Computer Graphics

-Texture Mapping Basics-

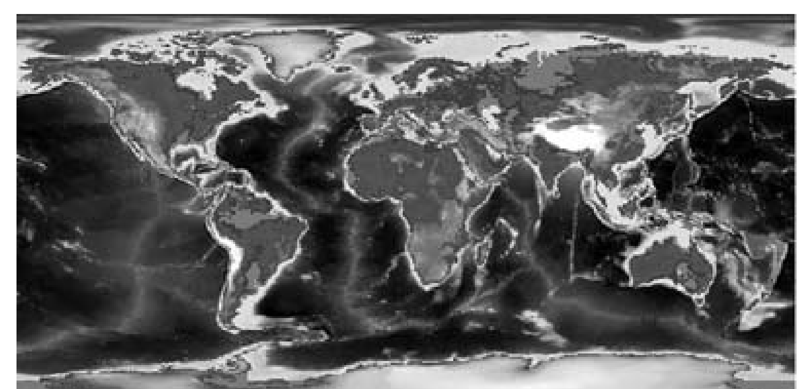
Oliver Bimber

Course Schedule

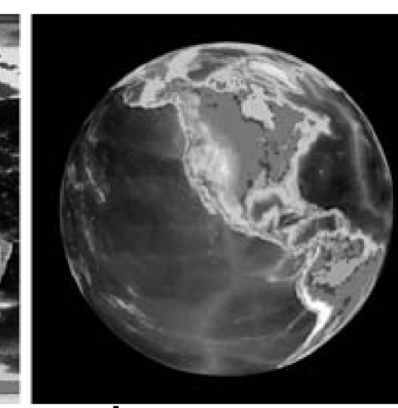
Type	Date	Time	Room	Topic	Comment
C1	01.03.2016	13:45-15:15	HS 18	Introduction and Course Overview	Conference
C2	15.03.2016	13:45-15:15	HS 18	Transformations and Projections	Easter Break
C3	05.04.2016	13:45-15:15	HS 18	Raster Algorithms and Depth Handling	
C4	12.04.2016	13:45-15:15	HS 18	Local Shading and Illumination	
C5	19.04.2016	13:45-15:15	HS 18	Texture Mapping Basics	
C6	26.4.2016	13:45-15:15	HS 18	Advanced Texture Mapping & Graphics Pipelines	
C7	03.05.2016	13:45-15:15	HS 18	Intermediate Exam	
C8	09.05.2016	17:15-18:45	HS 18	Global Illumination I: Raytracing	
C9	10.05.2016	13:45-15:15	HS 18	Global Illumination II: Radiosity	Conference / Holiday
C10	31.05.2016	13:45-15:15	HS 18	Volume Rendering	
C11	07.06.2016	13:45-15:15	HS 18	Scientific Data Visualization	
C12	14.06.2016	13:45-15:15	HS 18	Curves and Surfaces	
C13	21.06.2016	13:45-15:15	HS 18	Basics of Animation	
C14	28.06.2016	13:45-15:15	HS 18	Final Exam	
C15	04.10.2016	13:45-15:15	TBA	Retry Exam	

Non-Uniform Surface Reflectance

- We have looked at how 3D models are shaded (i.e., how reflection of light on the surface with uniform reflectance can be simulated)
- What about surfaces with fine surface details (i.e., non-uniform reflectance)?
- One option would be to have an extremely high resolution geometric model with each vertex/triangle having a different reflectance
- This would be a rendering-overkill
- Better: low resolution geometry + details „wall-painted“ on the geometry



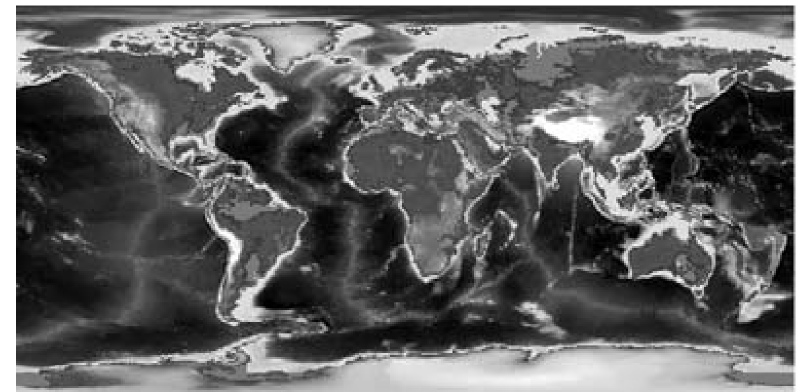
details are stored in image (texture)...



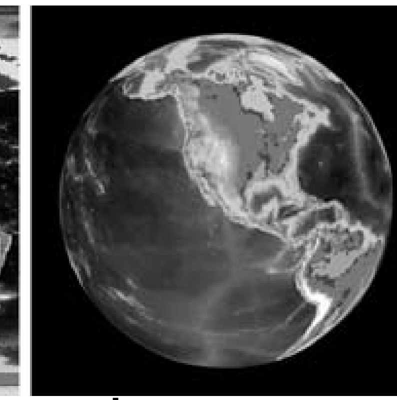
...and are mapped onto simple geometry

2D Texture Mapping

- There are two basic forms of texture-mapping techniques: 2D texture mapping and 3D texture mapping
- Let's look at 2D texture-mapping first (this is the most common case in CG)
- Texture details (called „texels“) are stored in a 2D image and are wall-painted onto the 3D surface geometry during rendering - but how?
- Let's assume a normalized (0...1) 2D texture space in u,v coordinates
- The reflectance value stored in the texture is then a function of R(u,v)
- Assigning a reflectance to a 2D parameterizable surface (e.g., a sphere) is straight forward



details are stored in image (texture)...



...and are mapped onto simple geometry

$$x = x_c + Rcos(\phi)sin(\theta)$$
$$y = y_c + Rsin(\phi)sin(\theta)$$
$$z = z_c + Rcos(\theta)$$

parametric sphere equation: x_c,y_c,z_c=center , R=radius

$$\theta = arccos(\frac{z - z_c}{R})$$
$$\phi = arctan(\frac{y - y_c}{x - x_c})$$

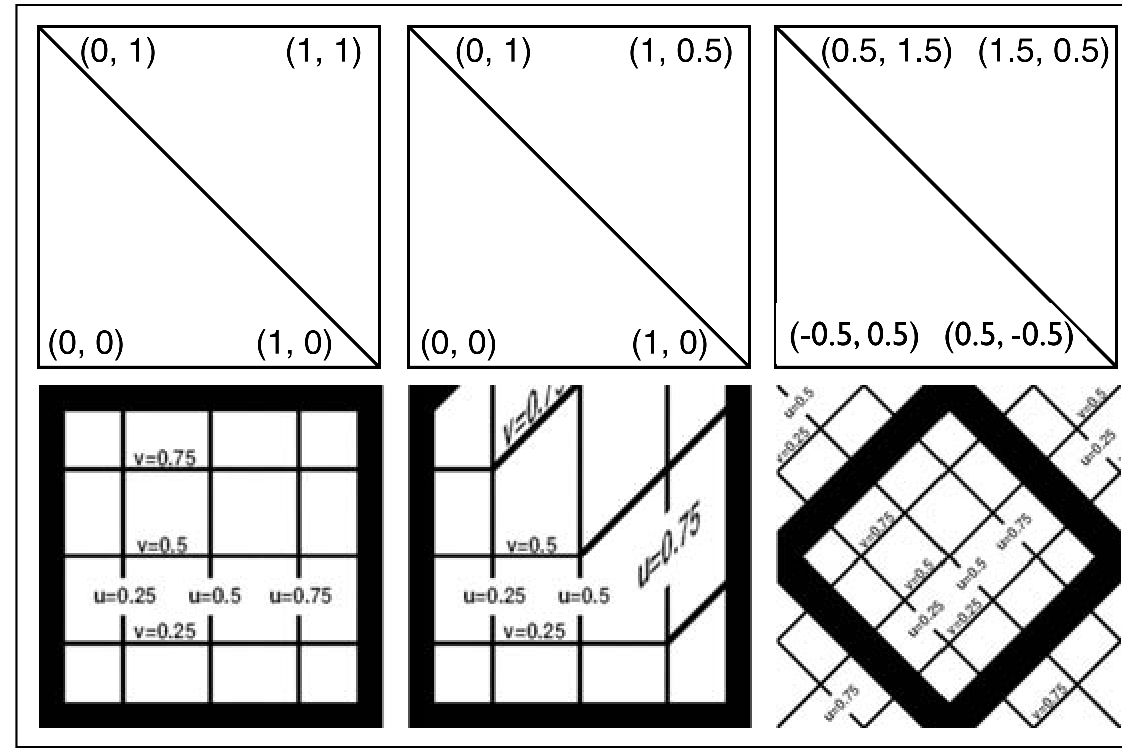
spherical coordinates (longitude and latitude)

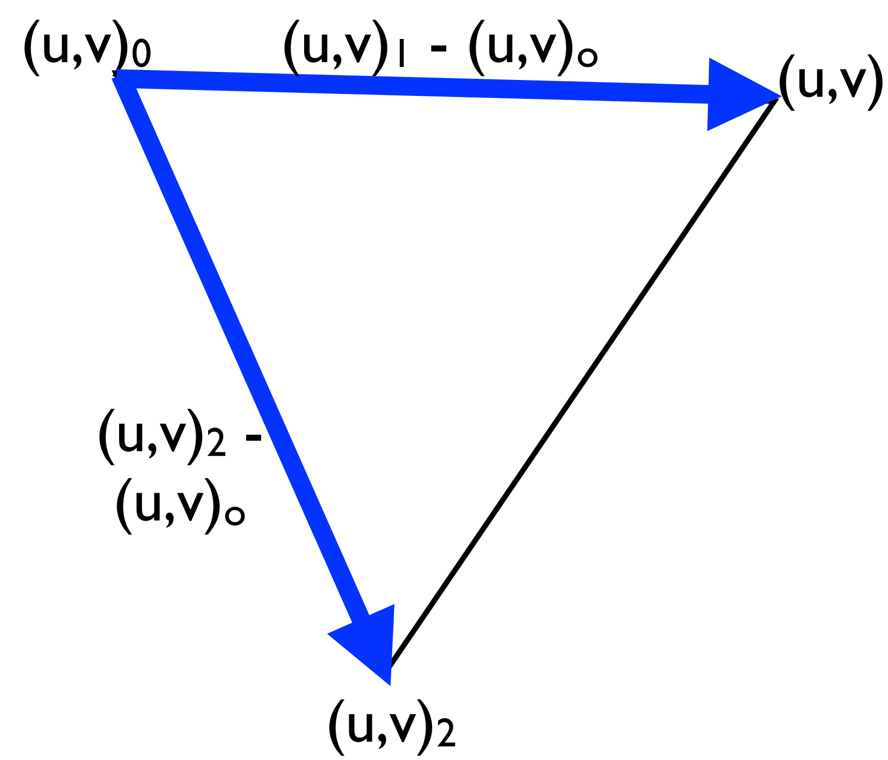
$$u = \frac{\phi}{2\pi}$$
$$v = \frac{\pi - \theta}{\pi}$$

