

CSC 433/533

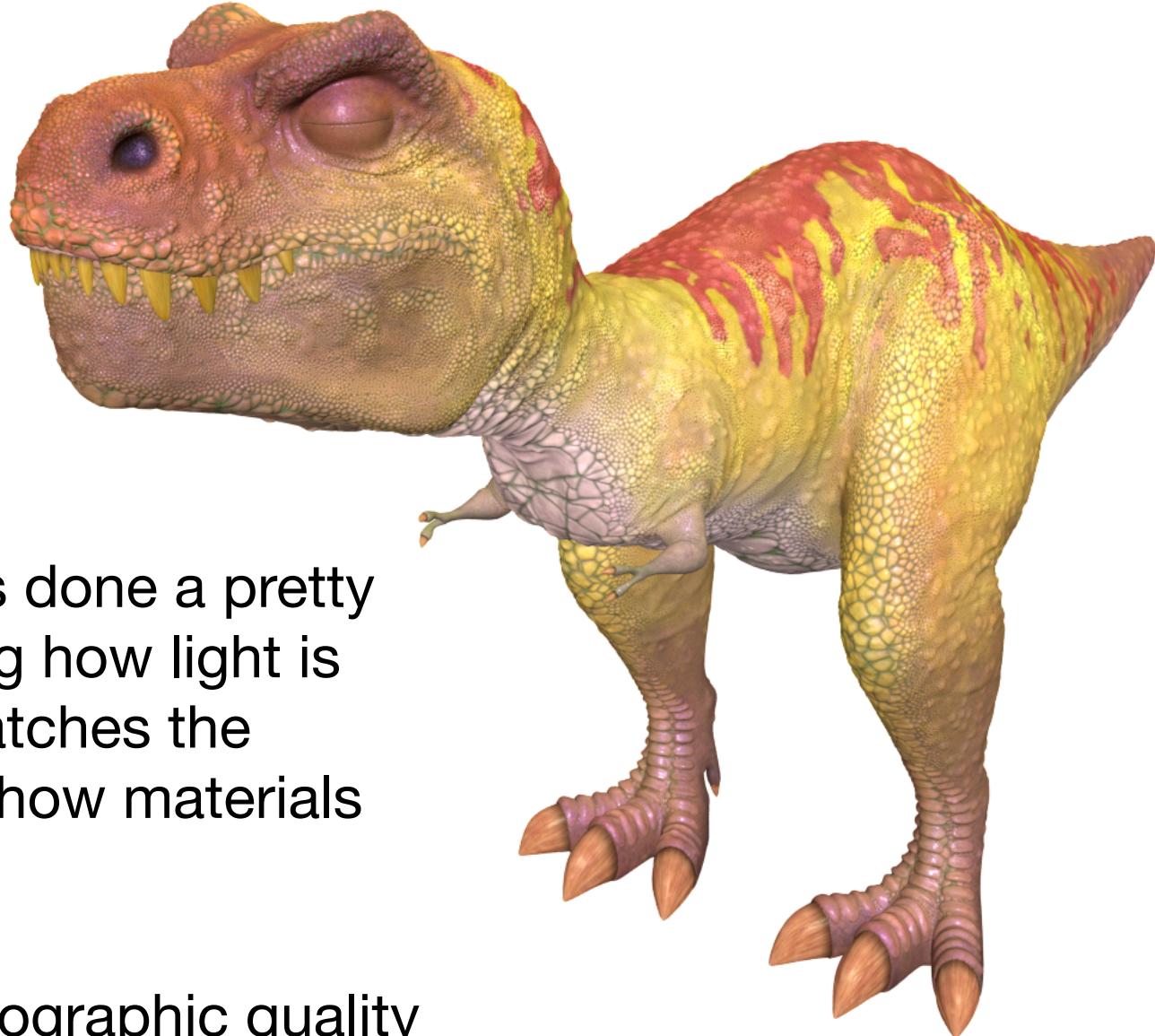
Computer Graphics

Alon Efrat
Credit: Joshua Levine

Textures

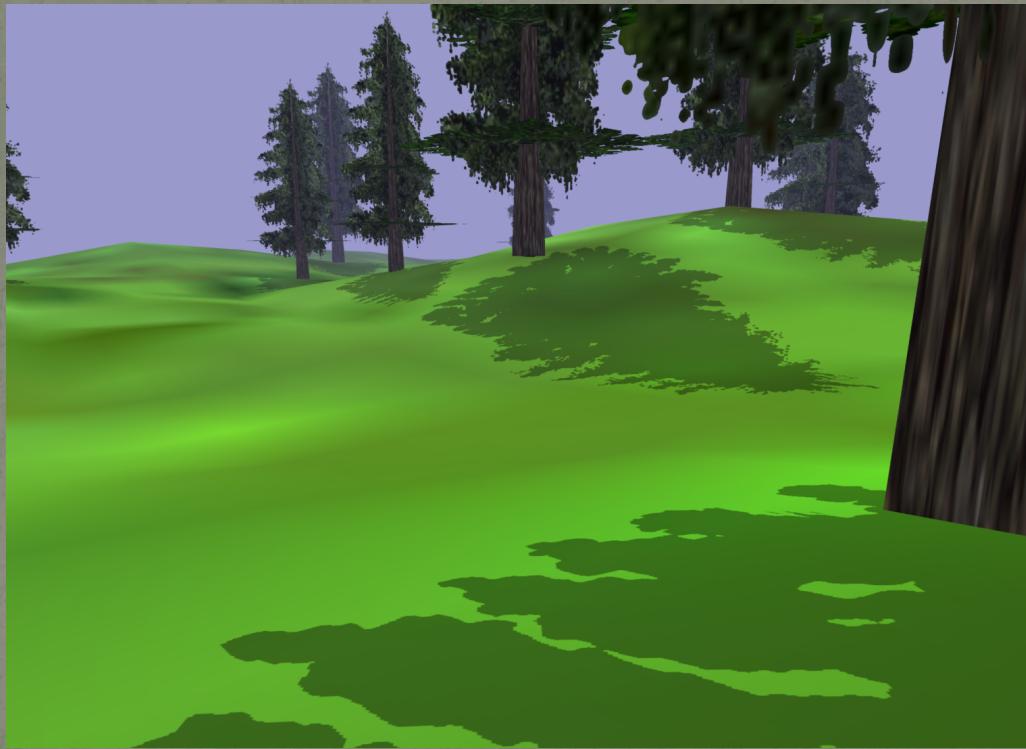
Challenge: Real World Surfaces Have Complex Materials

- While ray tracing has done a pretty good job of capturing how light is modeled, it only scratches the surface at modeling how materials look
- Goal: Replicate photographic quality by varying shading parameters?



Texture Mapping

Shadow Mapping



Normal Mapping



Texture Mapping

- Models attributes of surfaces that **vary as position changes**, but do not affect the shape of the surface.
- Examples: wood grain, wrinkles in skin, woven structures in cloth, defects in metal surfaces, patterns (in general), ...

Texture Maps

- Idea: model this variation using an image, called a **texture map** (or, sometimes “texture image” or just “texture”)
- The texture map stores the surface details
 - Typically, shading parameters like k_d and k_s
- Can be used in lots of interesting ways to achieve complex effects



Texture Lookups

- Since the texture is an image, we need a way to index into it given a surface position
 - Or, where on the surface does the image go?
- Given a position, we lookup the **texture coordinates**, given as (u,v) values that refer to positions in the image
 - Easy to define for some shapes, can be very hard for others

Computing Texture Coordinates

- Idea: We will model this problem using a **texture coordinate function**,

$$\phi: S \rightarrow T, \text{ for all } (x,y,z) \in S \text{ and } (u,v) \in T$$

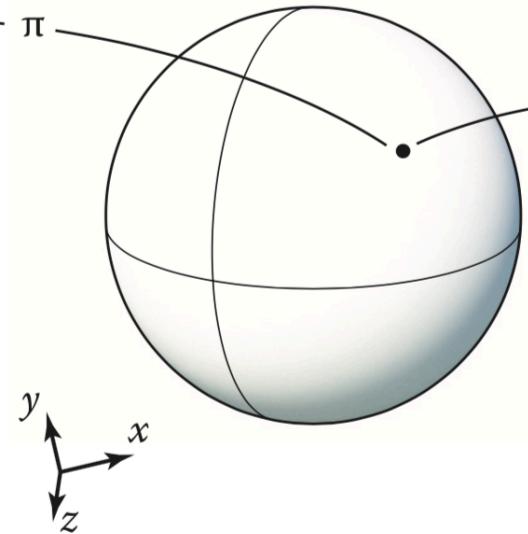
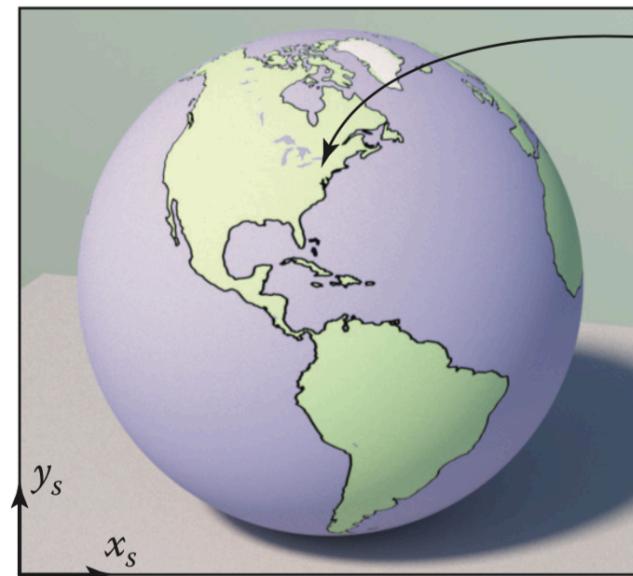
- When shading a point (x,y,z) , we compute $\phi(x,y,z)$ to get the appropriate pixel (u,v) in the texture.
- u and v normally values in $[0,1]$ (and then are scaled to the size of the texture)

Computing Texture Coordinates - Example

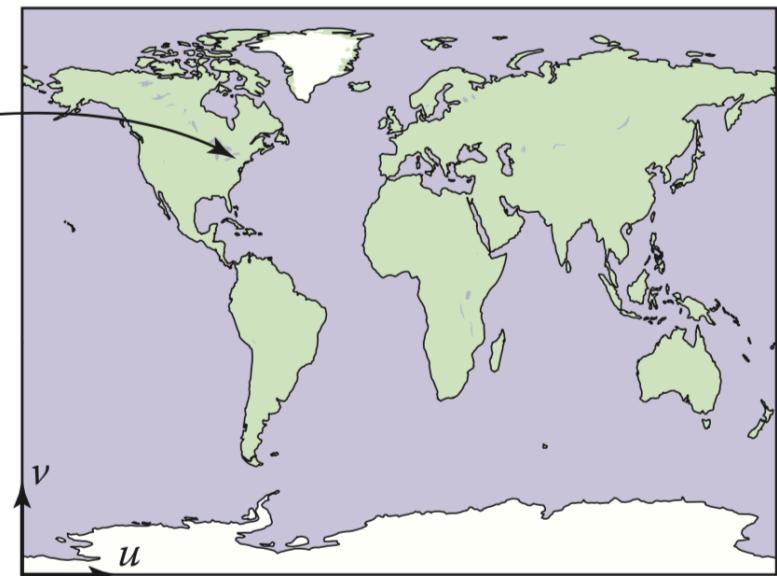
```
function texture_lookup(tex, u, v) {  
    let i = Math.round(u * tex.width() - 0.5);  
    let j = Math.round(v * tex.height() - 0.5);  
    return tex.get_pixel(i,j);  
}  
  
function shade_surface_point(surf, pt, tex) {  
    let normal = surf.get_normal(pt);  
    [u,v] = surf.get_texcoord(pt);  
    let diffuse_color = texture_lookup(tex,u,v);  
    //compute shading using diffuse_color and normal  
    //return shading result  
}
```

Three Spaces

- Just like we have mappings from world space to image, we use ϕ as another mapping



Surface S in world space



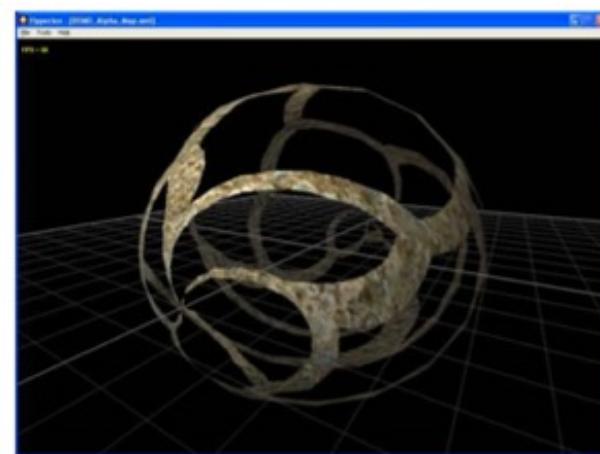
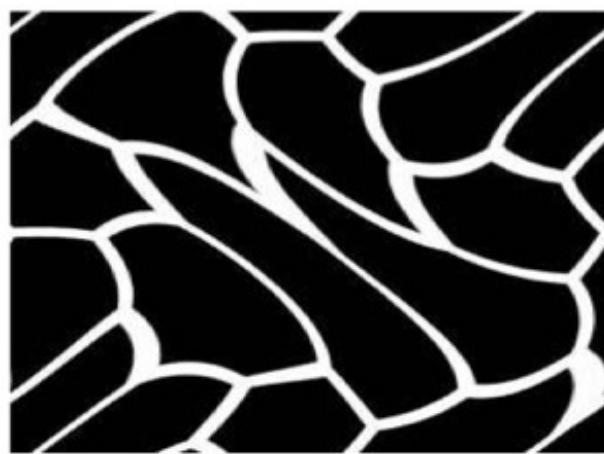
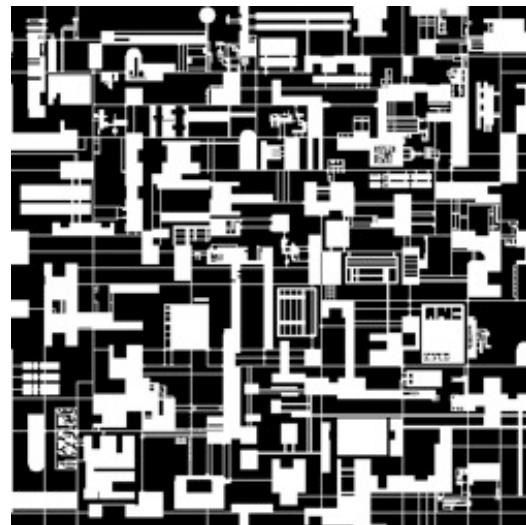
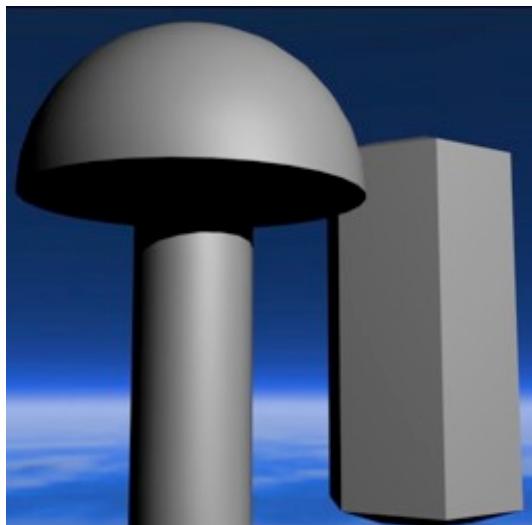
Texture space, T

