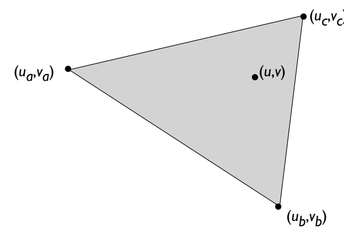
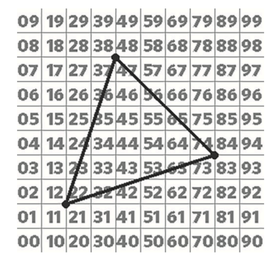
CSC 433/533 Computer Graphics

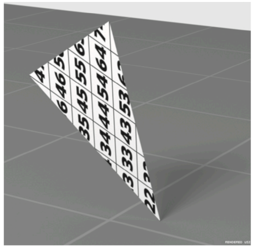
Alon Efrat
Credit: Joshua Levine

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?

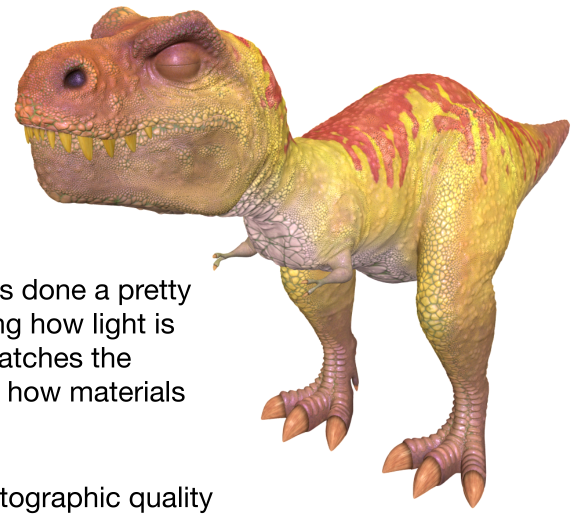






Textures

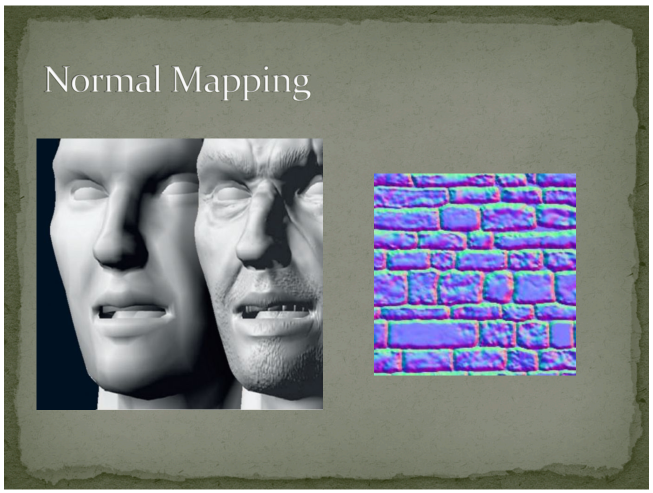
Challenge: Real World Surfaces Have Complex Materials



- 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?

http://ptex.us/ptexpaper.html

Texture 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

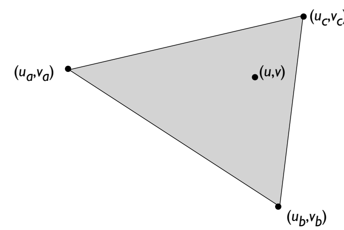
- Idea: model this variation using an image, called a **texture map** (or, sometimes "texture image" or just "texture")
- The texture map stores the surface details
  - Typically, shading parameters like k_d and k_s
- Can be used in lots of interesting ways to achieve complex effects

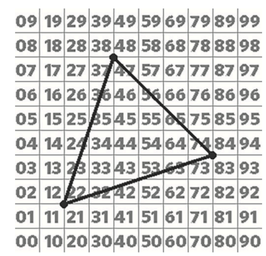
Texture Lookups

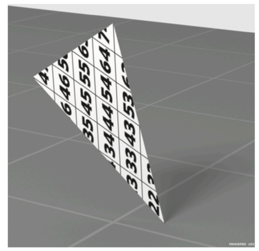
- Since the texture is an image, we need a way to index into it given a surface position
    - Or, where on the surface does the image go?
- Given a position, we lookup the **texture coordinates**, given as (u,v) values that refer to positions in the image
    - Easy to define for some shapes, can be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

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!