assignment to texture coordinates

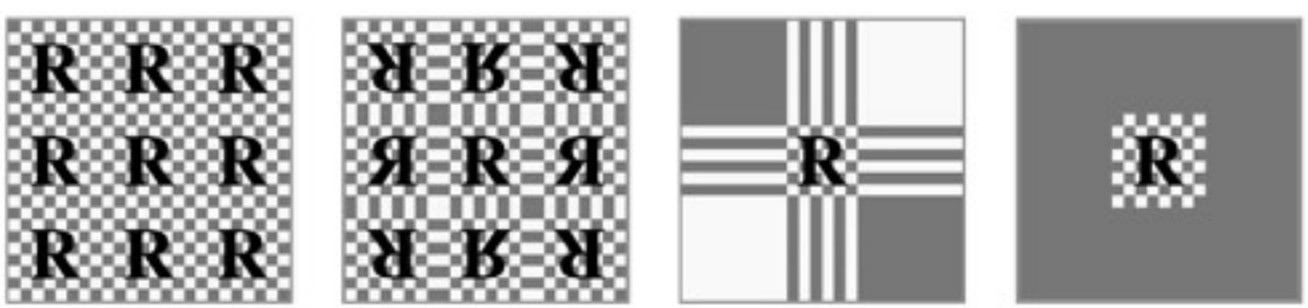
Tessellated Models

- How about texture mapping tessellated surfaces (i.e., triangle meshes)?
- Determine texture coordinate for each vertex (can be computed with a function, or is part of the modeling process)
- Bilinear interpolate texture coordinate across triangle via barycentric coordinates (just like for other parameters, such as normals, color, shading, etc.)
- Thus, texture-mapping is also part of the rasterization stage
- If texture coordinates are beyond 0 and 1, then textures can either be repeated, mirrored, clamped or bordered





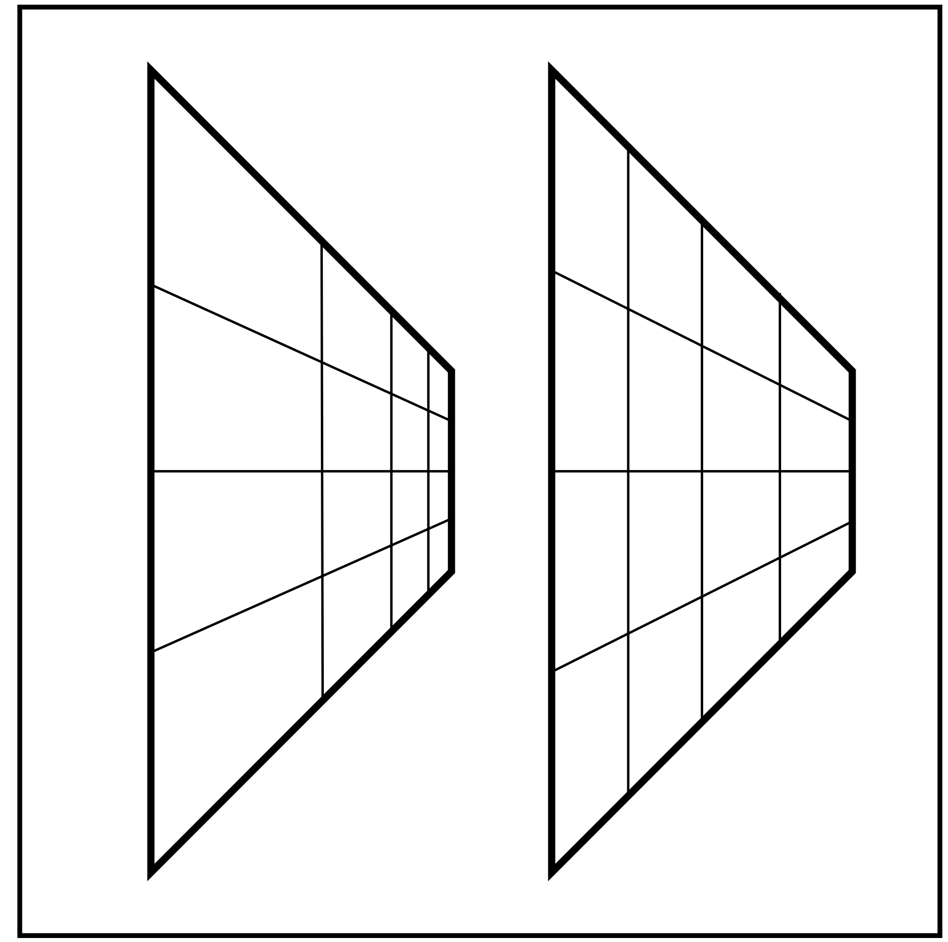
$$u(\beta, \gamma) = u_0 + \beta(u_1 - u_0) + \gamma(u_2 - u_0)$$
$$v(\beta, \gamma) = v_0 + \beta(v_1 - v_0) + \gamma(v_2 - v_0)$$



repeat mirror clamp border

Perspective Correct Textures

- The problem is, that interpolating texture coordinates in screen space results in incorrect perspectives
- If we interpolate in screen space, parallel lines in the texture remain evenly spaces, for example
- This is not correct, since for perspective images, features get smaller the further they are away from the viewer
- Thus, we have to carry out some sort of correction to the texture coordinates to ensure a perspectively correct appearance
- Let‘s go back to perspective projection for a moment



correct appearance of perspective texture

incorrect appearance of perspective texture (interpolated in screen space)

Perspective Correct Textures

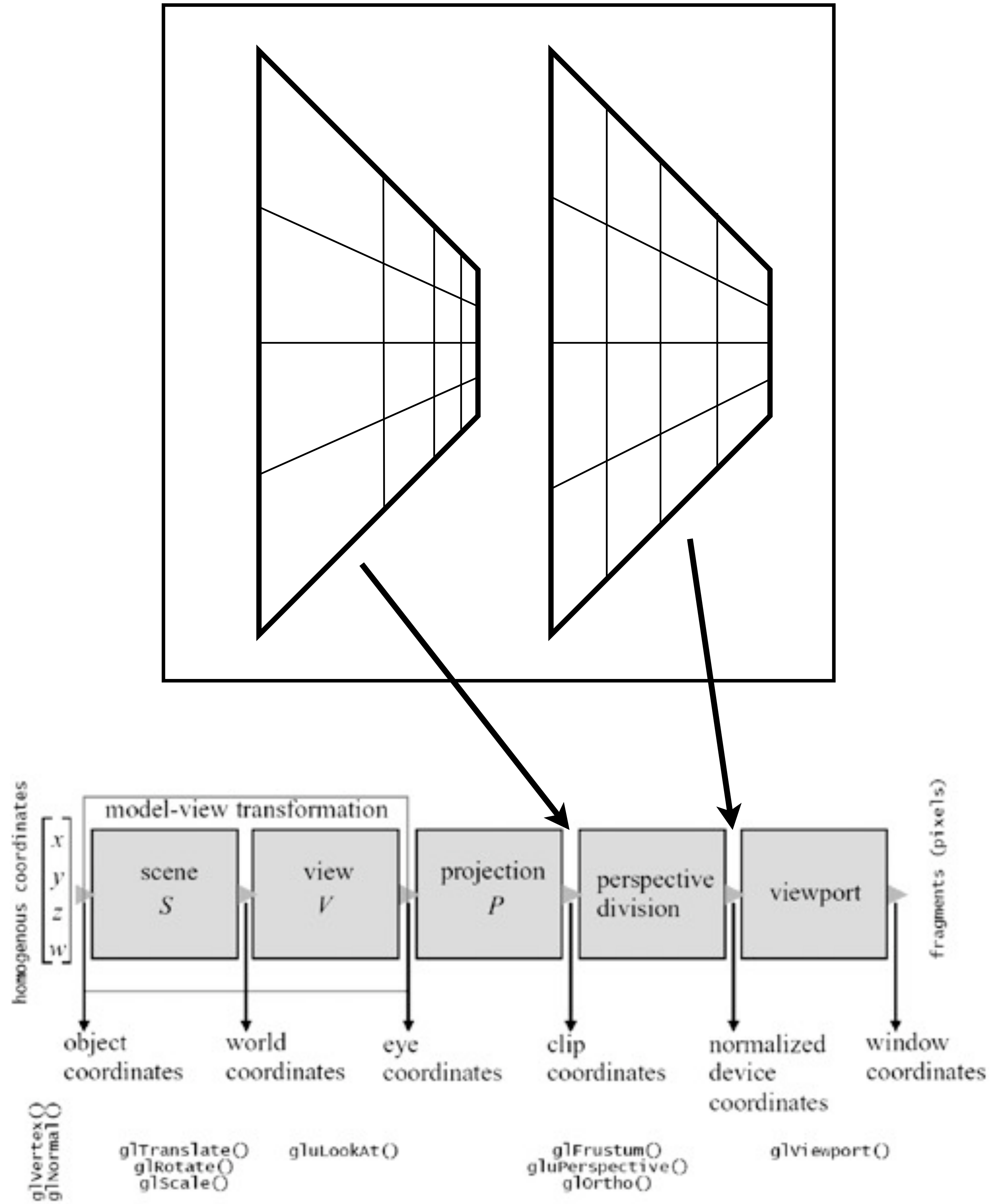
- Interpolating texture coordinates in screen space means that this is done AFTER the perspective division

- Interpolating AFTER perspective division does not lead to perspectively distorted interpolation values (which are interpolate linearly in screen space)!

- However, for correct perspective, we want to interpolate BEFORE the perspective division

- Thus, we want to compute texture coordinates with barycentric coordinates in world space - not in screen space

- However, all other parameters are interpolated in screen spaces - as we learned earlier



Perspective Projection Recap.

