecture: shadows

- Mon Nov 14, 13.15–15.00
- T-141
- Hughes et al.: 15.6.5 (not much!)
- This is better: Eisemann et al., *Shadow Algorithms for Real-time Rendering*, Chapters 1, 2, 7.