Computing Texture Coordinates

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

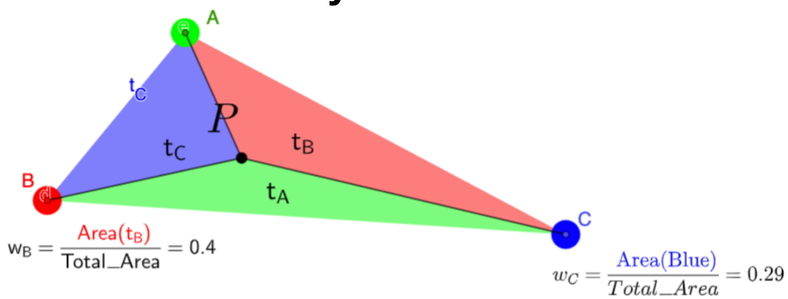
$$\phi: S \to T, \text{ for all } (x,y,z) \in S \text{ and } (u,v) \in T$$

- When shading a point (x,y,z), we compute ϕ(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

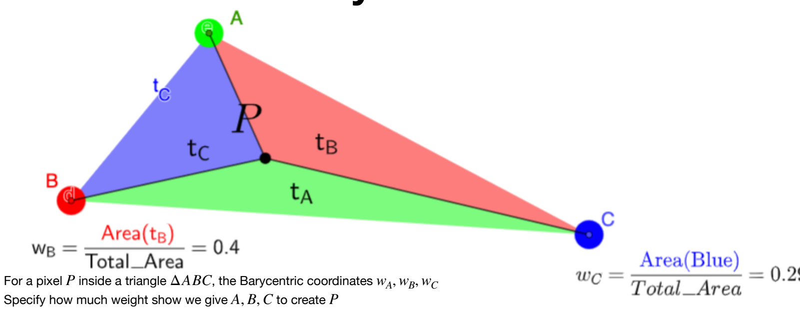
```
function texture_lookup(tex, u, v) {
  let i = Math.round(u * tex.width() - 0.5);
  let j = Math.round(v * tex.height() - 0.5);
  return tex.get_pixel(i,j);
}

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
}
```

Reminder: Barycentric coordinates



- For a pixel P inside a triangle ΔABC, the Barycentric coordinates w_A, w_B, w_C
- Specify how much weight show we give A, B, C to create P
- Specifically $P = w_A \cdot A + w_B \cdot B + w_A \cdot C$
- If A,B,C specifies locations, then P is on the triangle they defines
- If A,B,C are colors, then the same linear combination specifies how to interpolates the colors.
- $w_A = \frac{Area(\Delta CBP)}{Area(\Delta ABC)}$ - note = the triangle of A is the triangle that does NOT include A.

Reminder: Barycentric coordinates



- For a pixel P inside a triangle ΔABC, the Barycentric coordinates w_A, w_B, w_C
- Specify how much weight show we give A, B, C to create P
- Specifically $P = w_A \cdot A + w_B \cdot B + w_A \cdot C$
- If A,B,C specifies locations, then P is on the triangle they defines
- If A,B,C are colors, then the same linear combination specifies how to interpolates the colors.
- $w_A = \frac{Area(\Delta CBP)}{Area(\Delta ABC)}$ - note = the triangle of A is the triangle that does NOT include A.

$$P = w_A \cdot A + w_B \cdot B + w_A \cdot C = \,=$$

$$1 \cdot A + (w_B - w_A)(A - B) + (w_C - w_A)$$

How to map one triangle to another

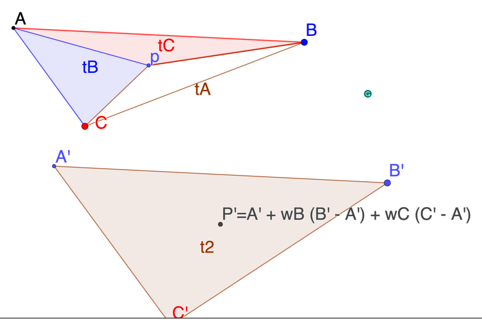
- Use barycentric coordinates. Recall - each point in image the texture triangle has coordinates