- The value (f) is the distance to the far clipping plane of our viewing frustum, and (n) the distance to its near clipping plane
- Here, the homogeneous coordinate (h) plays an important role
- If we allow the homogeneous coordinate to take up any value (not just 0 or 1), then it can be used to encode how much the other three coordinates must be scaled after a perspective transform
- Dividing these three coordinates by the homogeneous coordinate is called homogenization or perspective division
- As for the orthographic case, the resulting value in the z component is not used yet - it will be used for hidden surface removal since it preserves depth order

Perspective transform matrix M_p

$$\boxed{\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix}} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} nx \\ ny \\ (n+f)z - fn \\ \boxed{z} \end{bmatrix}$$

homogeneous coordinate

$$\begin{bmatrix} nx \\ ny \\ (n+f)z - fn \\ z \end{bmatrix} \boxed{/z} = \begin{bmatrix} \frac{n}{z}x \\ \frac{n}{z}y \\ n+f-\frac{fn}{z} \\ 1 \end{bmatrix}$$

homogenization
or perspective division

$$M_p^{-1} = \begin{bmatrix} \frac{1}{n} & 0 & 0 & 0 \\ 0 & \frac{1}{n} & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -\frac{1}{fn} & \frac{n+f}{fn} \end{bmatrix}$$

Perspective Correct Textures

- We compute the barycentric coordinates of a triangle in world space directly from barycentric coordinates in screen space
- One key observation is, that the homogeneous coordinate (1/h , or 1/z in the previous slide) that is used for perspective division is linear in screen space (i.e. it can be interpolated linearly without causing any error)
- This allows us to compute the correct barycentric coordinates that considers perspective
- The world space barycentric coordinates are used for computing the texture coordinates

$$p = p_0 + \beta(p_1 - p_0) + \gamma(p_2 - p_0)$$

$$p = (1 - \beta - \gamma)p_0 + \beta p_1 + \gamma p_2$$

interpolation in a triangle (recap.)

β_w, γ_w is in world space

β_s, γ_s is in screen space

$\frac{\beta_w}{h}, \frac{\gamma_w}{h}$ can be interpolated in world space without error!

$$\frac{\beta_w}{h} = \frac{\beta_{w0}}{h_0} + \beta_s\left(\frac{\beta_{w1}}{h_1} - \frac{\beta_{w0}}{h_0}\right) + \gamma_s\left(\frac{\beta_{w2}}{h_2} - \frac{\beta_{w0}}{h_0}\right)$$

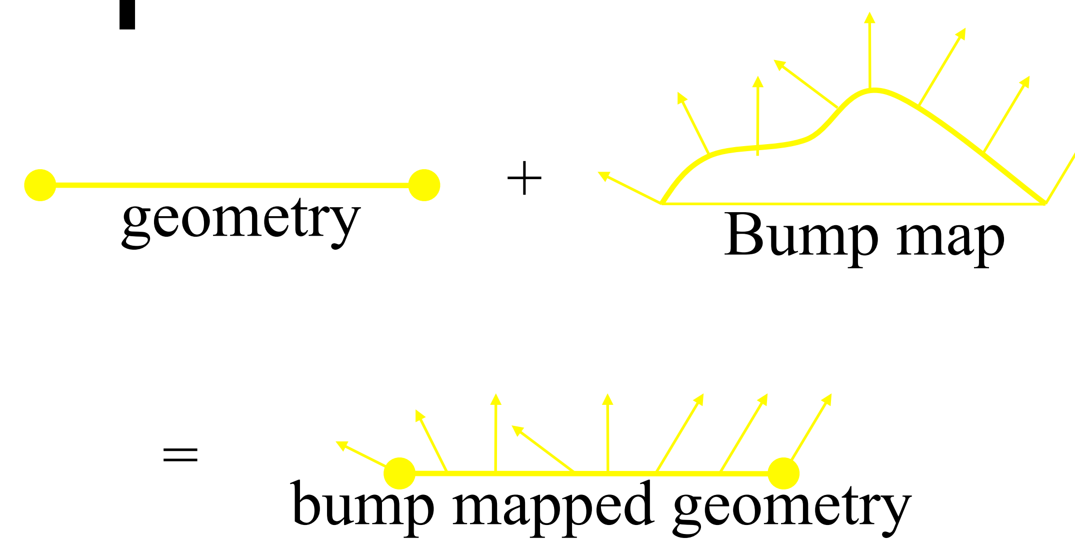
knowing the barycentric coordinates in world space for the three vertices simplifies this equation

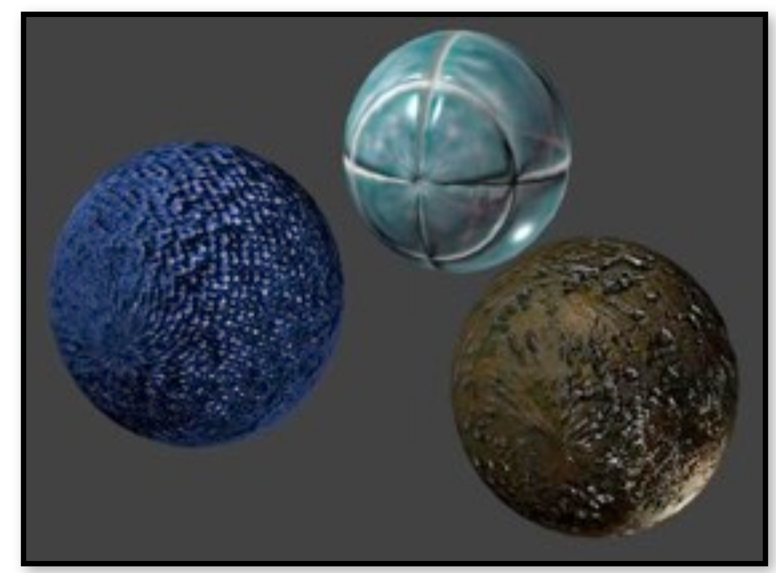
$$\beta_{w0} = \beta_{w2} = 0 \qquad \beta_{w1} = 1$$

$$\frac{\beta_w}{h} = \frac{0}{h_0} + \beta_s\left(\frac{1}{h_1} - \frac{0}{h_0}\right) + \gamma_s\left(\frac{0}{h_2} - \frac{0}{h_0}\right) = \frac{\beta_s}{h_1}$$

Bump Maps

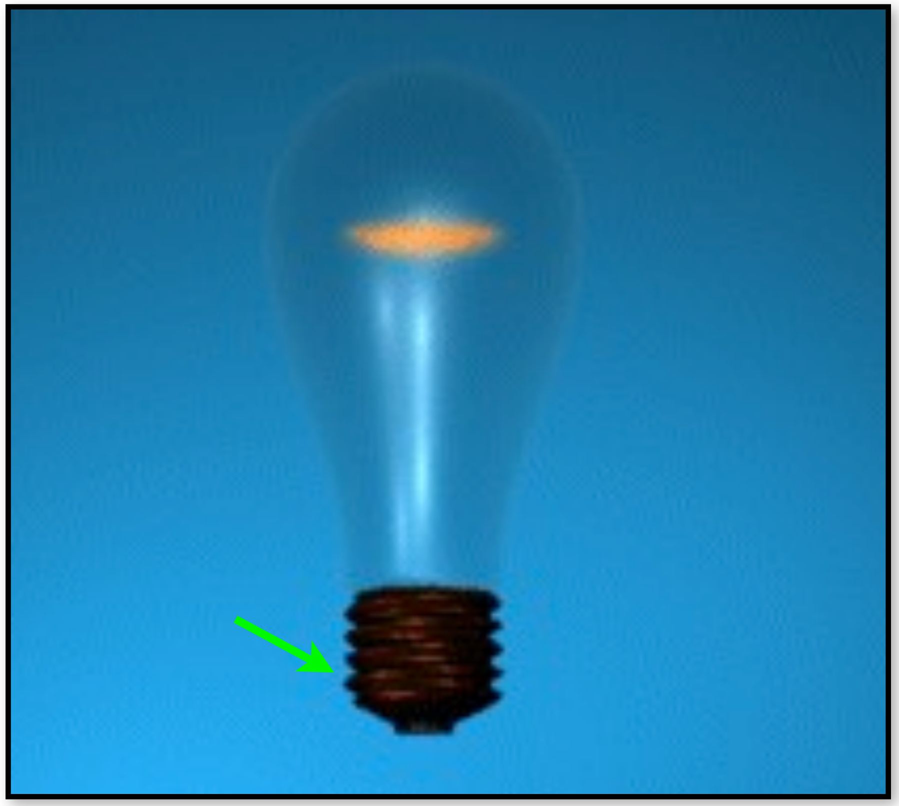
- Note, that if the reflectance of a pixel / surface point has been looked up in the texture, the illumination computations are exactly the same as discussed earlier
- Yet, the structure in 2D textures is flat, which does not cause realistic shading effects of a bumpy surface
- One trick is to store perturbed surface normals in 2D textures, rather than / together with the surface reflectance (this is then called a „bump map“)
- These normals are mapped to the 3D surface geometry in the same way as textures - but the normals are looked up and used for lighting during rasterization