Examples of Texture Coordinate Functions

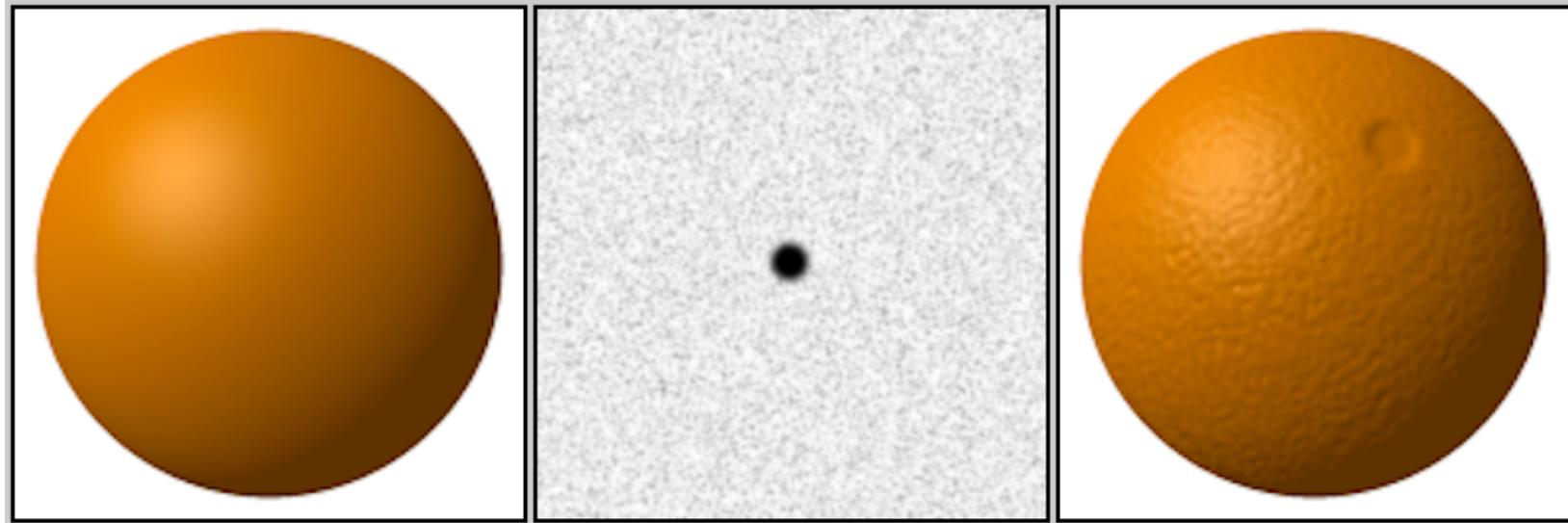
Problem #1: Defining Texture Coordinate Functions

- Defining ϕ can be very difficult for complex shapes
- Similar to the problem of taking a surface and flattening it
 - e.g. Cartographers problem
- Inevitably will have distortion of areas, angles, or distances

Opacity mapping

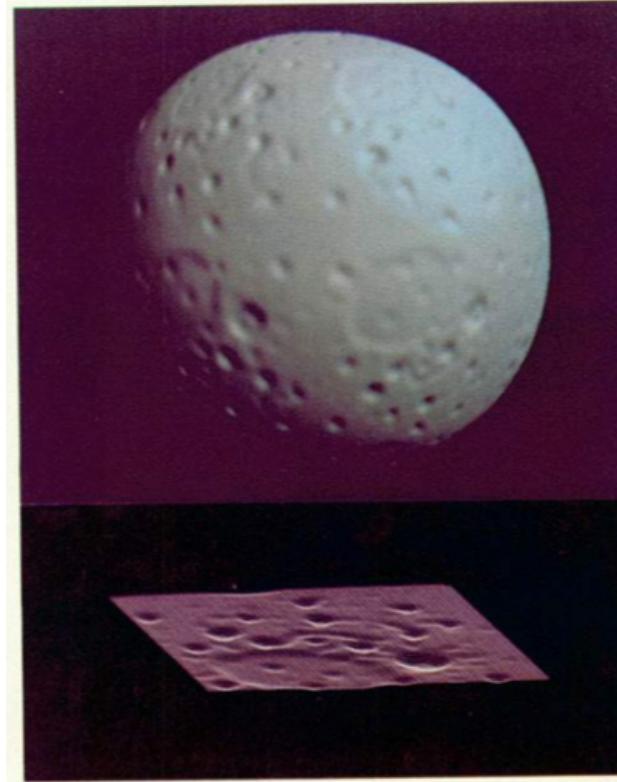
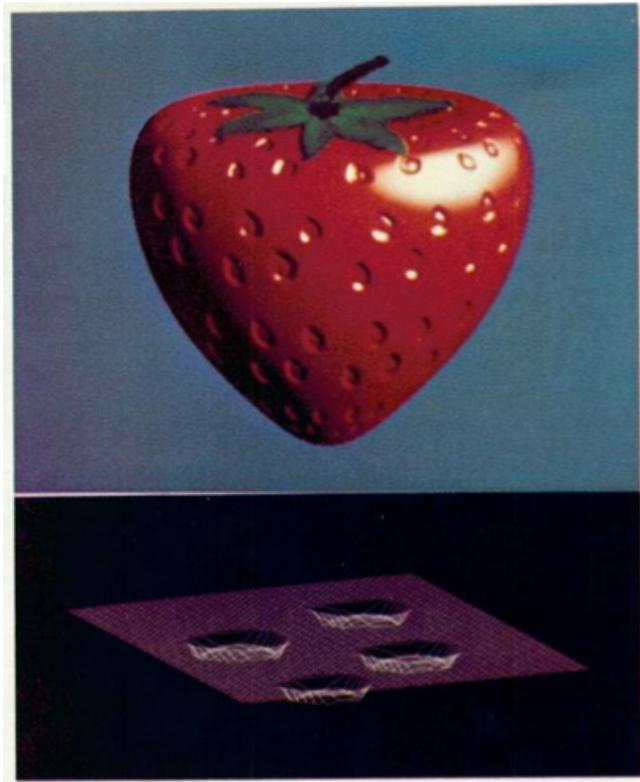


Bump mapping



- Simulates roughness (“bumpiness”) of a surface without adding geometry
- Uses a two-dimensional height field (bump map) to perturb the normal during per-fragment shading calculations
- Limitations: the mapping of texture onto the surface is unaffected; silhouette is also unaffected.

Bump mapping



- Simulates roughness (“bumpiness”) of a surface without adding geometry
- Uses a two-dimensional height field (bump map) to perturb the normal during per-fragment shading calculations
- Limitations: the mapping of texture onto the surface is unaffected; silhouette is also unaffected.

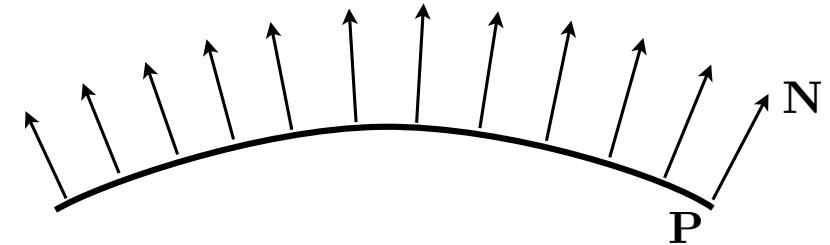
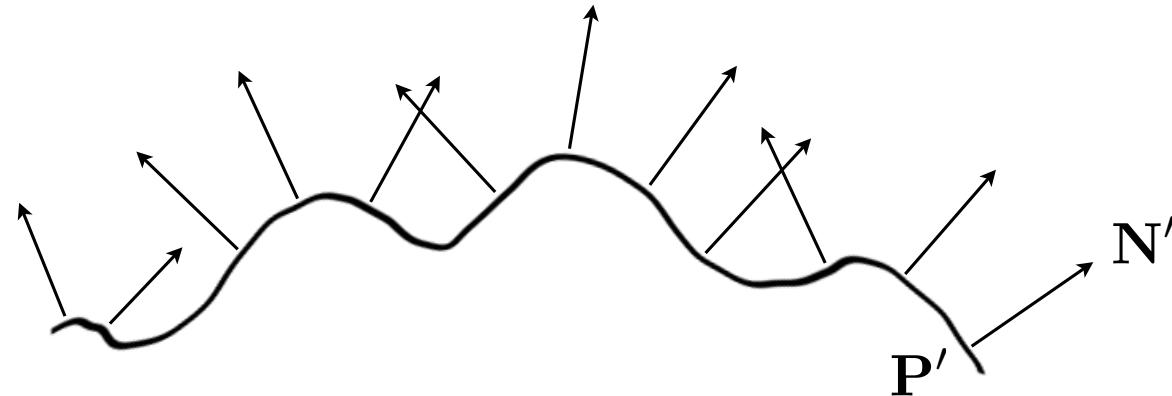
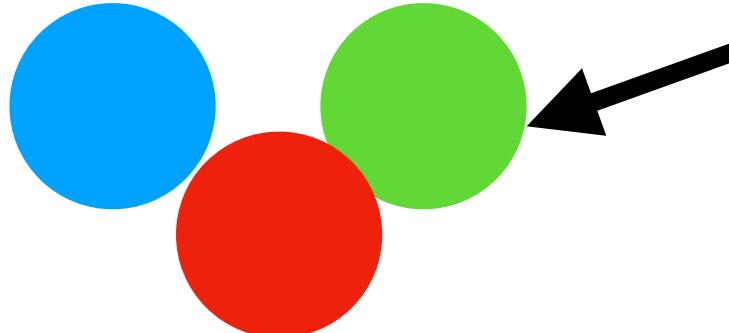
Important

When we perform ray-tracing/z-buffering to find the first object hits by a ray, the texture does not change the answer.

For example, we still treat a sphere as. a smooth normal sphere.

We use the bumps map only after computing

- 1) The first sphere hit, and
- 2) The hitting point (with some exceptions here)



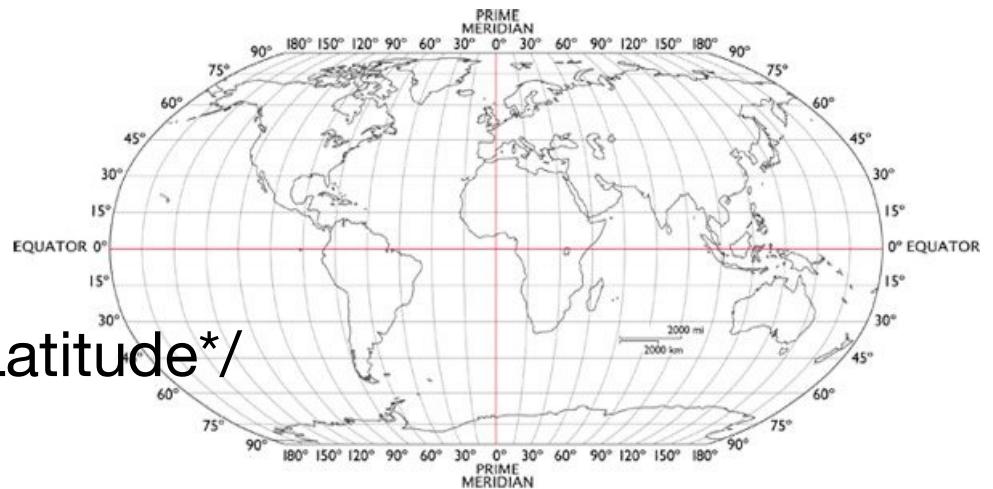
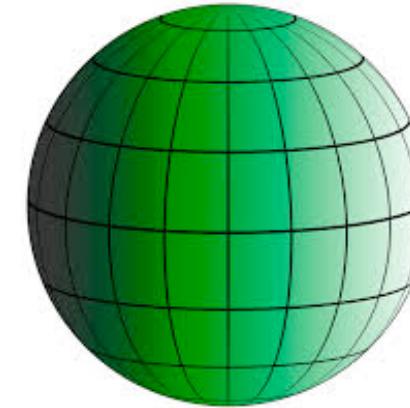
Spherical Projection