$\phi(p) = (w_B, w_C)$ where $w_B = \dfrac{area(t_B)}{area(\Delta ABC)}$ and $w_B = \dfrac{area(t_B)}{area(\Delta ABC)}$

$w_B = \dfrac{\text{area}(tB)}{\text{area}(\Delta ABC)} = 0.39$

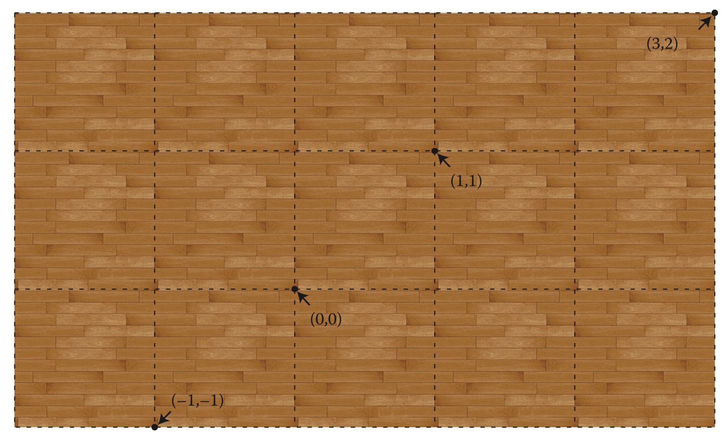
$p = A + w_A(B - A) + w_C(C - A) =$
$A + 0.39\ (B - A) + 0.33(C - A)$

We map $A \rightarrow A'$, $B \rightarrow B'$, $C \rightarrow C'$



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

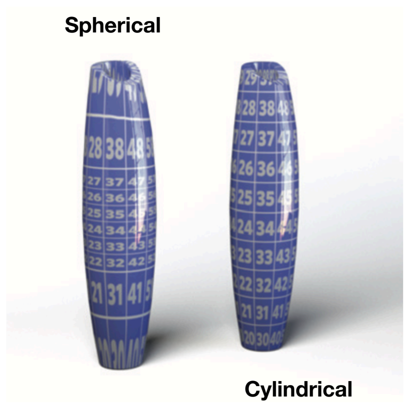


Cylindrical Projection

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

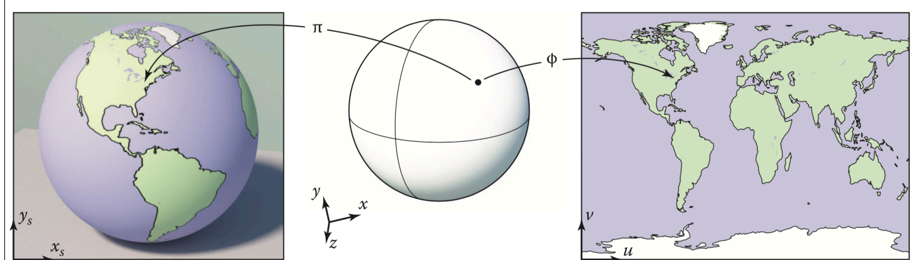
  u = (π + atan2(y,x)) / (2π)