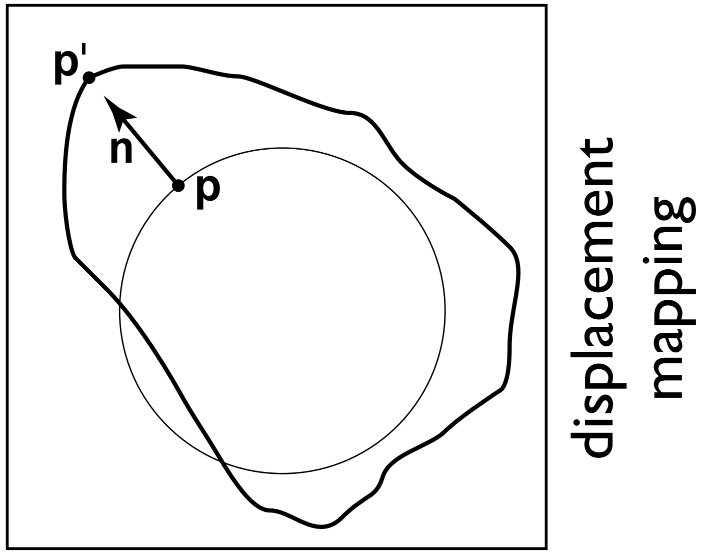




Displacement Maps

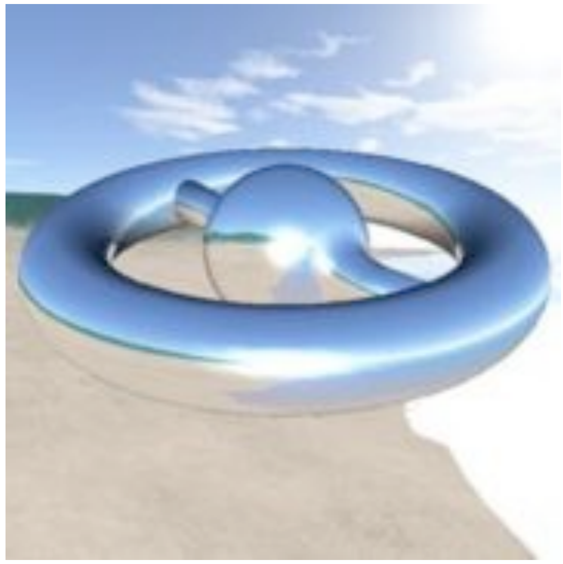
- The problem with bump maps is that they do not create realistic shadow effect, since only surface normals are changed
- They also do not produce bumpy silhouettes
- A way of dealing with this is called „displacement maps“
- Displacement maps are 2D maps that store small displacement offsets along the surface normal
- They can also be mapped onto the surface geometry - but instead of alternating its reflectance of normal vector, they change the surface geometry (displacement by corresponding value along surface normal)
- It can be efficiently implemented in a z-buffer code



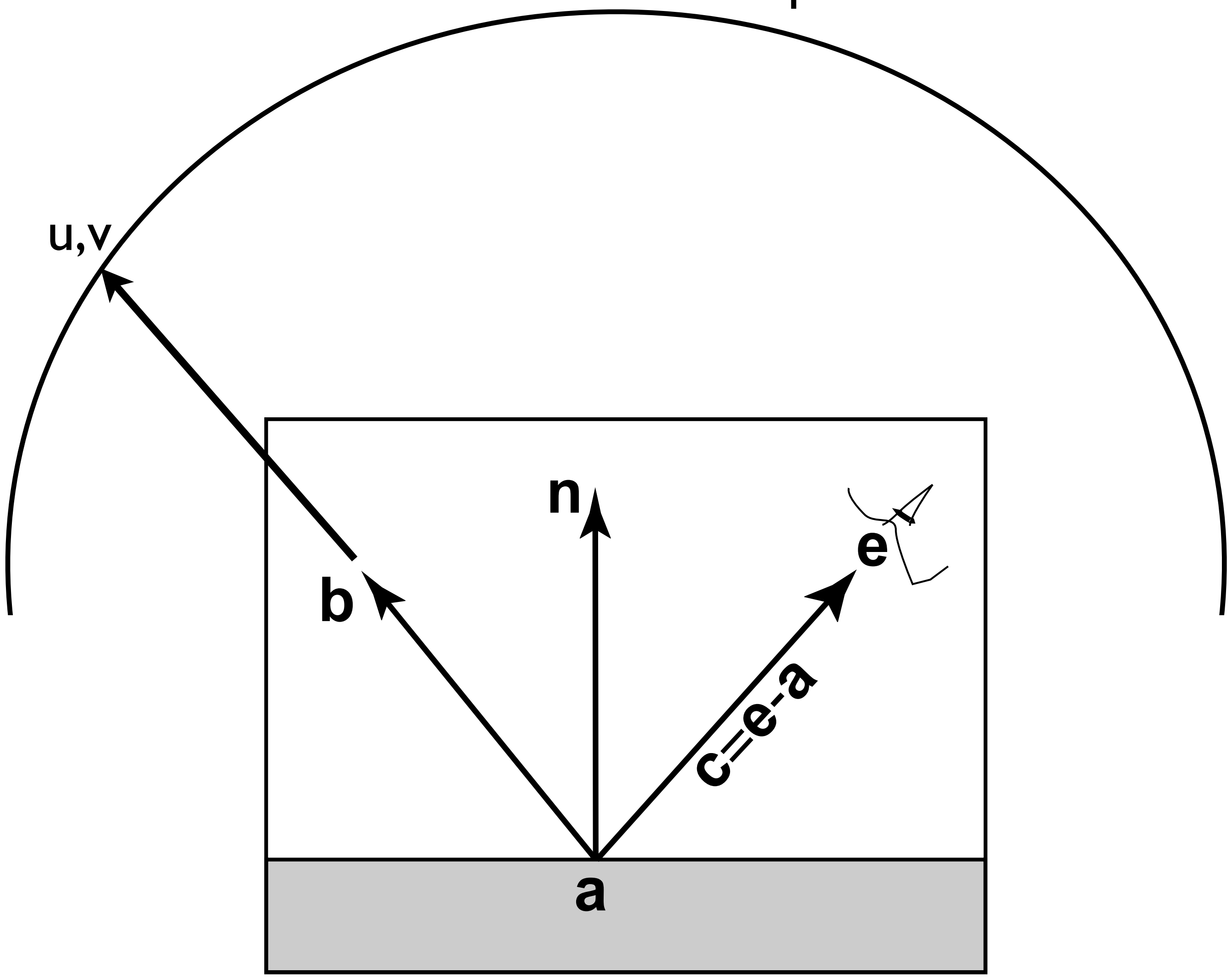


Environment Maps

- Obviously, the texture mapping concept is very powerful and supports much more than simple wall-papering
- Environment maps are yet another example - they allow simulating specular reflections of backgrounds (not only of light sources, as we have seen earlier) on the surface
- One can think of a non-planar, parameterizable texture surface that surrounds the scene
- The texture coordinates that are applied to a surface point (a) depend on the reflection vector (b) on the surface
- There are different ways to store environment maps: eg. spherical index table or cube-based table

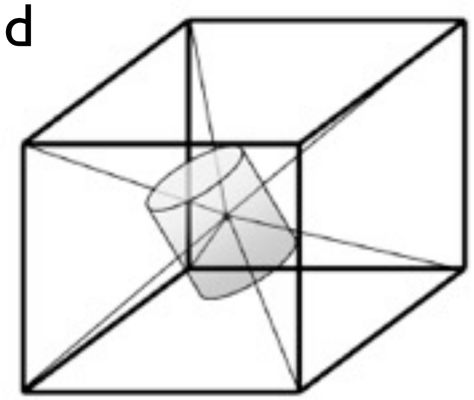


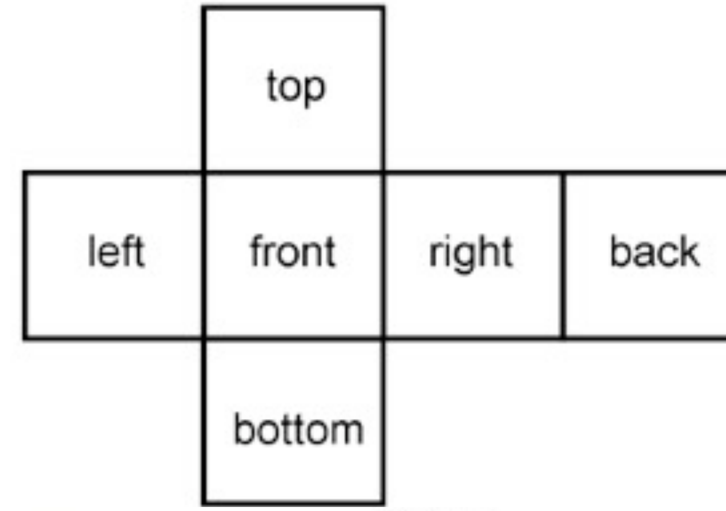
environment map

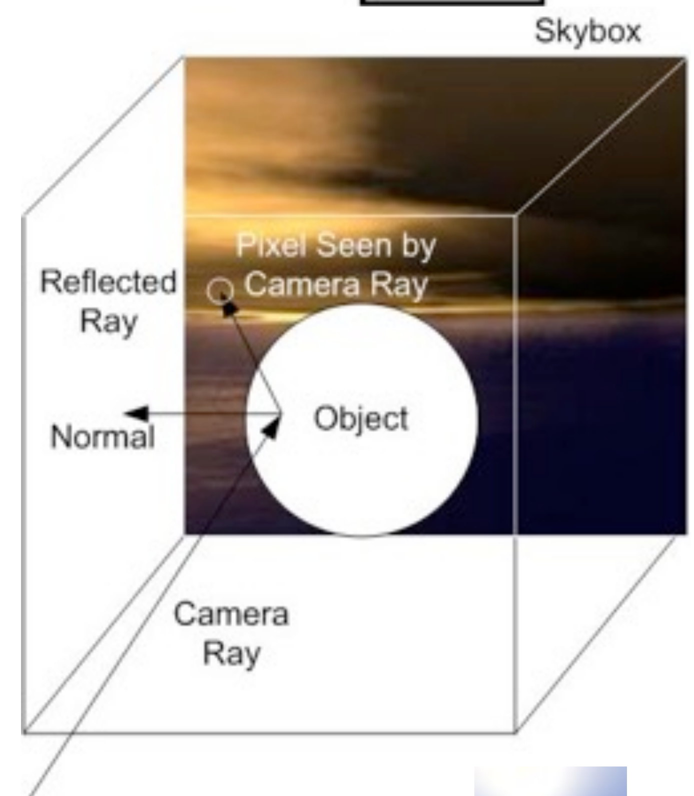


Cube Maps

- A cube map is special form of an environment map that applies a cube-based texture table (mainly because it is easy to generate)
- How to generate a cube map: place your camera in the center of the environment (only containing background objects) and render 6 perspective images for all 6 directions (they are stored in 6 2D textures)
- How to apply a cube map: when rendering the foreground objects from an arbitrary camera position and direction, the reflection vector is computed for each surface point - this is used to index the correct cube face and to compute the correct texture coordinate within it
- So, the texture coordinates are automatically computed for each vertex in this case (they are not user defined)
- Note, that reflections for environment maps are actually only correct for a single 3D point (the camera center or focal point that was used for generating them)