- Convert (x,y,z) to spherical coordinates, discard radius

$$r = \sqrt{x^2 + y^2 + z^2}$$

$$u = (\pi + \text{atan2}(y,x)) / (2\pi)$$

$$v = (\pi + \text{acos}(z/r)) / (\pi) /*\text{Latitude}*/$$



- Similar to casting a ray outward from center

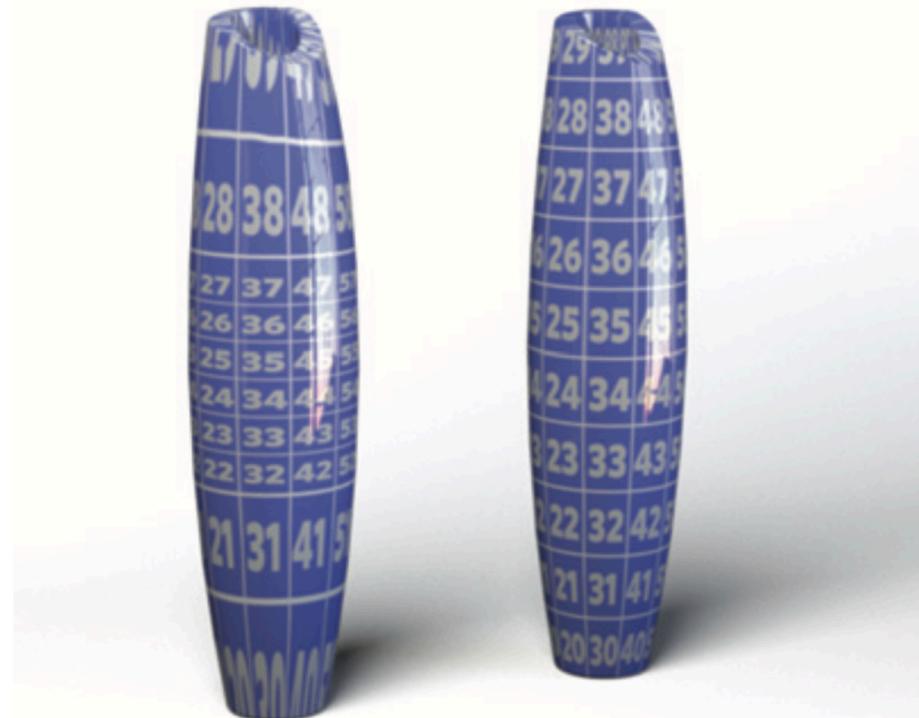
Cylindrical Projection

- Convert (x,y,z) to cylindrical coordinates, discard radius

$$u = (\pi + \text{atan2}(y,x)) / (2\pi)$$

$$v = 0.5 + z/2$$

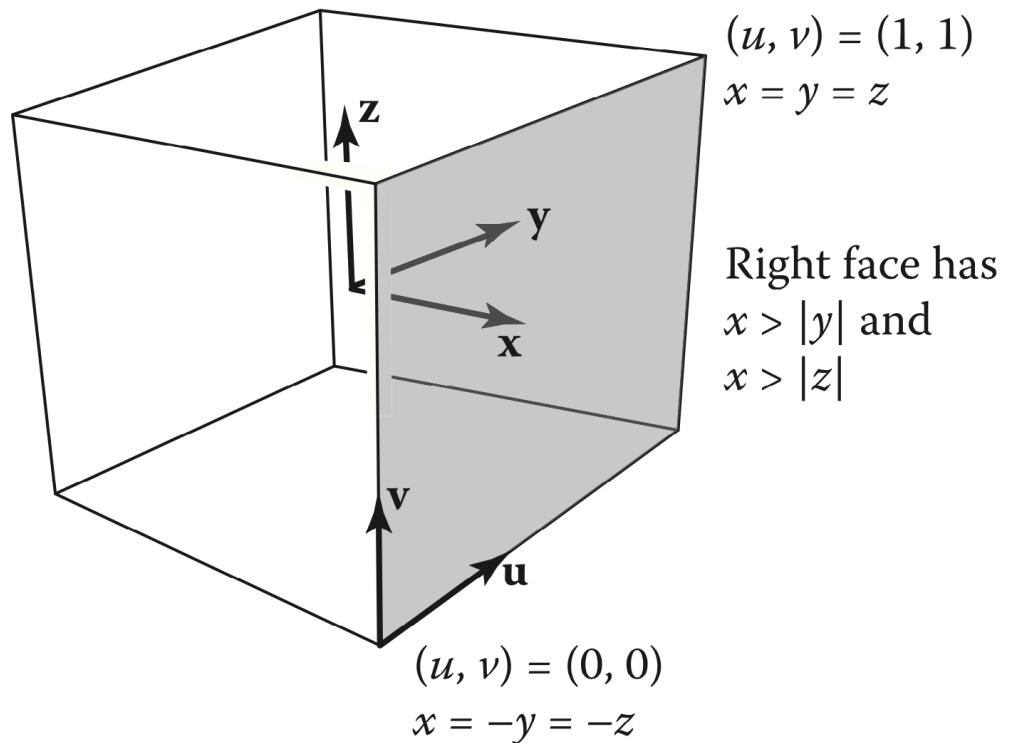
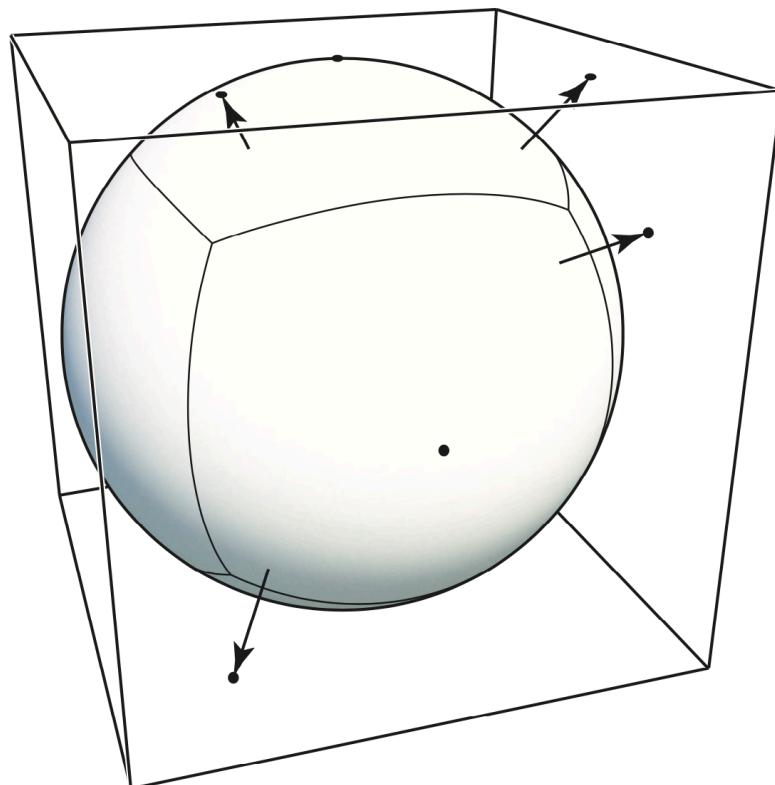
Spherical



Cylindrical

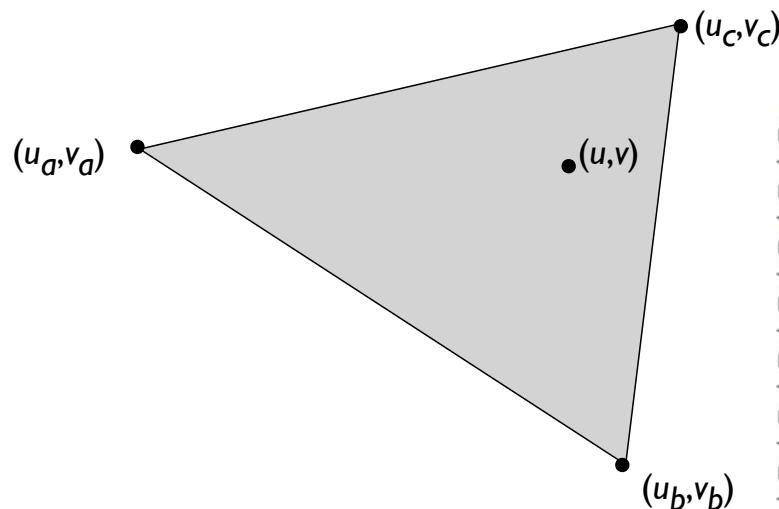
Cubemaps

- Project onto faces of a cube, 6 planar projections
- Seams can be tricky to handle.

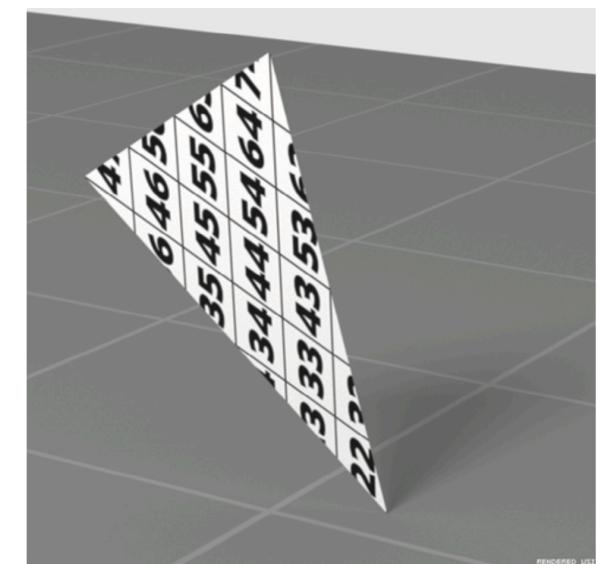


Interpolated Texture Coordinates

- Explicitly store (u,v) coordinates on the vertices of a triangle mesh, interpolate in the center using barycentric coordinates
- Texture coordinates just another per-vertex data. How to compute them? Can be difficult!



09	19	29	39	49	59	69	79	89	99
08	18	28	38	48	58	68	78	88	98
07	17	27	37	47	57	67	77	87	97
06	16	26	36	46	56	66	76	86	96
05	15	25	35	45	55	65	75	85	95
04	14	24	34	44	54	64	74	84	94
03	13	23	33	43	53	63	73	83	93
02	12	22	32	42	52	62	72	82	92
01	11	21	31	41	51	61	71	81	91
00	10	20	30	40	50	60	70	80	90

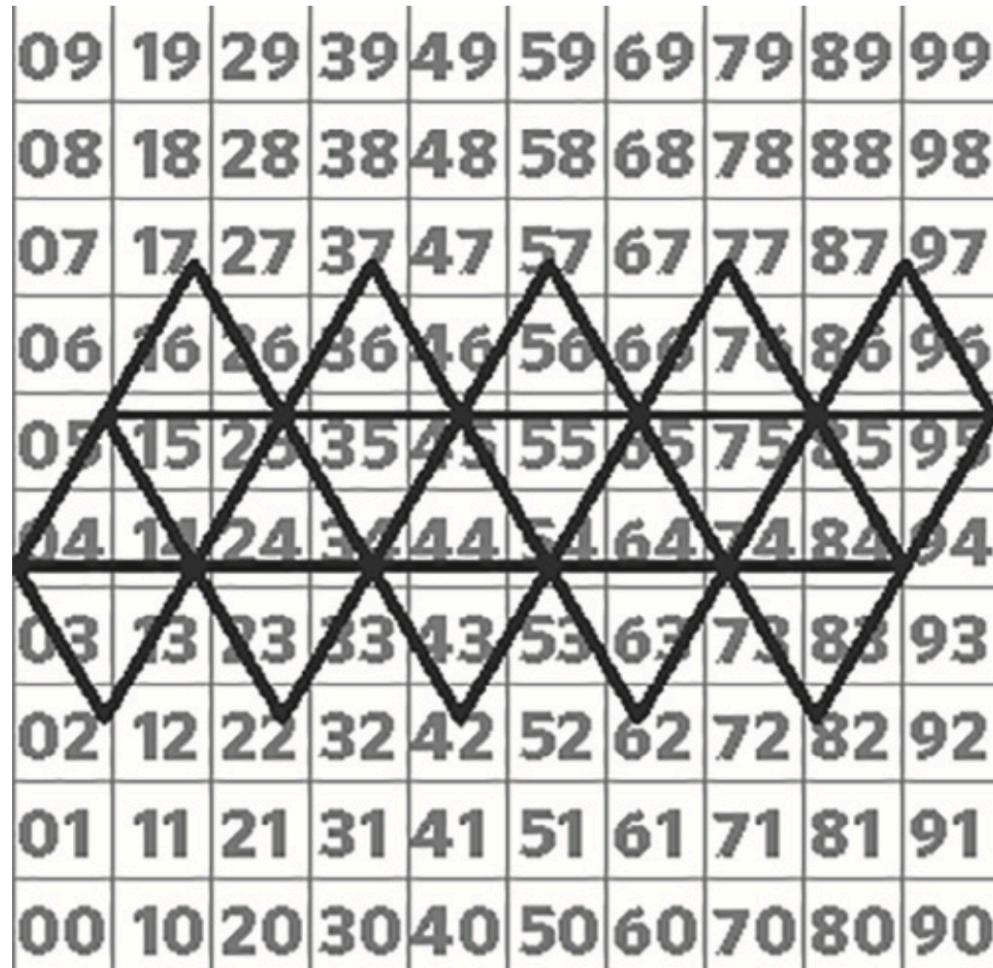


Properties of Texture Coordinate Functions

Goals for Texture Functions

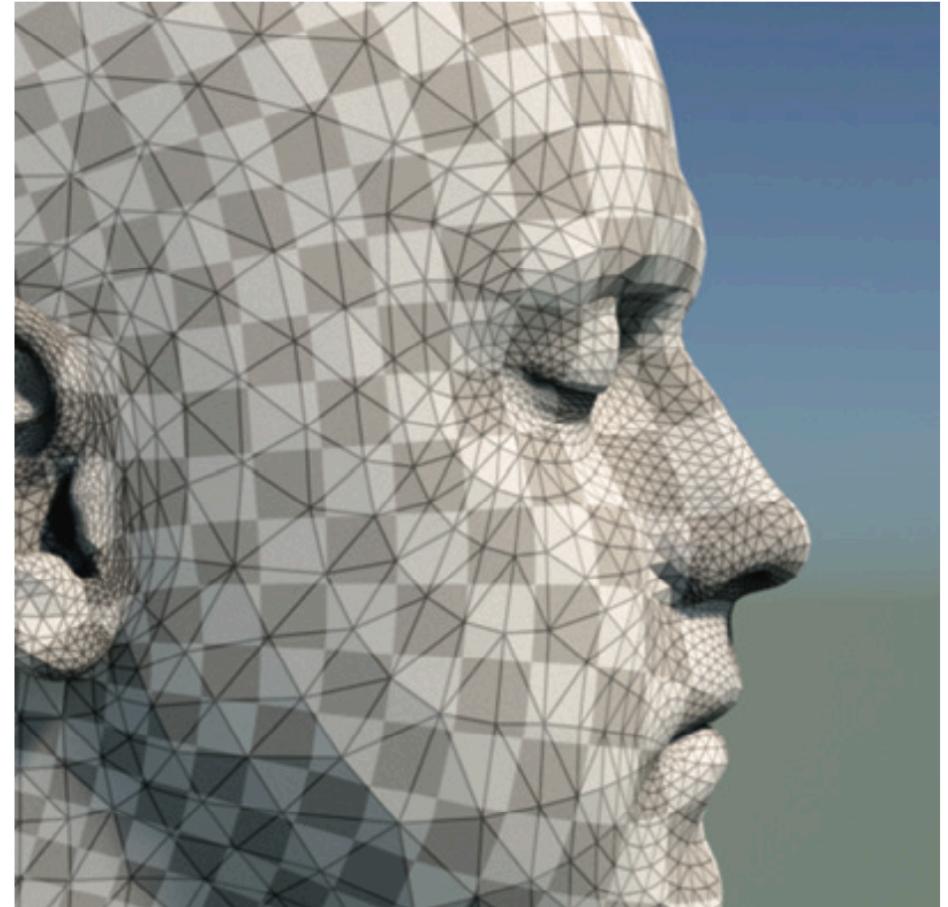
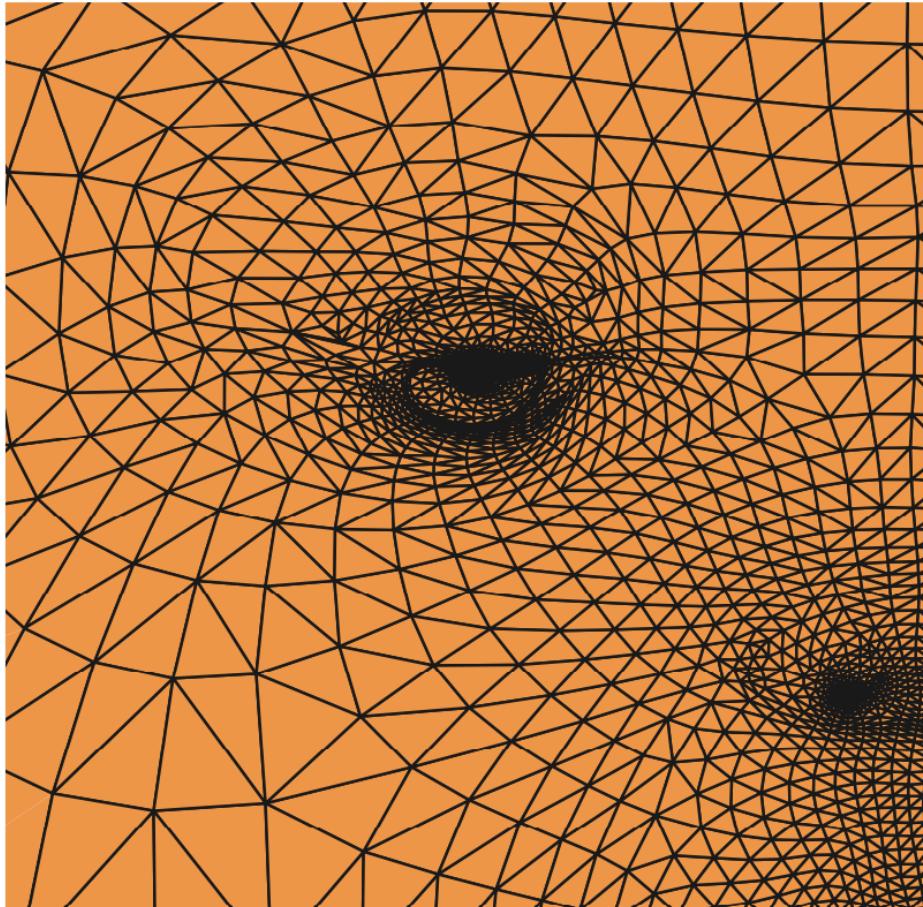
1. One-to-one vs one-to-many:
 - Each point on the surface should map to a different point on the texture, unless you want repetition
2. Size distortion:
 - Scale of texture kept constant across the surface
3. Shape distortion:
 - Shapes/Angles in the texture should state similarly shaped
4. Continuity:
 - Are there visible seams? ϕ should have as few discontinuities as possible