  v = height=z

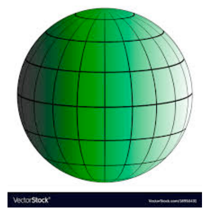


Three Spaces

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



Spherical Projection

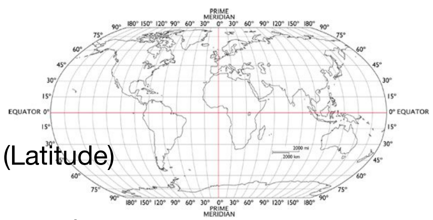


- Convert (x,y,z) to spherical coordinates, discard radius

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

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

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

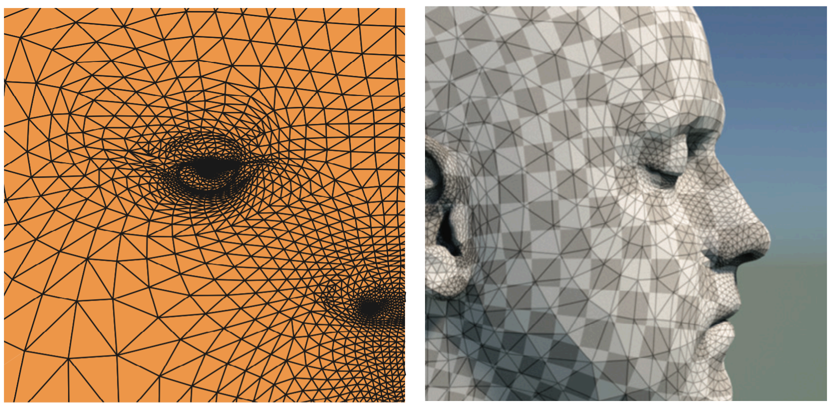


- Similar to casting a ray outward from center

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

Shape vs. Area Distortions



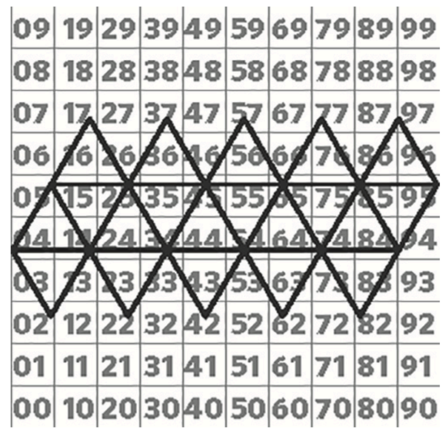
Low shape distortion,
Moderate area distortion

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 cuts ? ϕ should have as few discontinuities as possible

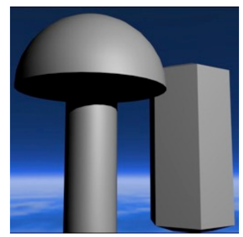
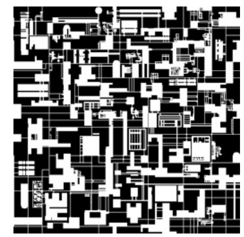
Distortions vs. Discontinuities

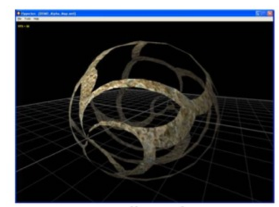


No distortion to area,
Many discontinuities

Applications of Textures

Opacity mapping

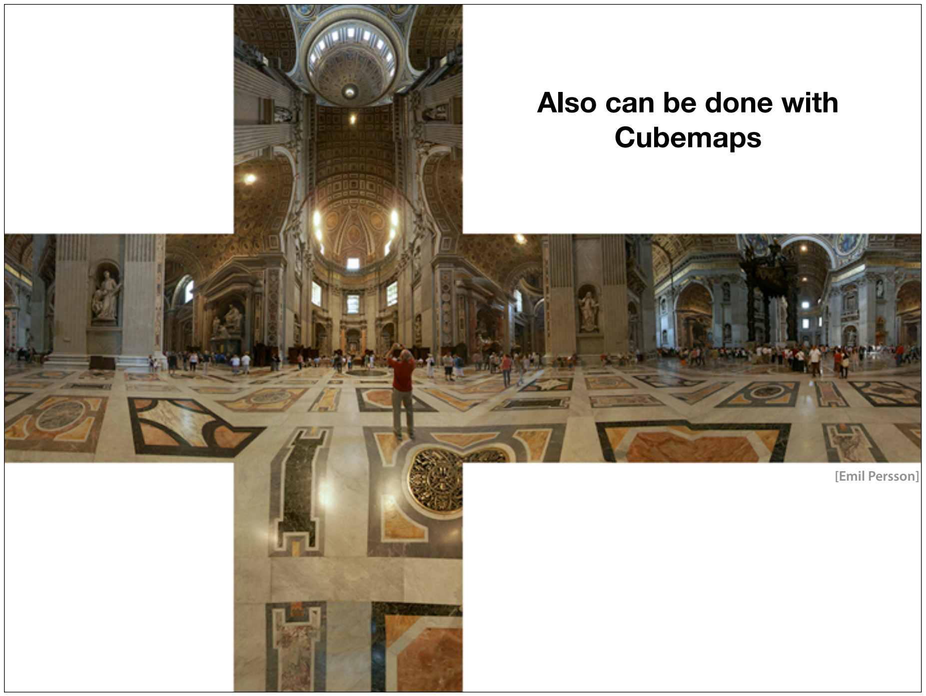
http://lectrablog.lectralizard.com/page/2