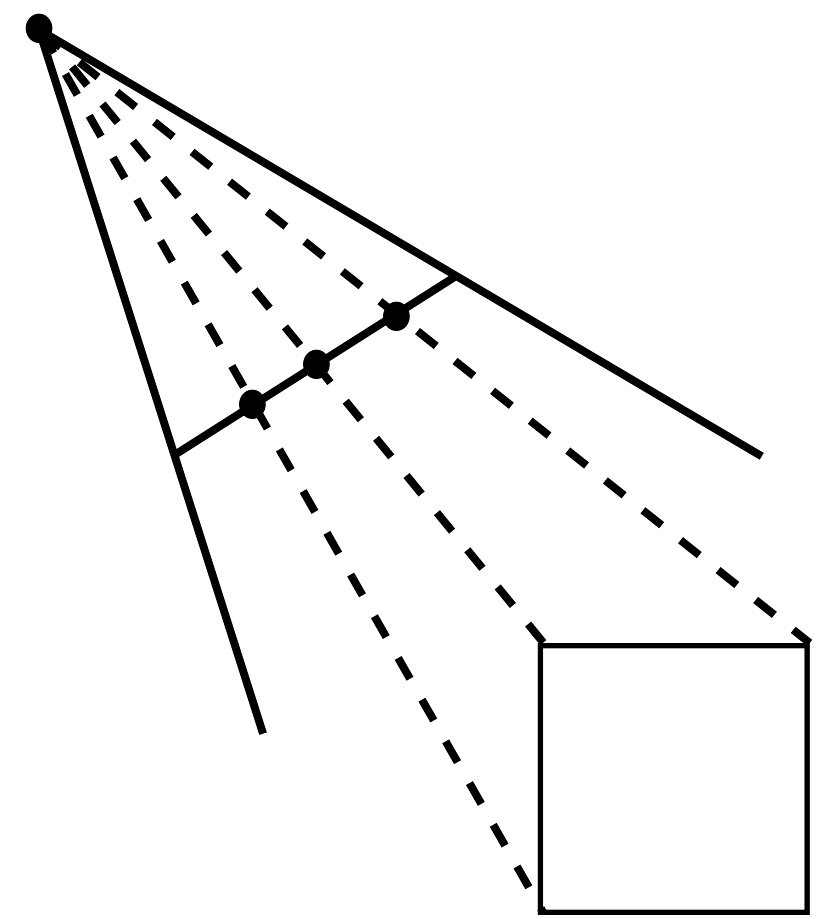


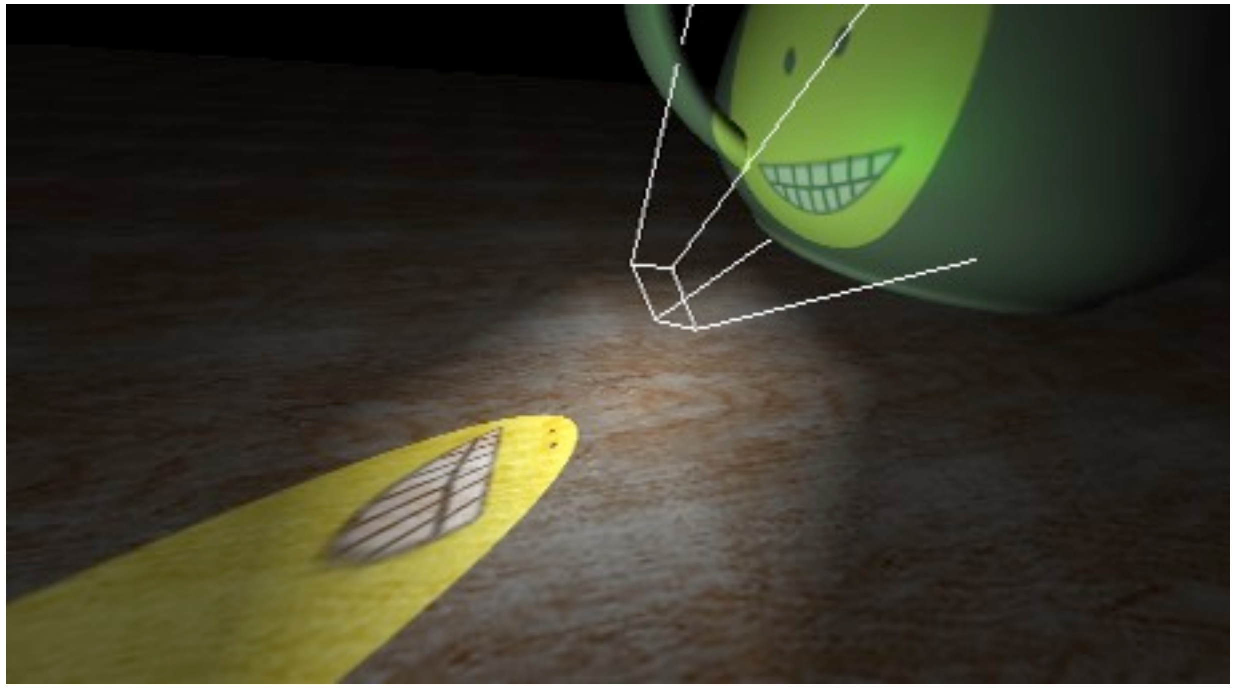




Projective Texture Maps

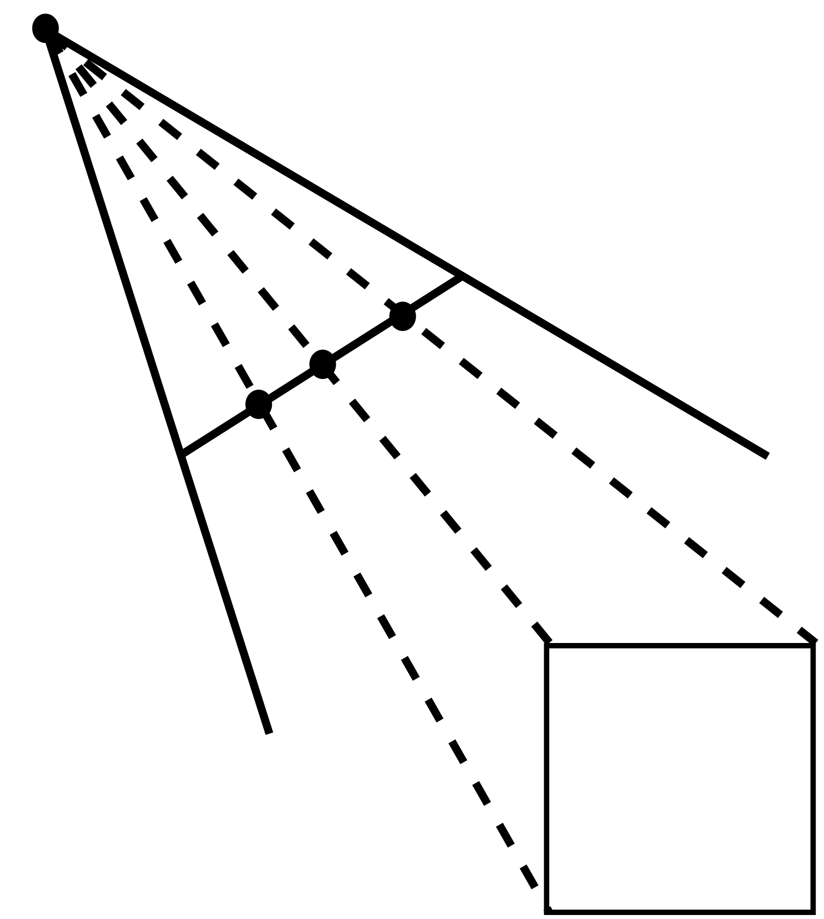
- For projective texture maps, texture coordinates (t) of vertex coordinates (v) are computed in such a way that they map to texels that are projected from a given perspective (like a slide projector)
- This is simple, if we remember how the transformation pipeline works: M_s=scene transformation, M_v=view transformation of projector, M_{pp}=perspective projection of projector, h=homogeneous coordinate for perspective division
- We need a mapping (M_{d2t}) from normalized device coordinates (-1..1) to texture coordinates (0..1)





Projective Texture Maps

- For projective texture maps, texture coordinates (t) of vertex coordinates (v) are computed in such a way that they map to texels that are projected from a given perspective (like a slide projector)
- This is simple, if we remember how the transformation pipeline works: M_s=scene transformation, M_v=view transformation of projector, M_{pp}=perspective projection of projector, h=homogeneous coordinate for perspective division
- We need a mapping (M_{d2t}) from normalized device coordinates (-1..1) to texture coordinates (0..1)