Distortions vs. Discontinuities



No distortion to area,
Many discontinuities

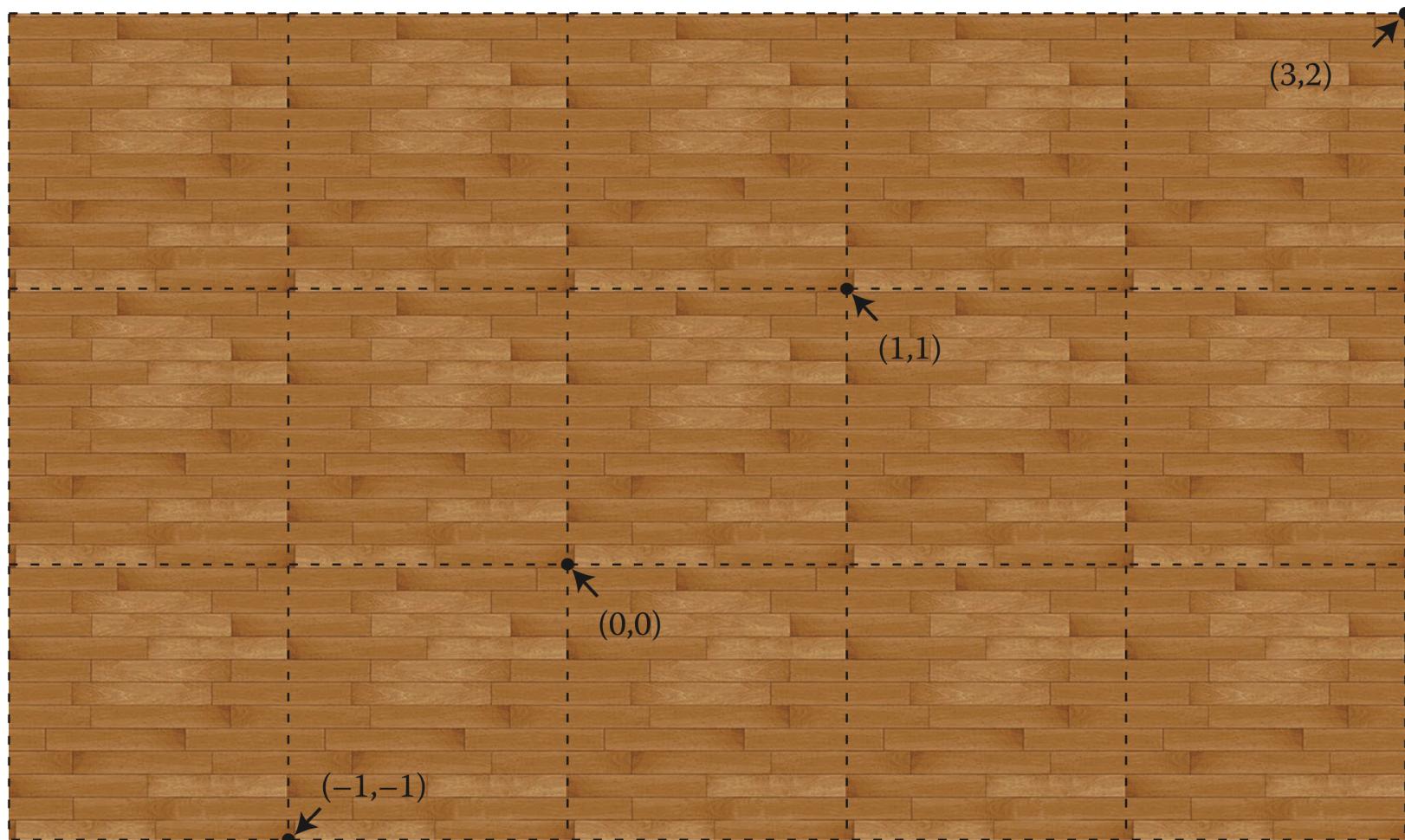
Shape vs. Area Distortions



**Low shape distortion,
Moderate area distortion**

Tiling and Wrapping

- Can be achieved by modifying the mapping to cycle around in various ways (similar to boundary conditions for image processing)
- Could also just clamp values

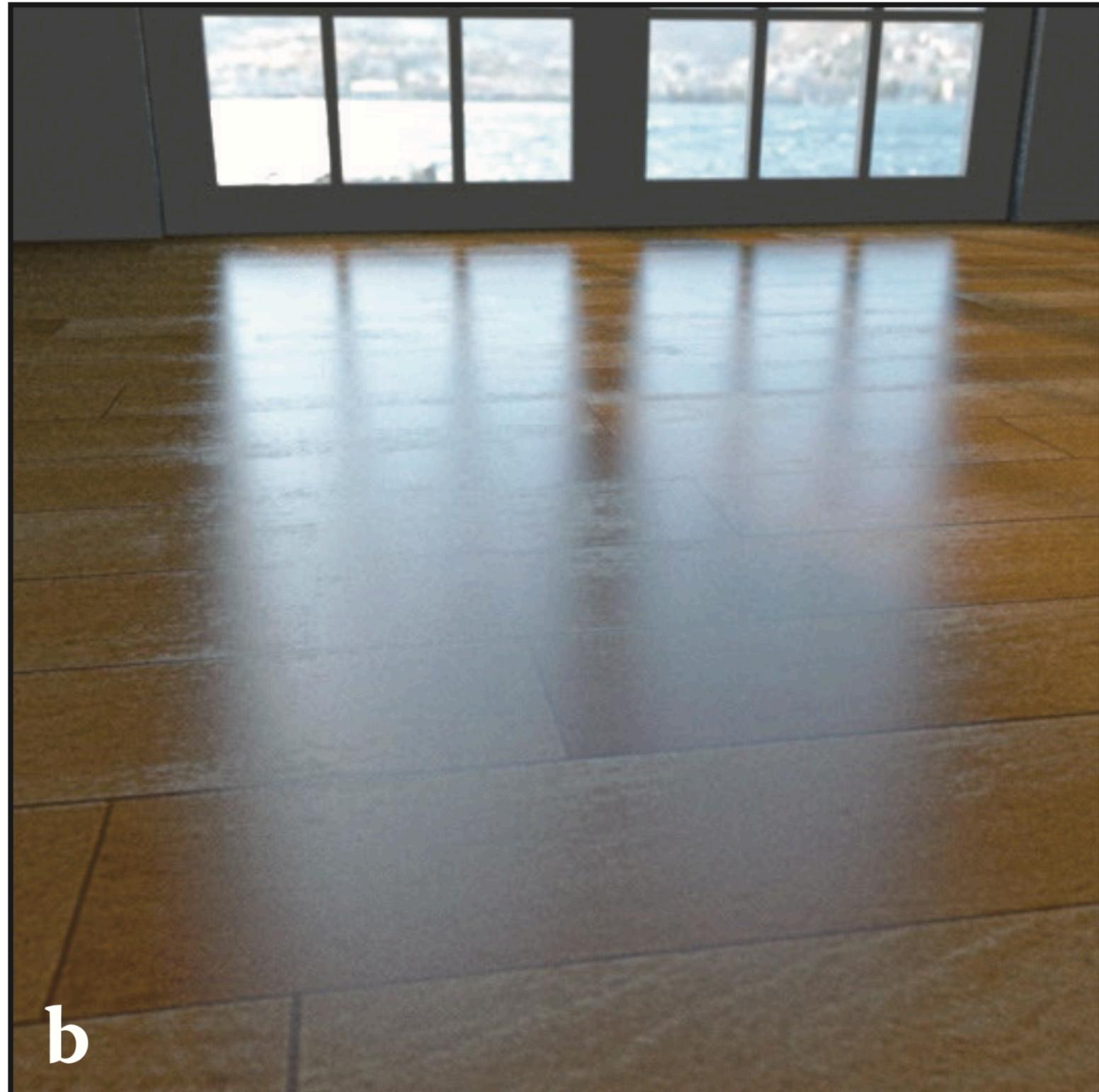
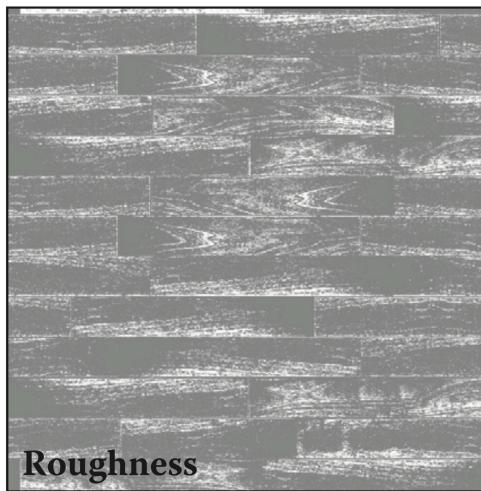


Applications of Textures

Controlling Shading Parameters

- Can look up k_d and k_s , or both



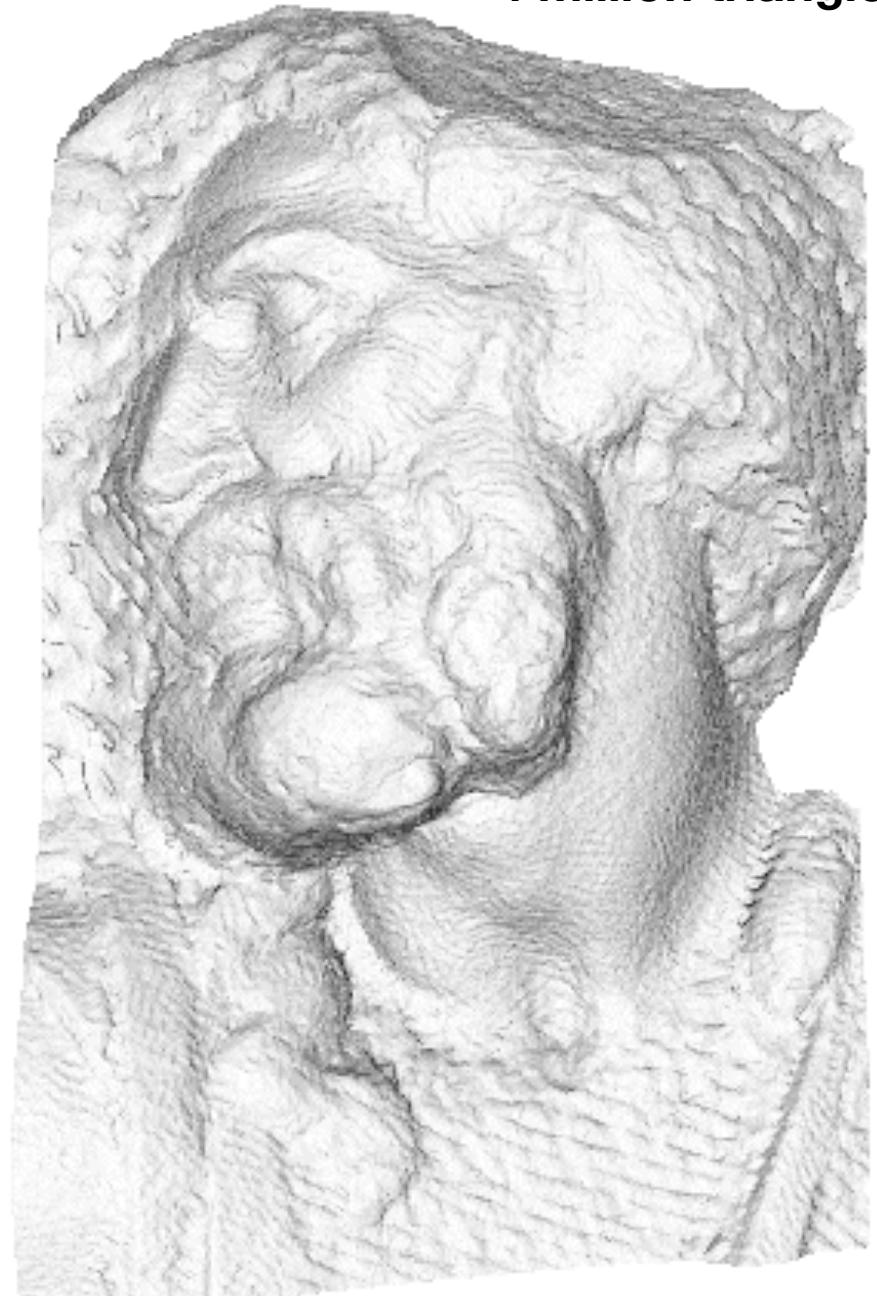


Normal Maps

- Can also use textures to look up normal information for the surface
- Typically, store the normal vector (n_x, n_y, n_z) as (r,g,b) values for the pixel
- Problem: orientation of the surface could change – normals are usually defined relative to a local coordinate space