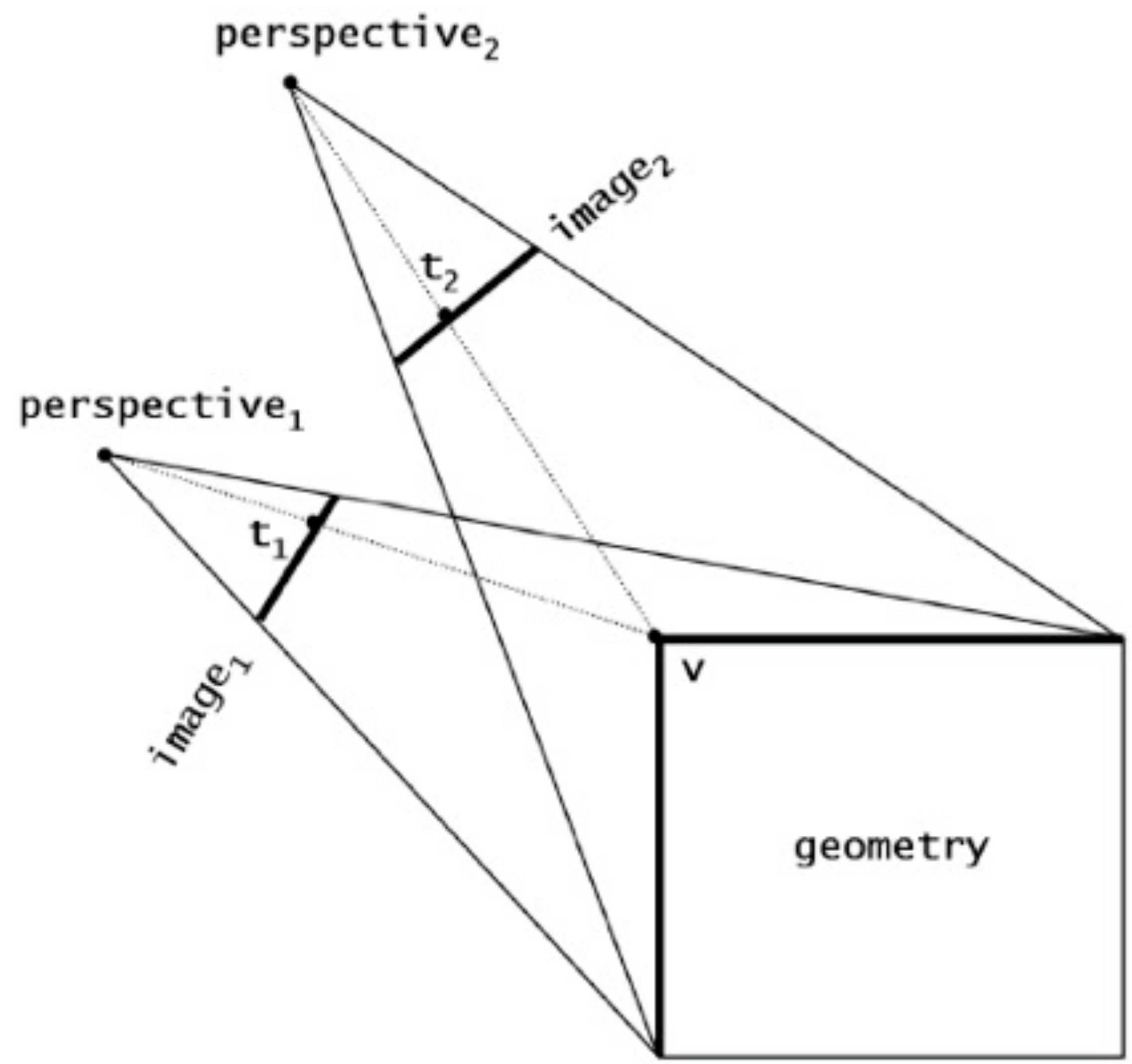


$$t = M_{d2t}(M_{pp}M_vM_sv)/h$$

$$M_{d2t} = \begin{bmatrix} 0.5 & 0 & 0.5 \\ 0 & 0.5 & 0.5 \\ 0 & 0 & 1 \end{bmatrix}$$

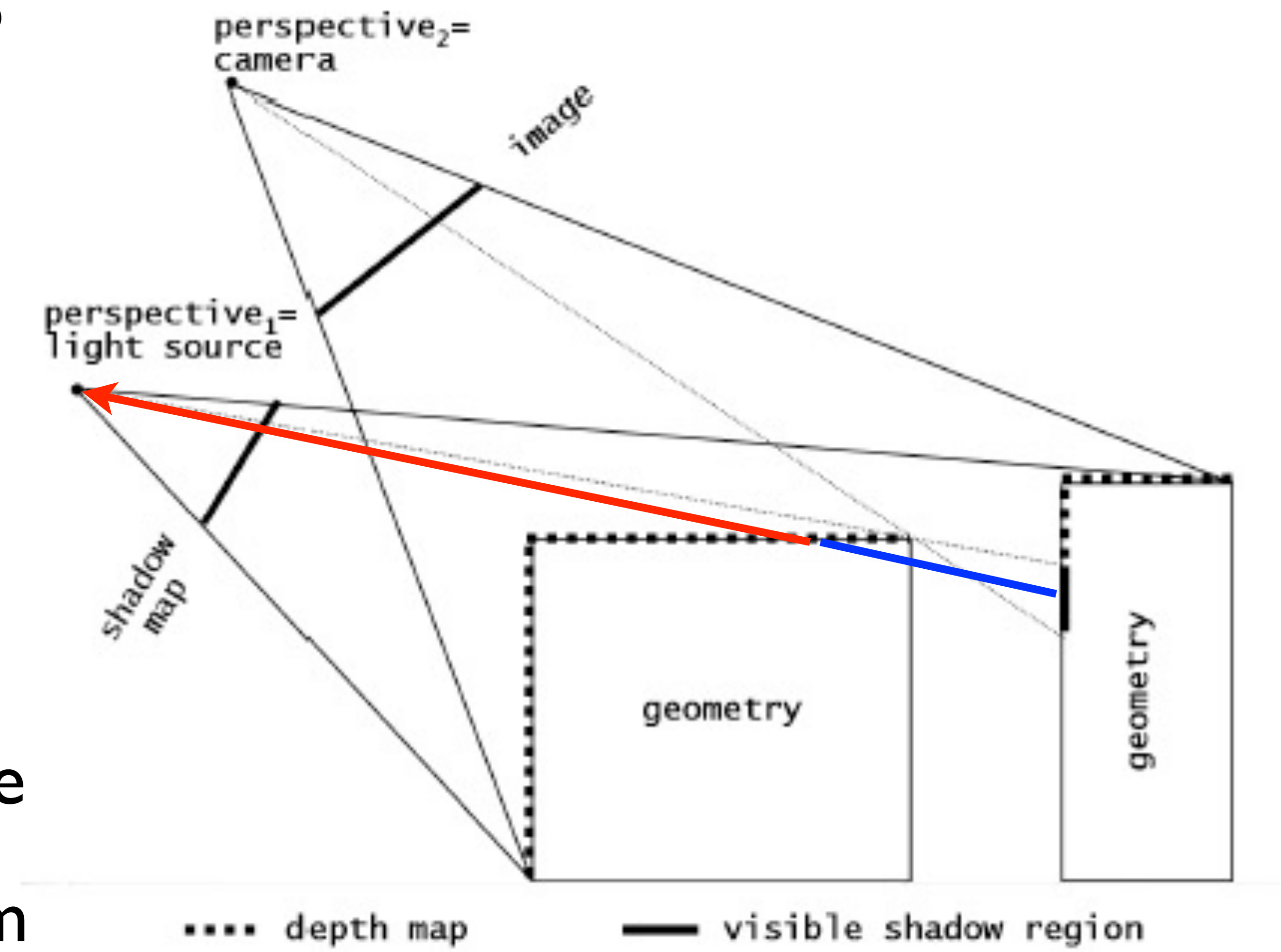
Projective Texture Maps

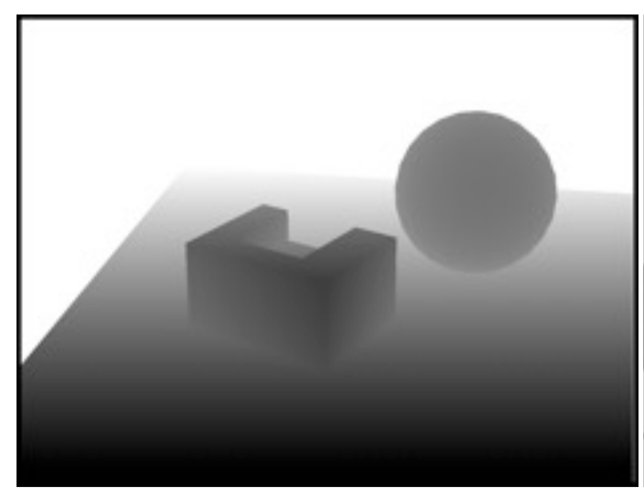
- Projective texture mapping can be used to warp one perspective into another one
  - rendering $image_1$ from $perspective_1$
  - compute projective texture coordinates for $perspective_1$ and assign them to the vertices
  - render scene from $perspective_2$, texture mapped with $image_1$ and the computed projective texture coordinates
- This does not make much sense, since the result is (almost) the same as directly rendered from $perspective_2$
- But $image_1$ can be an arbitrary texture



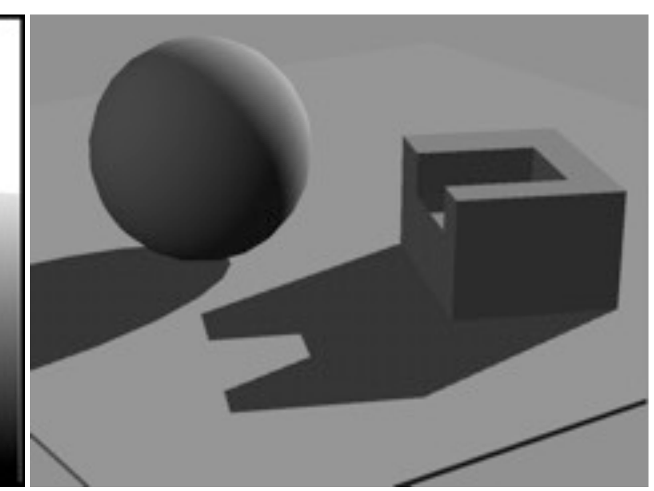
Shadow Maps

- But projective texture maps also enable casting hard shadows
- In this case, perspective$_1$ is the perspective of the light source, and image$_1$ contains only the z-value for this perspective (called „shadow map“ - although it is more a „light map“) - these z-values represent the distance (d_1) of all (for the light source visible) points to the light source
- Then, the scene is rendered from the perspective of the camera (perspective$_2$) - each, from the camera visible scene point is rastered- for these scene points, the distance (d_2) to the light source is also computed
- If d2>d1, then point is in shadow, else it is not