Normal Map Example

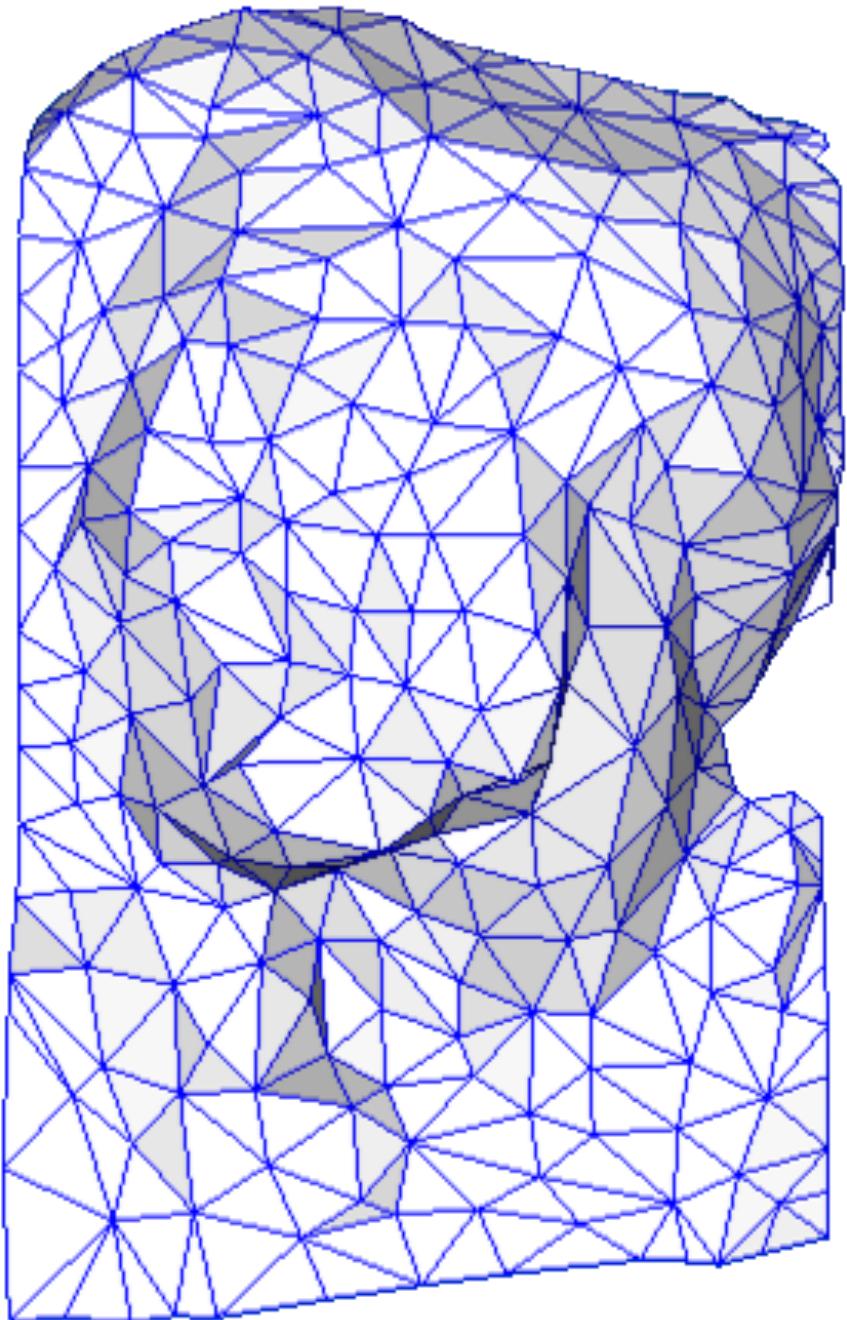
4 million triangles



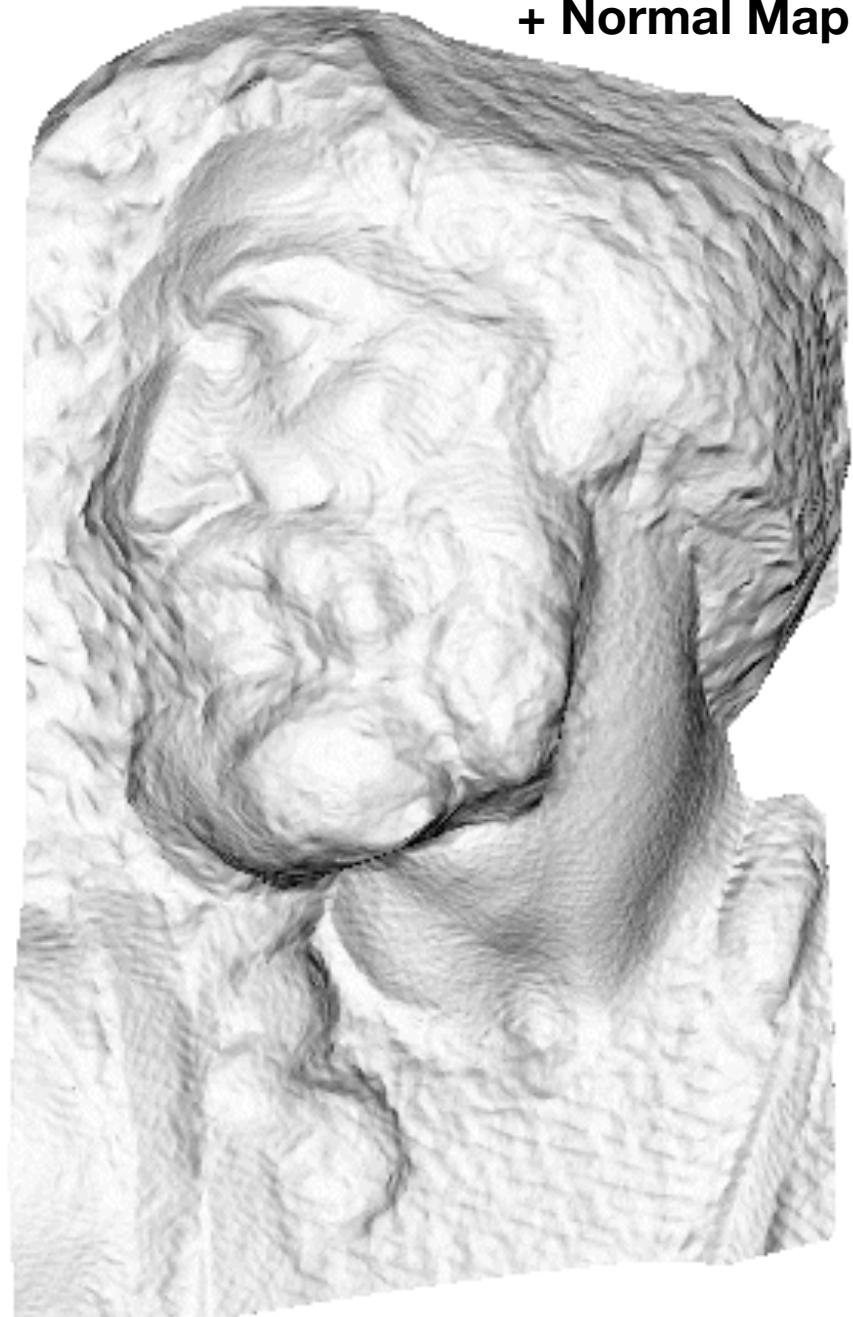
- Transfer details from high resolution mesh to normal map image

https://en.wikipedia.org/wiki/Normal_mapping

500 triangles



**500 triangles
+ Normal Map**



https://en.wikipedia.org/wiki/Normal_mapping

Bump Maps

- Normals specified indirectly using a height field
 - The bump map encodes a local offset of the detailed surface above the original smooth surfaces
- Normal map is derived by taking the derivative of the bump map

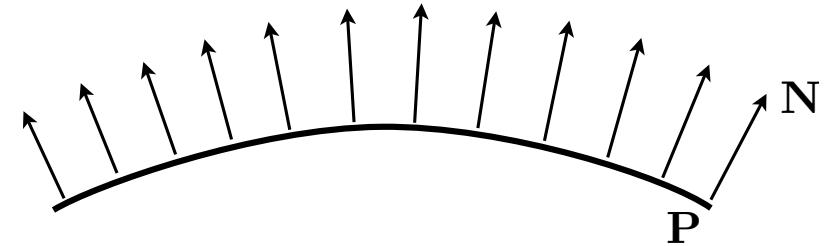
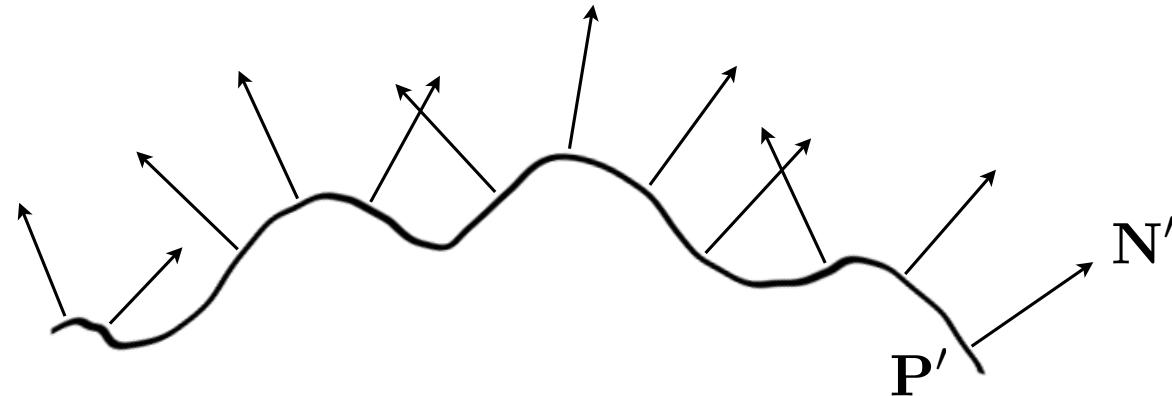
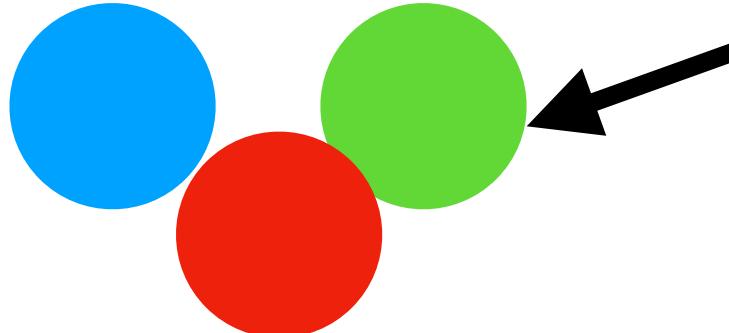
Important

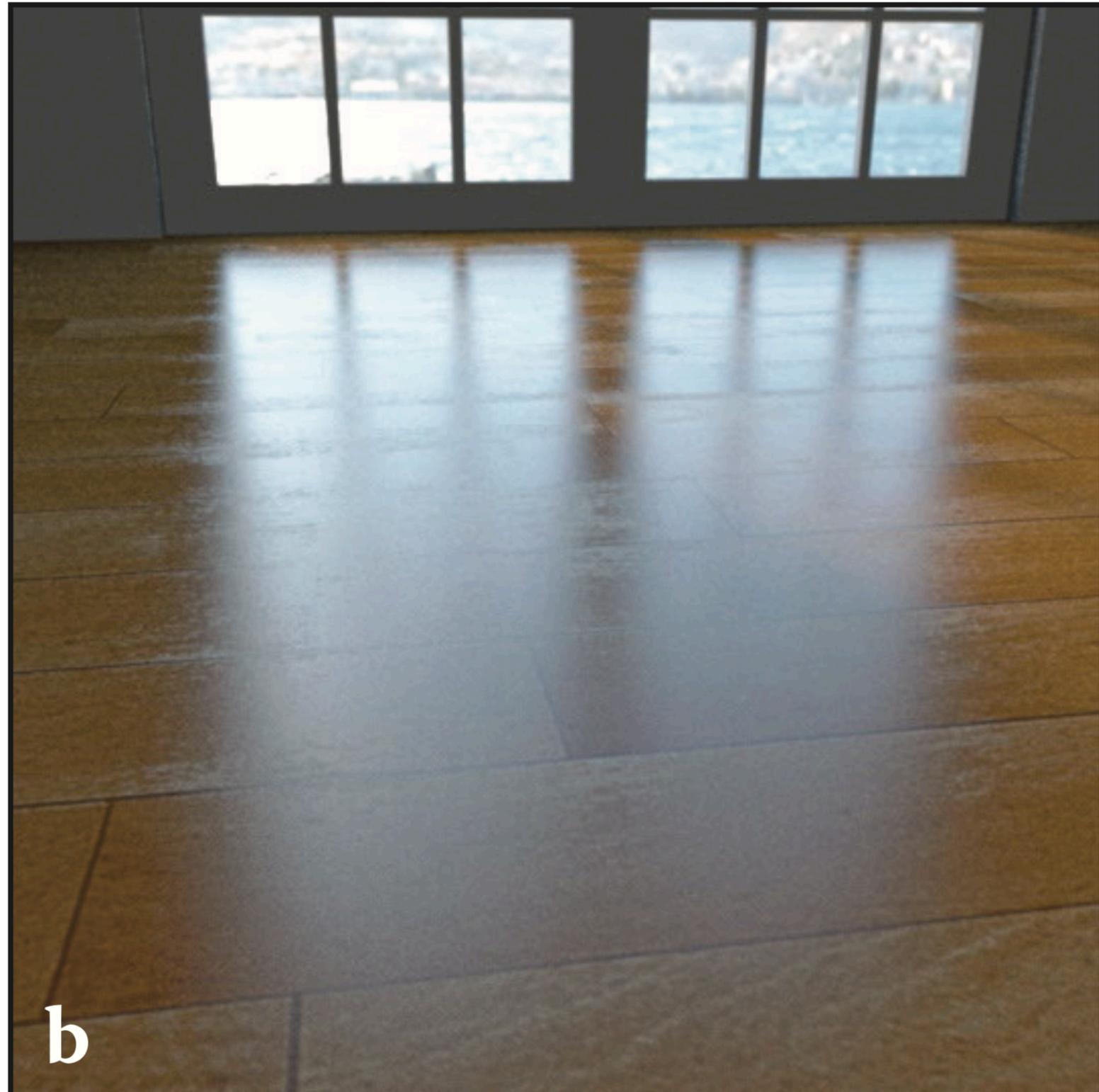
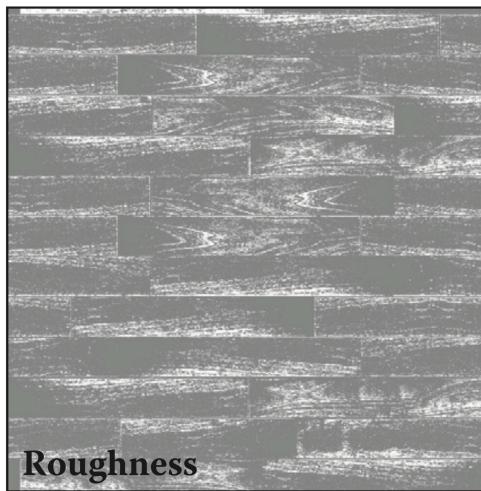
When we perform ray-tracing/z-buffering to find the first object hits by a ray, the texture does not change the answer.