shadow map
1st pass



final rendering
2nd pass

Shadow Maps

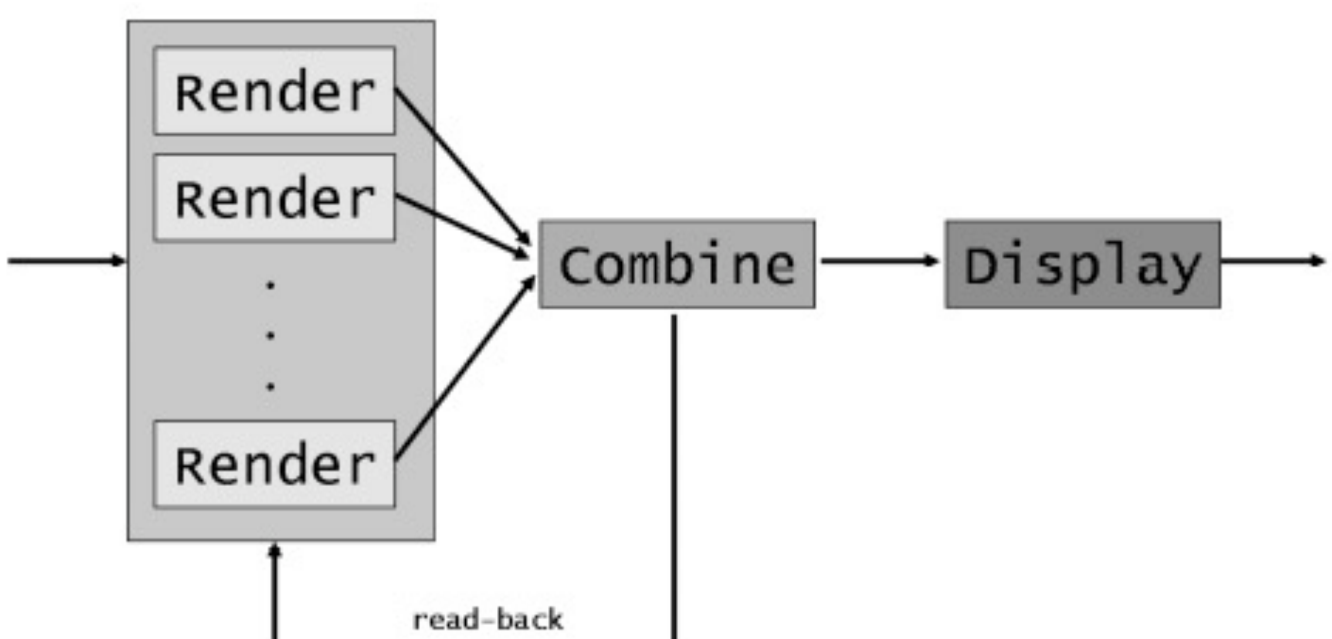
- But projective texture maps enable casting hard shadows
- In this case, perspective_1 is the perspective of the light source, and image_1 contains only the z-value for this perspective (called „shadow map“ - although it is more a „light map“) - these z-values represent the distance (d_1) of all (for the light source visible) points to the light source
- Then, the scene is rendered from the perspective of the camera (perspective_2) - each, from the camera visible scene point is rastered- for these scene points, the distance (d_2) to the light source is also computed
- If d2>d1, then point is in shadow, else it is not

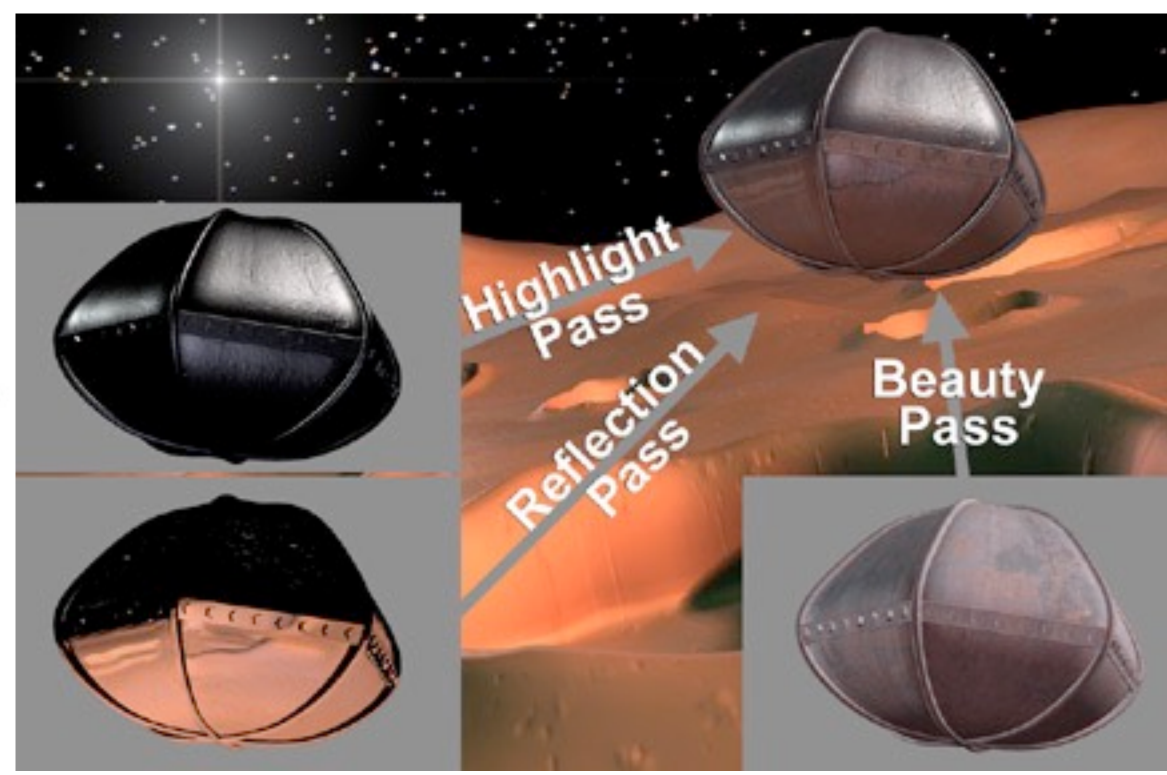
Note on 2D Textures

- We have seen that there are different types of 2D textures that can be realized with a 2D mapping technique
  - reflectance, normals, displacement, environment, shadows
- They all can be combined
  - multiple image textures (i.e., reflectance, shadows, environment) can be combined with alpha blending
  - in addition to the image textures, normals and displacements can be added
- Some techniques (environment maps, shadow maps, and blending of different textures) require more than one rendering step
- Textures can also be used to store other parameters that can be useful for rendering (not necessarily information that will be directly visible) - i.e., they can be used as general 2D matrix data structure

Multi-Pass Rendering

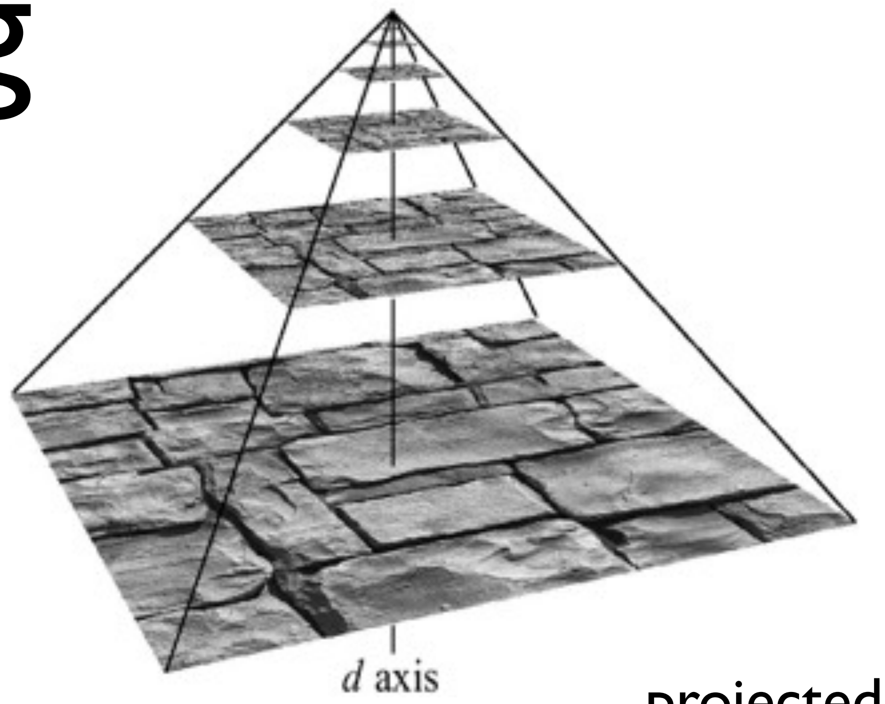
- In a simplest form, we have a single rendering pass (this is what we have done so far) - this is called „single-pass“ rendering
- To achieve more complex rendering effects, we render the scene multiple times off-screen (i.e., not displayed) into a texture
  - each image shows the scene from the same perspective, but renders different effects (e.g., shadows, environment reflections, complex lighting, etc.)
  - finally, the images that result from the different rendering passes are combined (e.g., with alpha blending) and the result is finally displayed
  - results from previous rendering passes can be used in following ones - they are transferred as textures
  - this is called „multi-pass“ rendering

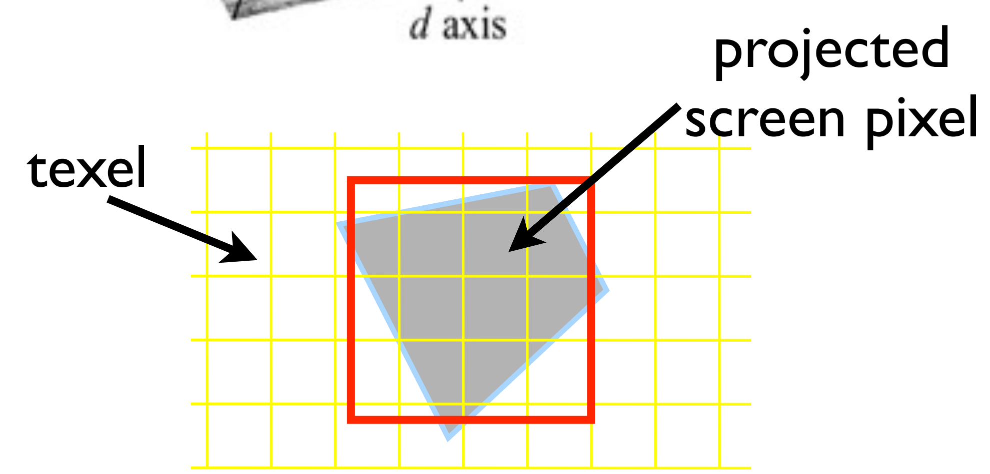


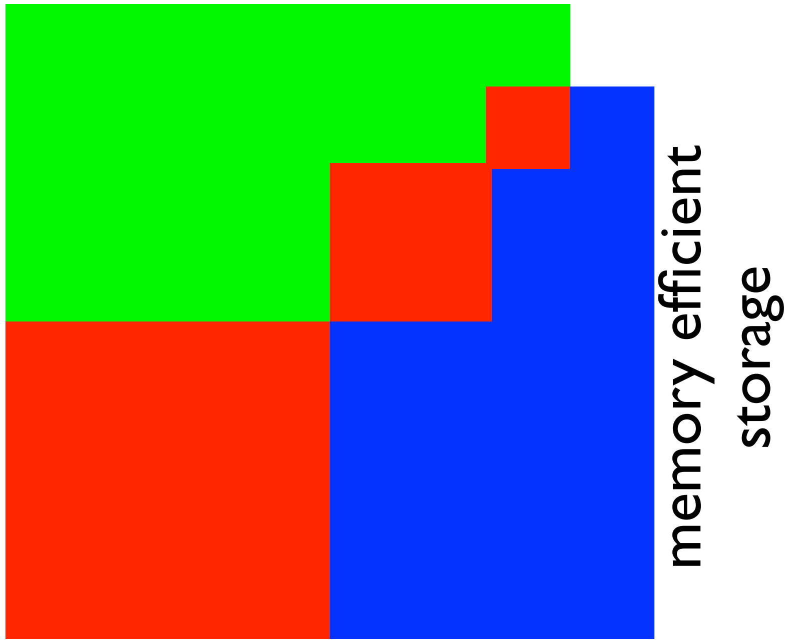


MIP Mapping

- The resolution of a texture actually matters
  - in terms of visual quality (high is better)
  - in terms of performance (low is better)
- A possible way to balance this is MIP mapping (multum in parvo = many things in one place)
- For each texture, multiple resolution versions are pre-computed (usually power-of-two edge lengths) - this results in an image pyramid
- Depending on how much of a projected screen pixel is mapped to texels, the appropriate level of detail (d) is chosen for texturing (depends on distance to camera)
  - if camera is close, high resolution textures provide details
  - if camera is far, low resolution textures are good enough (fine details will not be visible anyway)
  - this always provides optimal rendering performance
- Efficient ways of storing MIP-maps exist

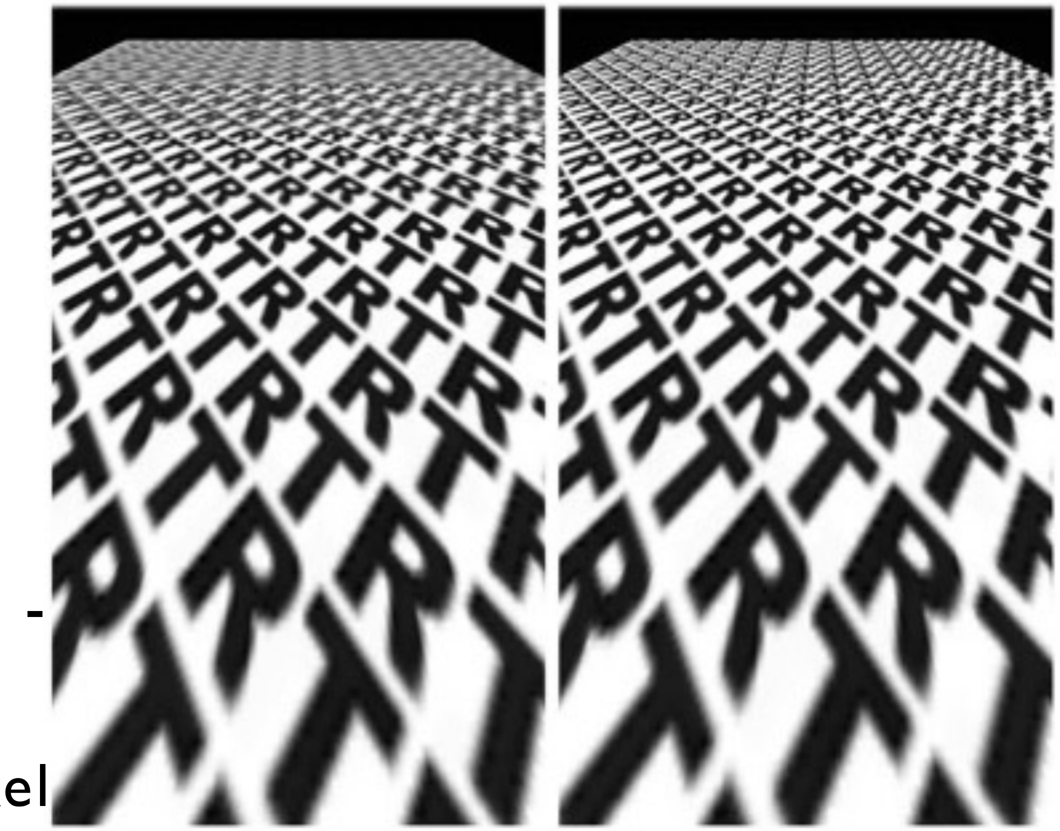






MIP Mapping

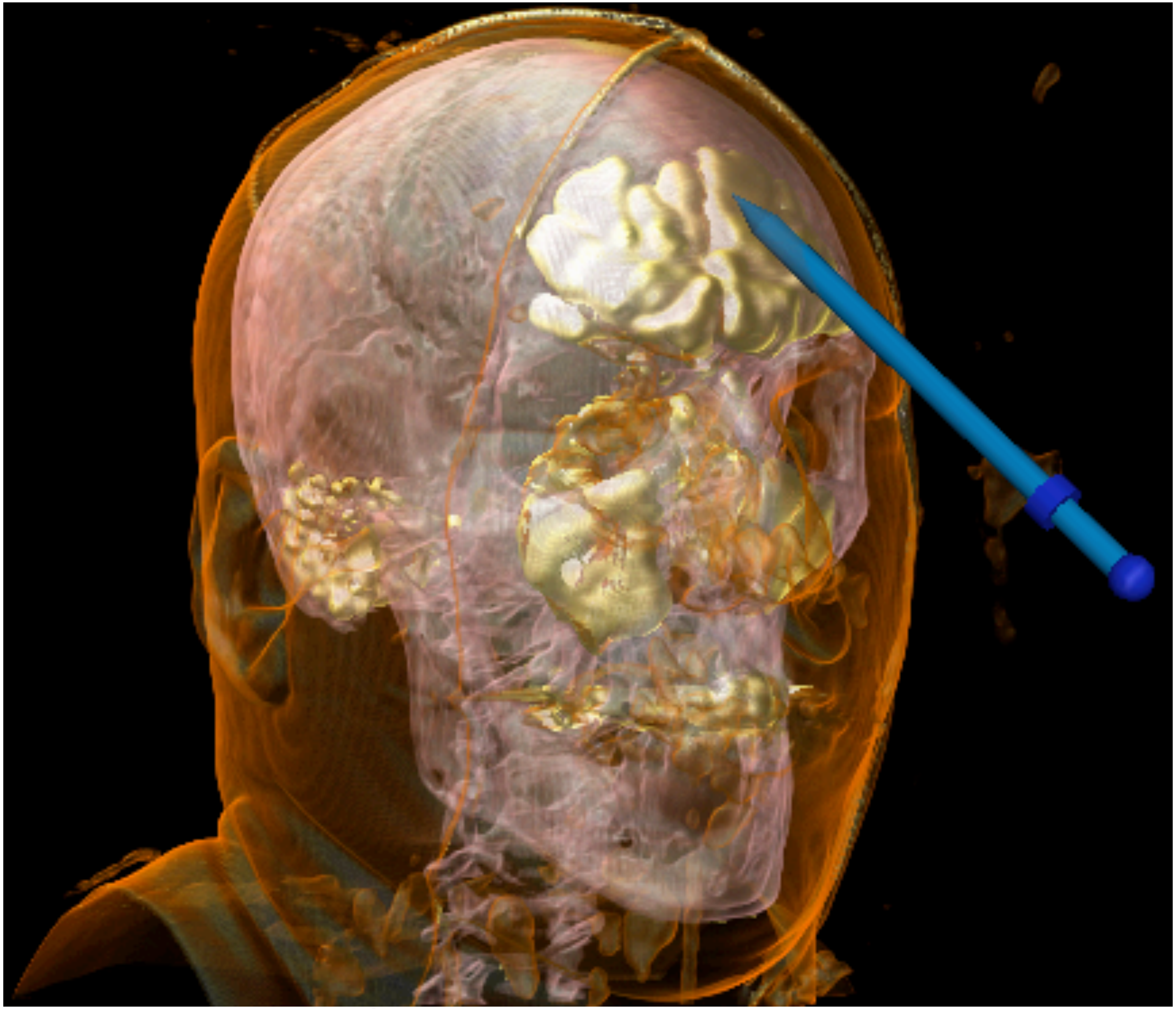
- The resolution of a texture actually matters
    - in terms of visual quality (high is better)
    - in terms of performance (low is better)
- A possible way to balance this is MIP mapping (multum in parvo = many things in one place)
- For each texture, multiple resolution versions are pre-computed (usually power-of-two edge lengths) - this results in an image pyramid
- Depending on how much of a projected screen pixel is mapped to texels, the appropriate level of detail (d) is chosen for texturing (depends on distance to camera)
    - if camera is close, high resolution textures provide details
    - if camera is far, low resolution textures are good enough (fine details will not be visible anyway)
    - this always provides optimal rendering performance
- Efficient ways of storing MIP-maps exist



3D Textures

- Although, 2D textures are very common, 3D textures also exist (sometimes referred to a solid textures)
- With 2D textures, we need to find a mapping from 2D texture coordinates to 3D surface coordinates
- For 3D textures, the mapping is simpler (3D to 3D)
- 3D textures are often described in form of analytical functions or algorithms (procedural textures)
- They can be used to model natural material, such as wood, water, noise materials, etc.

Example: Volume Rendering



Course Schedule

Type	Date	Time	Room	Topic	Comment
C1	01.03.2016	13:45-15:15	HS 18	Introduction and Course Overview	Conference
C2	15.03.2016	13:45-15:15	HS 18	Transformations and Projections	Easter Break
C3	05.04.2016	13:45-15:15	HS 18	Raster Algorithms and Depth Handling	
C4	12.04.2016	13:45-15:15	HS 18	Local Shading and Illumination	
C5	19.04.2016	13:45-15:15	HS 18	Texture Mapping Basics	
C6	26.4.2016	13:45-15:15	HS 18	Advanced Texture Mapping & Graphics Pipelines	
C7	03.05.2016	13:45-15:15	HS 18	Intermediate Exam	
C8	09.05.2016	17:15-18:45	HS 18	Global Illumination I: Raytracing	
C9	10.05.2016	13:45-15:15	HS 18	Global Illumination II: Radiosity	Conference / Holiday
C10	31.05.2016	13:45-15:15	HS 18	Volume Rendering	
C11	07.06.2016	13:45-15:15	HS 18	Scientific Data Visualization	
C12	14.06.2016	13:45-15:15	HS 18	Curves and Surfaces	
C13	21.06.2016	13:45-15:15	HS 18	Basics of Animation	
C14	28.06.2016	13:45-15:15	HS 18	Final Exam	
C15	04.10.2016	13:45-15:15	TBA	Retry Exam	

Thank You!