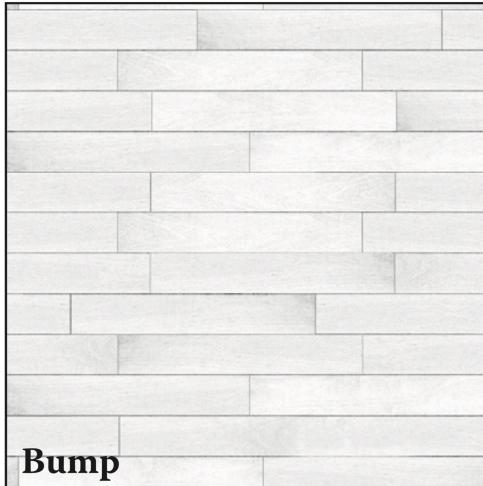
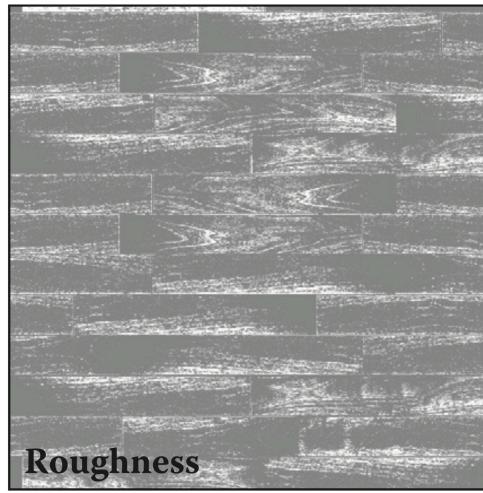
For example, we still treat a sphere as. a smooth normal sphere.

We use the bumps map only after computing

- 1) The first sphere hit, and
- 2) The hitting point (with some exceptions here)



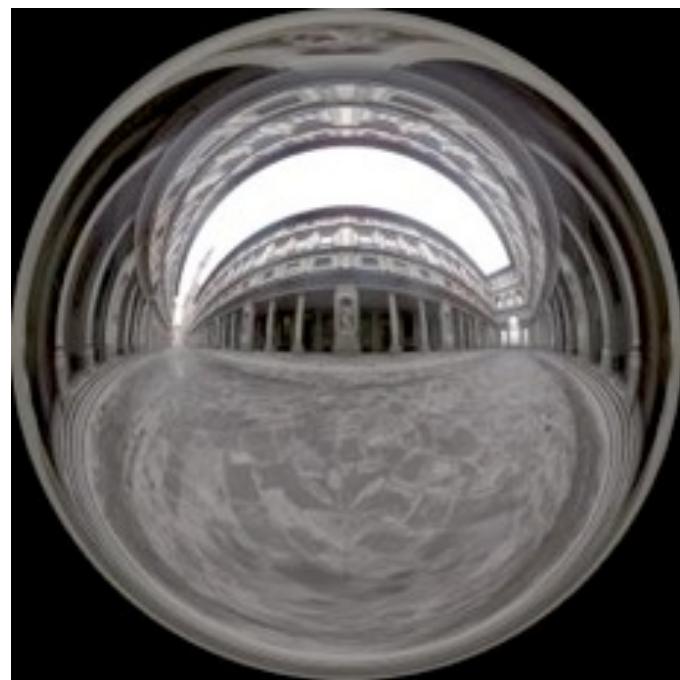




Environment Maps

- Use a texture to lookup values for rays that don't hit any objects

```
function trace_ray(ray, scene) {  
    if (surface = scene.intersect(ray)) {  
        return surface.shade(ray);  
    } else {  
        u,v = spheremap_coords(ray.direction);  
        return texture_lookup(scene.env_map, u, v);  
    }  
}
```

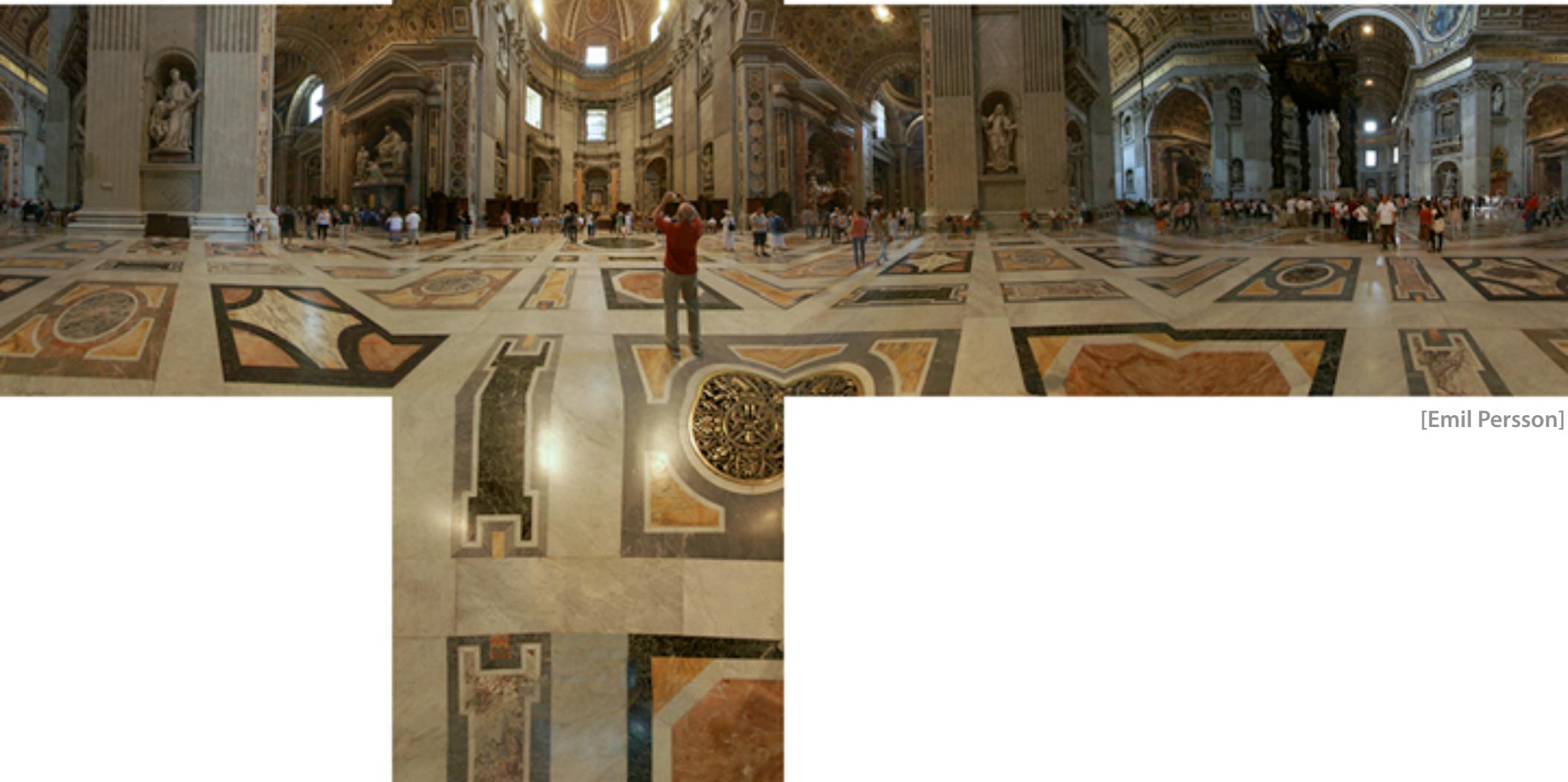


[Paul Debevec]





**Also can be done with
Cubemaps**



Relief mapping

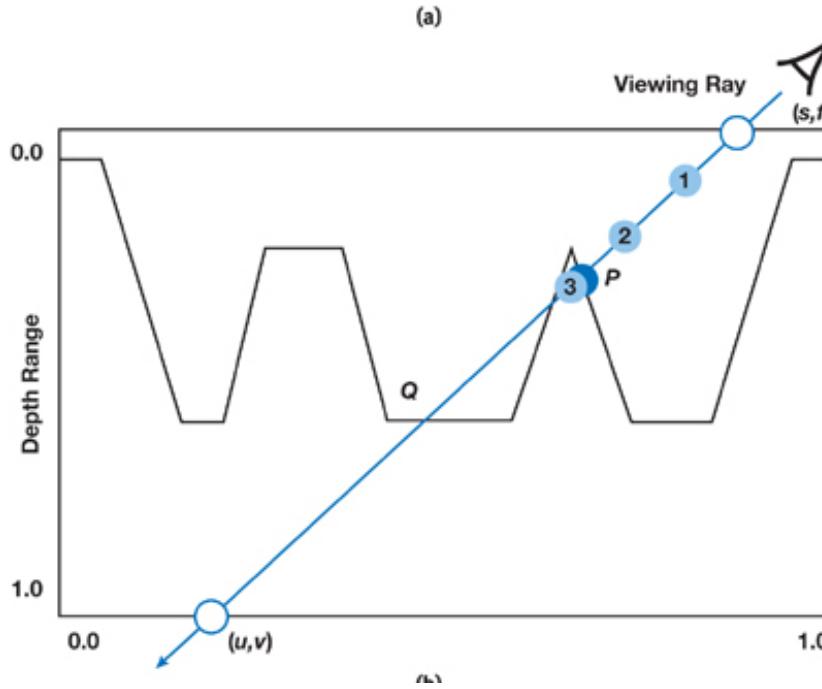


normal mapping



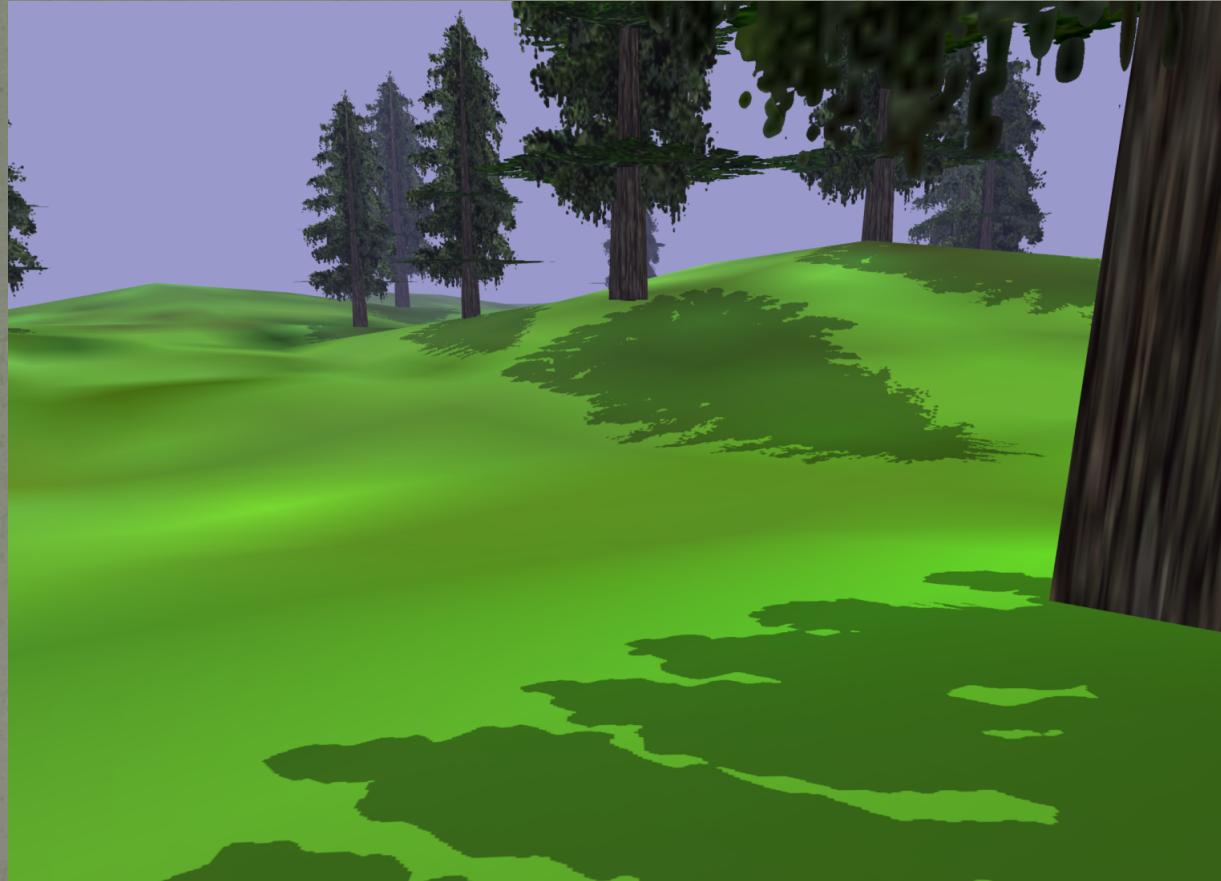
relief mapping

Relief mapping



- Trace the eye ray into the bump map. A simple implementation can rasterize the projection of the ray onto the tangent plane, stepping along (e_t, e_b) and adjusting the height by a factor proportional to e_h .

Shadow Mapping



Shadow Mapping (Concept)

- Render image from POV of light to create shadow map
- (ie. what would the scene look like if rendered from the POV of the light?) Light's view matrix = gluLookAt? glOrtho?
- When rendering a pixel, deciding if it is in shadow?
 - Project into light clip space
 - Compare z values (ie. distance from light source)
 - Distance from light > Z value of rendered texel in shadow map => occluding object => shadow!

Rendering **Large** Environments
(in real time)

Siddhartha Chaudhuri

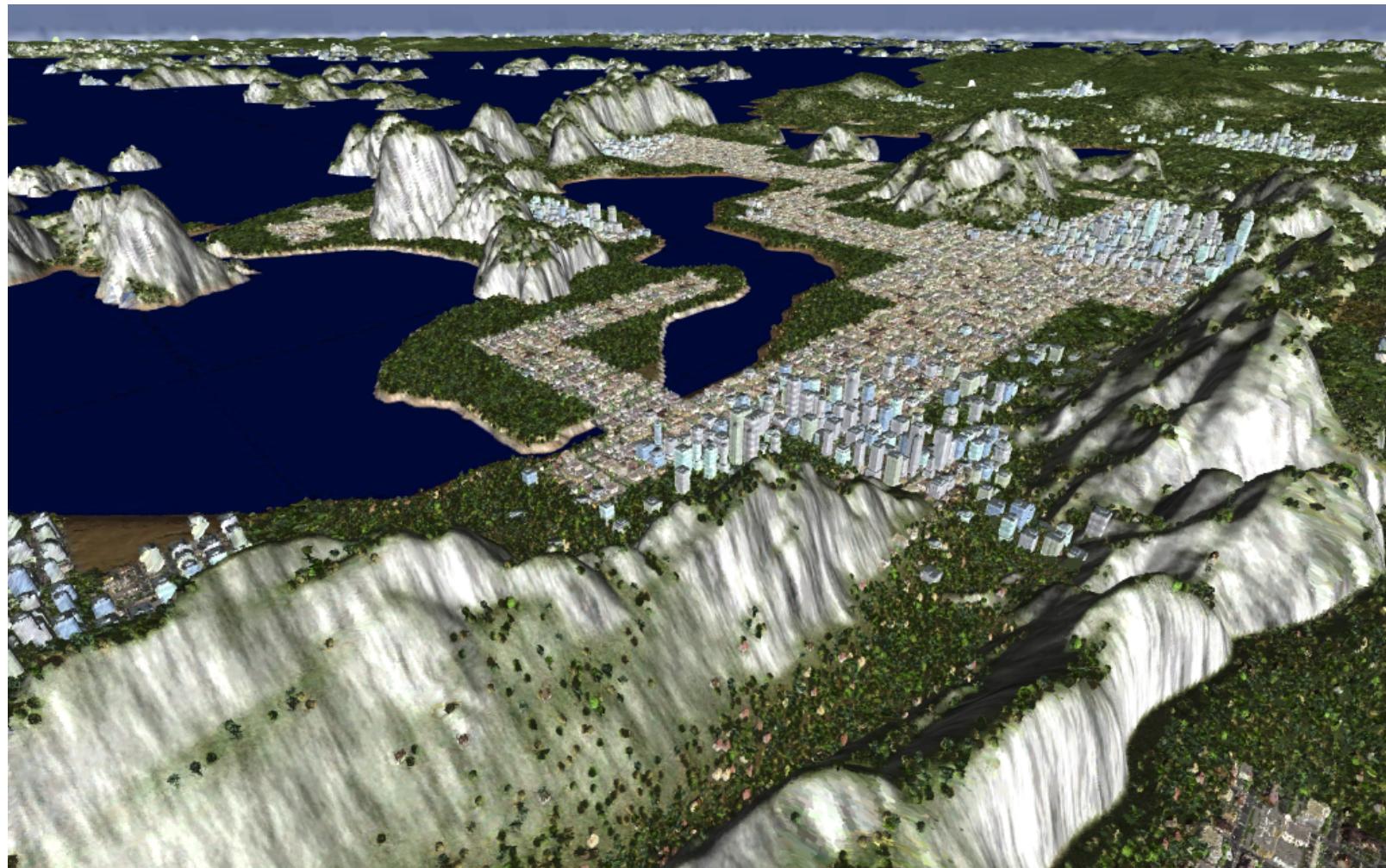
~500,000 polygons



~3,000,000 polygons



~25,000,000,000 polygons



Nvidia 2070

\$500

24M shaded polygons in 40 fps

Approximately. Depends on various parameters

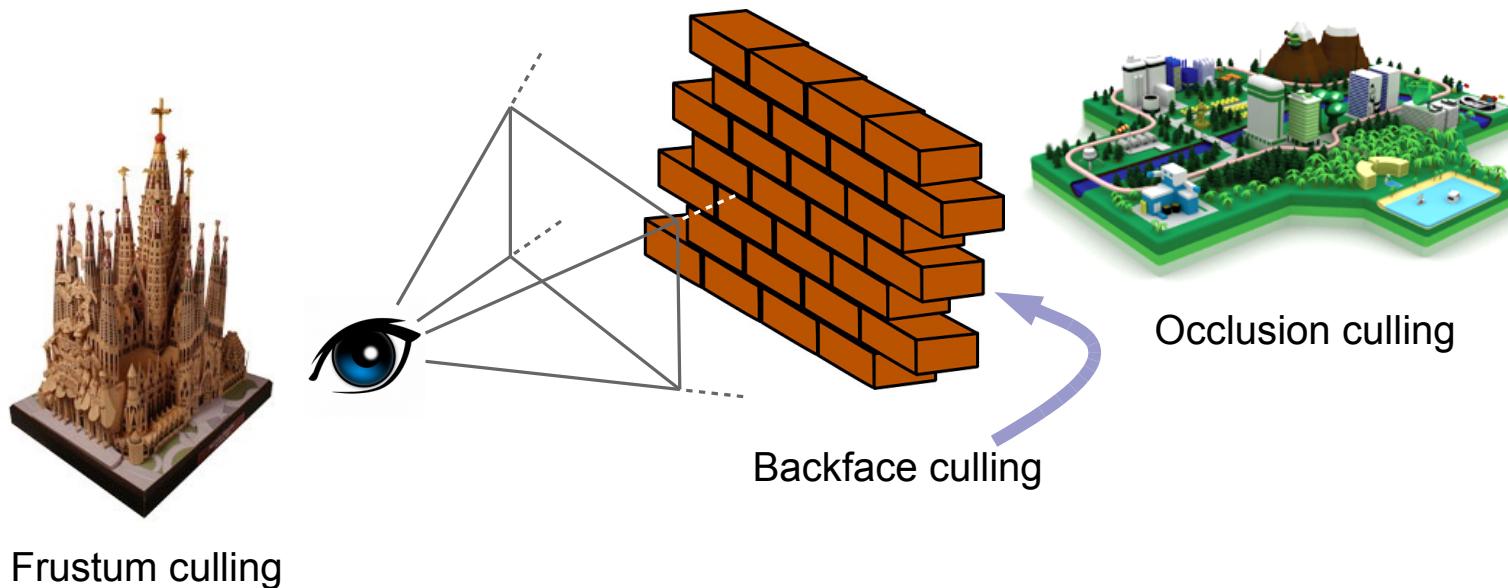


Largescale Rendering Cheat Sheet

- Don't render what you can't see
- Don't render what the display can't resolve
- People won't notice small errors, especially in background objects
- If all else fails, fog is your best friend :)

Don't render what you can't see

- Rasterizing invisible objects is wasteful
- Detect such objects early and ignore (**cull**) them



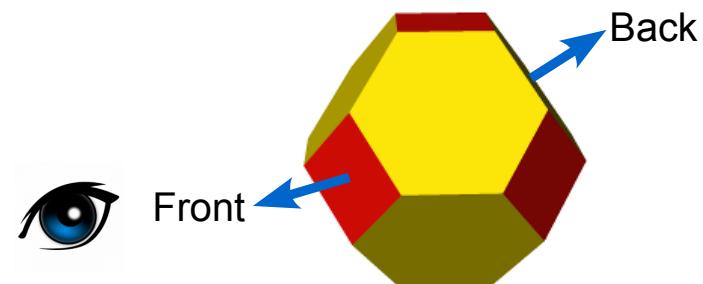
Difficulty

Backface < Frustum <<< Occlusion

Backface Culling

- Drop faces on the far side of object meshes
 - Assume face normals consistently point **inside-out**
 - Back faces have **normals pointing away** from the camera
- OpenGL:

```
glEnable(GL_CULL_FACE);  
glCullFace(GL_BACK);
```

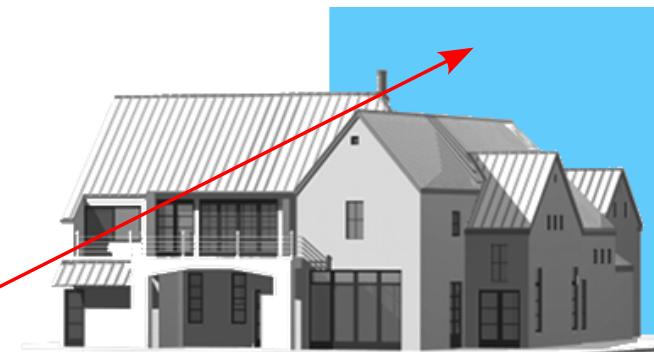
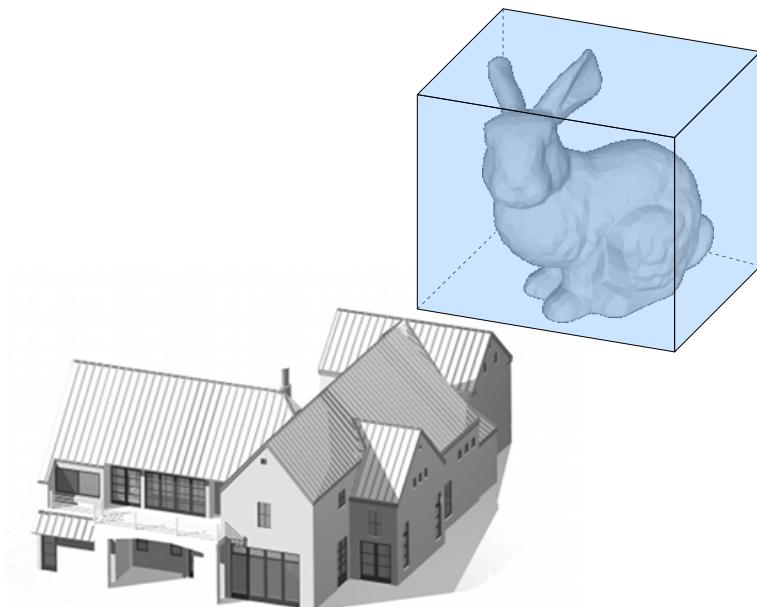


Frustum Culling

- Test each object against the view frustum
 - Much faster: **test the bounding box** instead
 - If object is visible, no frustum plane can have all 8 corners on invisible side
- Optimization:
 - **Group objects hierarchically**
 - *Octree* (or *quadtree* for 2.5D scenes)
 - *Binary Space Partitioning (BSP) tree* (or a restricted version called a *kd-tree*)
 - *Bounding box/sphere hierarchy*
 - Traverse tree top-down and ignore subtrees whose roots fail the bounding box test

Hardware Occlusion Queries

- Part of OpenGL/D3D API
- At any time, pretend to draw a dummy shape (say the bounding box of a complex object) and check if any pixels are affected
- Accelerated by hierarchical z-buffer
- Works for dynamic scenes



Unoccluded pixels of bounding box,
so object is potentially visible

From-Region Visibility

- Preprocessing:
 - Break scene up into regions
 - For each region, compute a *potentially visible set (PVS)* of objects
- Runtime:
 - Detect the region containing the observer
 - Render the objects in the corresponding PVS
- PVS is usually quite conservative, so further culling is needed

Portal-Based Rendering

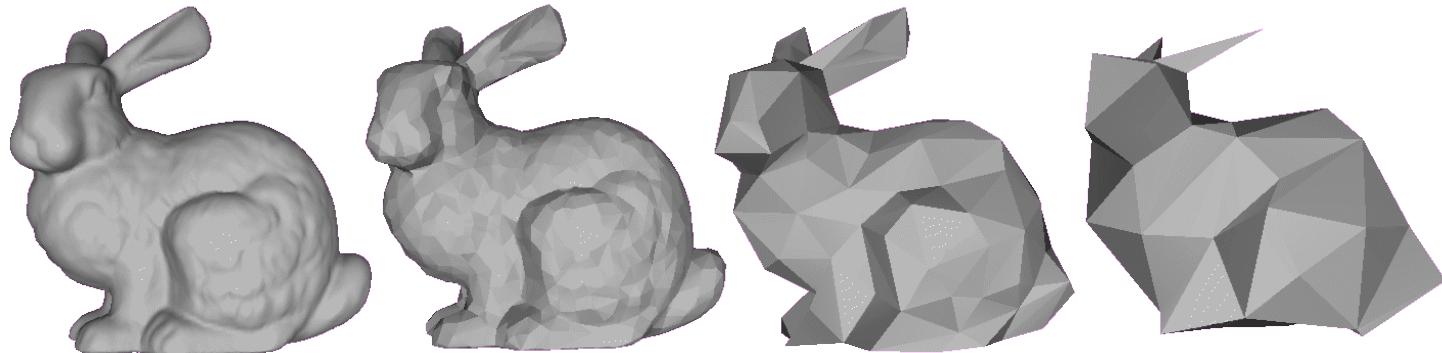
- Suitable for indoor environments
- Divide environment into **cells**, connected by simple polygonal **portals** (doors/windows/...)
- Render:
 - Neighboring cells with visible portals (check if projected polygon is within screen limits)
 - Neighbors-of-neighbors with portals visible through the first set of portals
 - ... and so on
- Further culling possible with frusta through portals

Guiding Principle

For every object, choose the **simplest possible representation** that will look nearly the same as the original *when rendered at the current distance*

Levels of Detail (LOD)

- **Coarser representations for distant objects**
 - Hierarchy of representations of the same object at different resolutions
 - The same idea can also be used for textures (*mipmapping*)



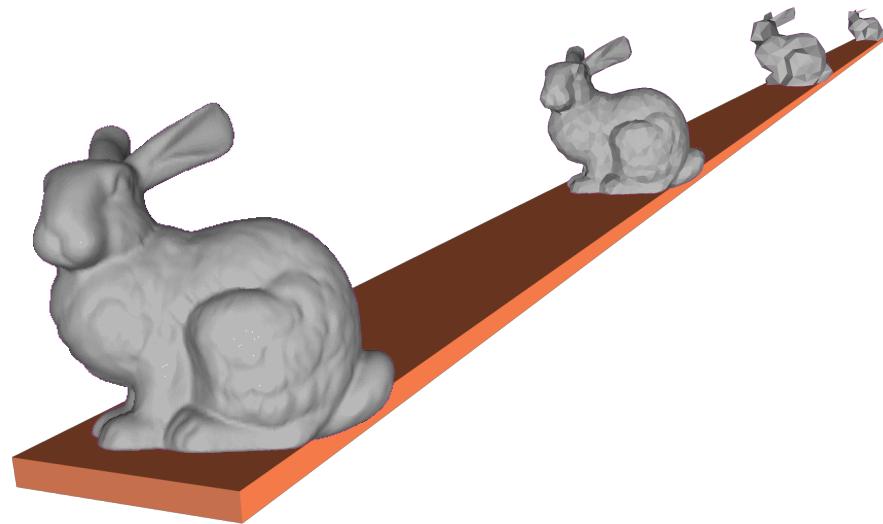
69,451 polys

2,502 polys

251 polys

76 polys

Levels of Detail (LOD)



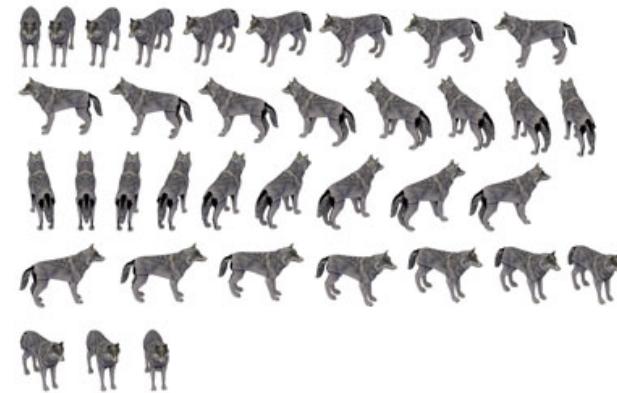
Environment Maps

- Very distant stuff looks the same from anywhere within reasonable limits
- Pre-render distant objects (including the sky) out to a 360° image
- Texture-map it onto a bounding cube at runtime



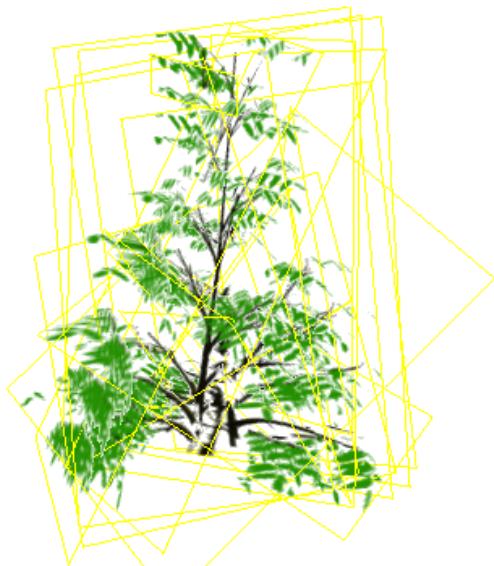
Image-Based Rendering

- Render complex objects to images and texture-map them to simple proxy shapes (**impostors**)
 - Environment mapping is a specific example
- **Billboards/sprites:** Textured quads always facing the viewer
 - Single image is valid if viewer doesn't move much

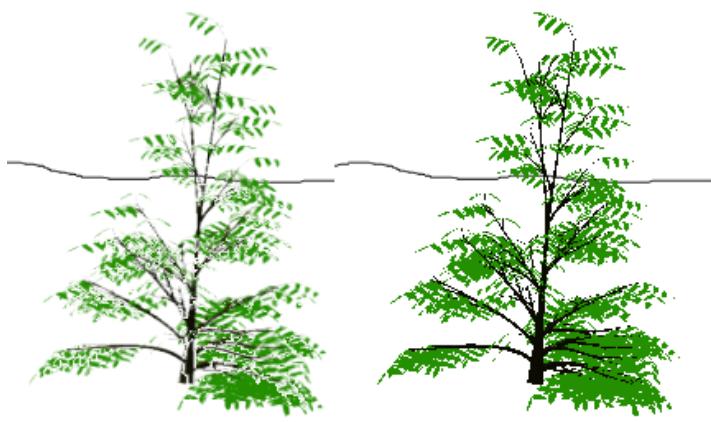


Problem: If we place the image of the wolf onto a rectangular billboard, how could we see the grass below the wolf ?

Image-Based Rendering



Tree decomposed into a cloud of
texture-mapped planar slices



Impostors

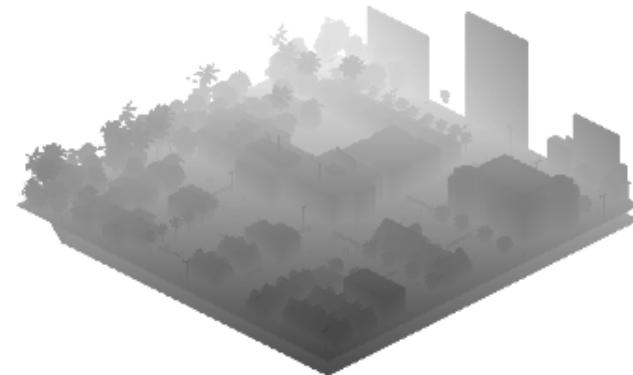
Original

Décoret, Sillion, Durand and Dorsey 2002

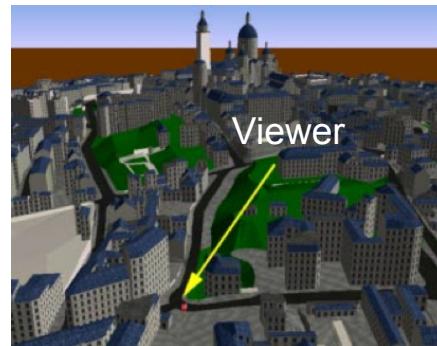
Adding Depth to Images



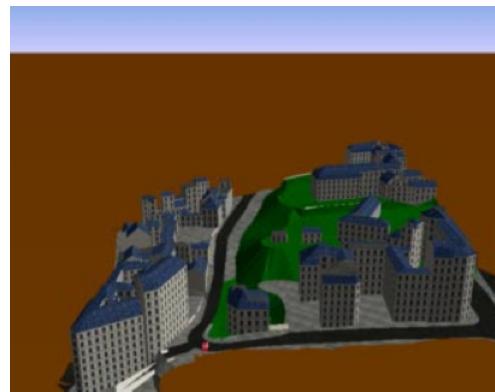
- Store the **depth map** as well as the color
- Impostor is **heightfield** defined by the depth map
- Fixes parallax errors (impostor is still valid when viewing position changes significantly)
- What are the drawbacks?



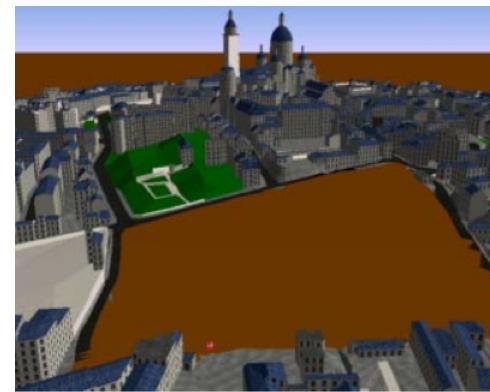
Images + Geometry



=



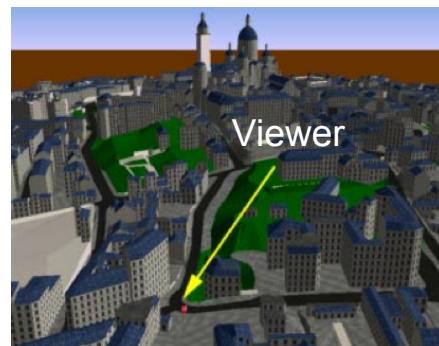
Foreground



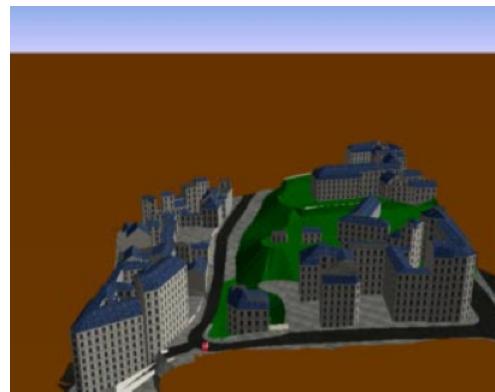
+

Background

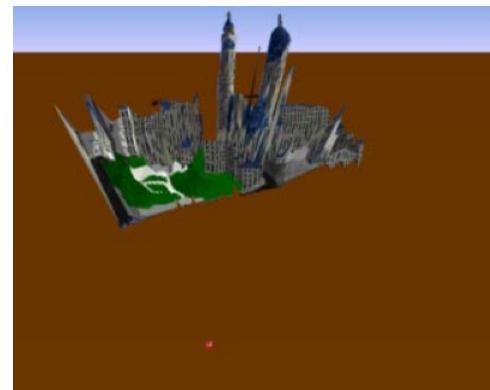
Images + Geometry



=



Foreground



Background Impostor

Images + Geometry (Rendered View)



=



Foreground



Background Impostor

Case Study: Quake

- Preprocessing:
 - Level map preprocessed into BSP-tree
 - Each leaf node stores potentially visible polygons from that region
- Runtime:
 - Leaf node containing player detected by searching the tree (very fast)
 - PVS of polygons for this node are rendered
 - (BSP-tree is NOT used for back-to-front rendering!)



Case Study: Quake

- Preprocessing:
 - Level map preprocessed into BSP-tree
 - Each leaf node stores potentially visible polygons from that region
- Runtime:
 - Leaf node containing player detected by searching the tree (very fast)
 - PVS of polygons for this node are rendered
 - (BSP-tree is NOT used for back-to-front rendering!)

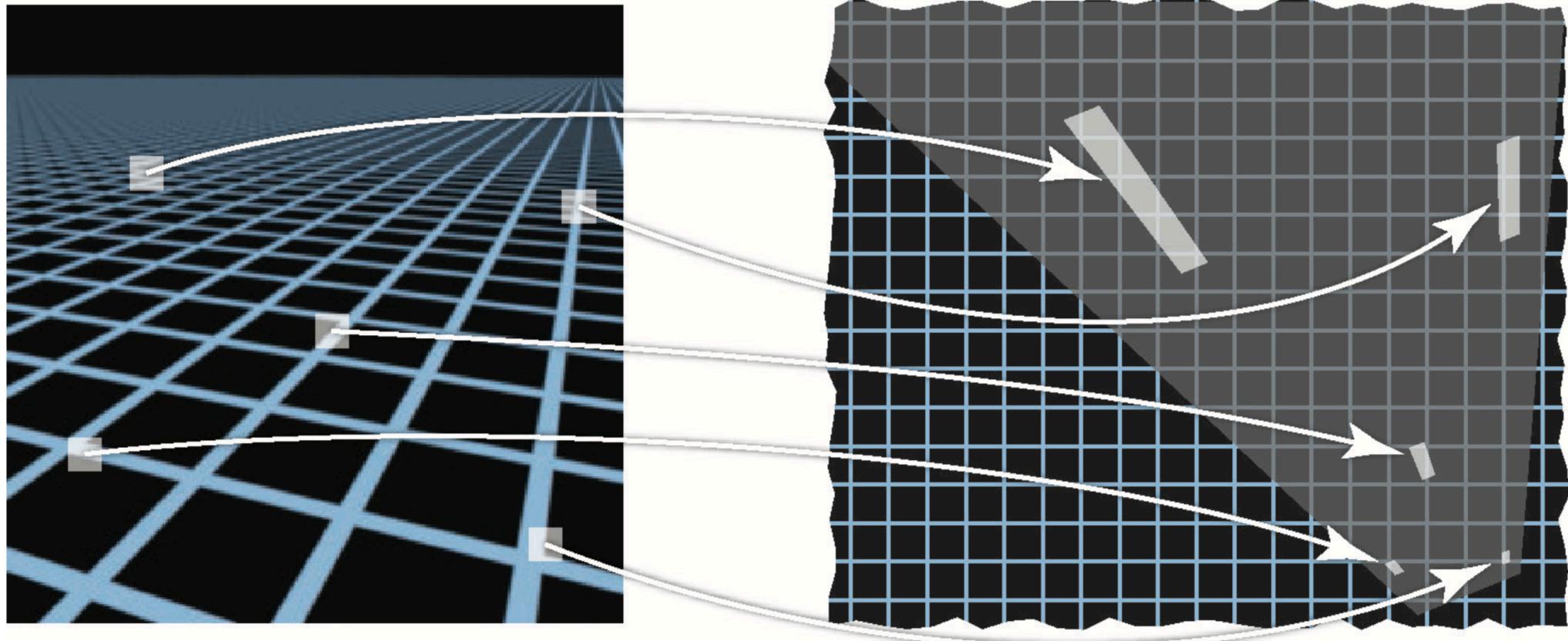
Antialiasing and Mipmaps

Problem: Sampling Textures Can Lead to Aliasing

- Just as we've seen with image processing and raytracing applications, if details are not captured with sufficient samples we can see noticeable artifacts
- Solution: use a better sampling/reconstruction

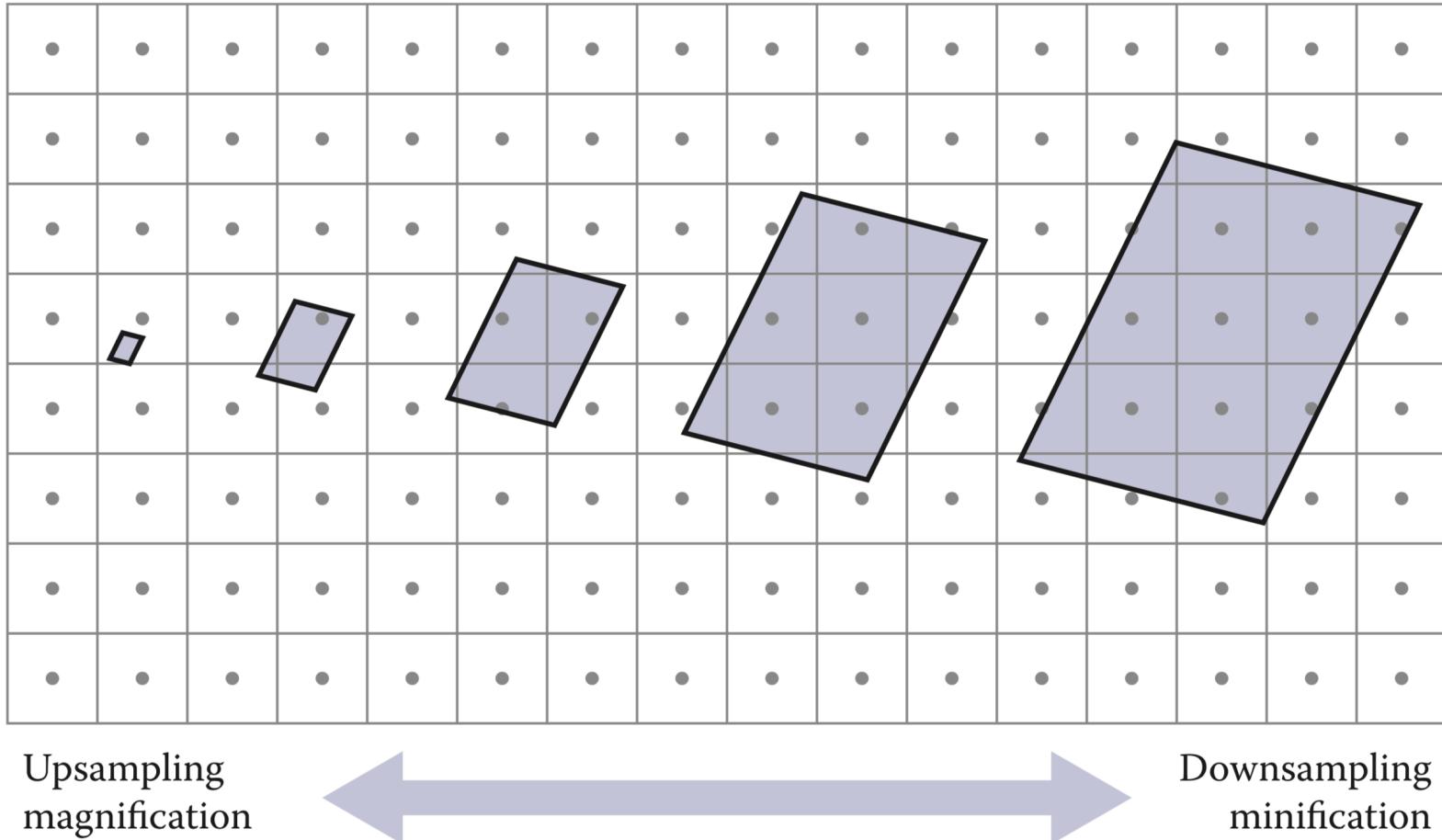
Pixel Footprints

- Can vary in size, shape, and orientation relative to the texture



Sampling and Reconstruction

- If footprint is small, need better reconstruction (e.g. bilinear instead of nearest neighbor)
- If the footprint is large, need to average many samples



Mipmaps

- To quickly compute averages, store the texture at multiple resolutions
- For each lookup, estimate the size of the footprint and index into the mipmap accordingly



<https://en.wikipedia.org/wiki/Mipmap>

Correcting Aliasing