Controlling Shading Parameters

- Can look up diffuse and specular coefficients k_d and k_s, or both

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



Also can be done with Cubemaps

[Emil Persson]

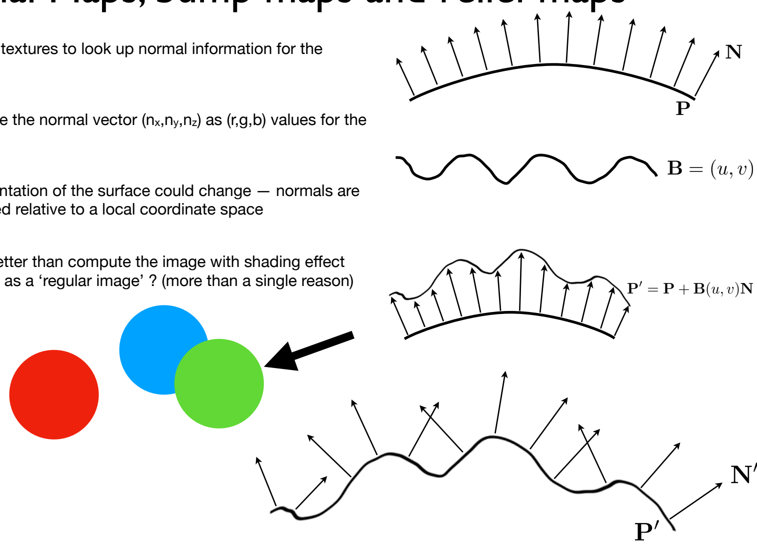
Environment Maps

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

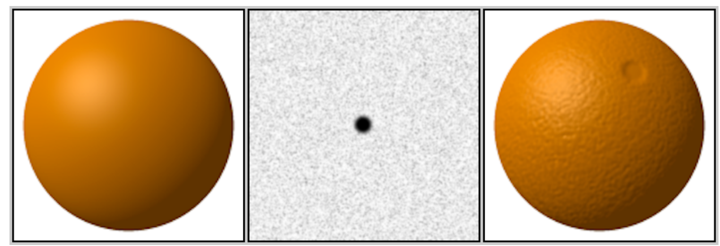
```
function trace_ray(ray, scene) {
  if (surface = scene.intersect(ray)) {
    return surface.shade(ray);
  } else {
    u,v = spheremap_coords(ray.direction);
    return texture_lookup(scene.env_map, u, v);
}
```

Normal Maps, bump maps and relief 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
- Why is this better than compute the image with shading effect (once) and us as a 'regular image' ? (more than a single reason)



Bump mapping

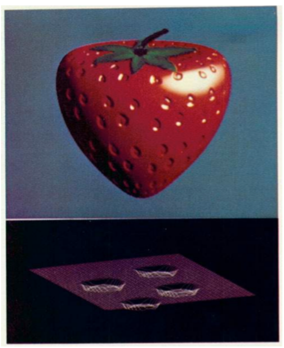


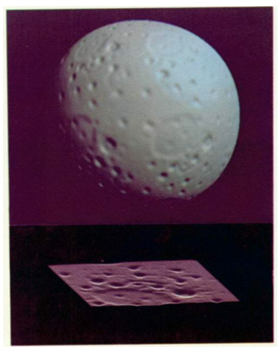
- Simulates roughness ("bumpiness") of a surface without adding geometry
- Uses a two-dimensional height field (bump map) to perturb the normal during per-fragment shading calculations
- Typically, store the normal vector (n_x,n_y,n_z) as (r,g,b) values for the pixel
- Limitations: the mapping of texture onto the surface is unaffected; silhouette is also unaffected.
    - Why is this different (and more efficient) than storing the 3D mesh ?

1. For hidden surfaces removal, the orange is still a (smooth) sphere
2. A patch of the surface appears multiple times via tiling, yet it looks different
3. Interactive settings

GDallimore (Wikimedia Commons)

Bump mapping





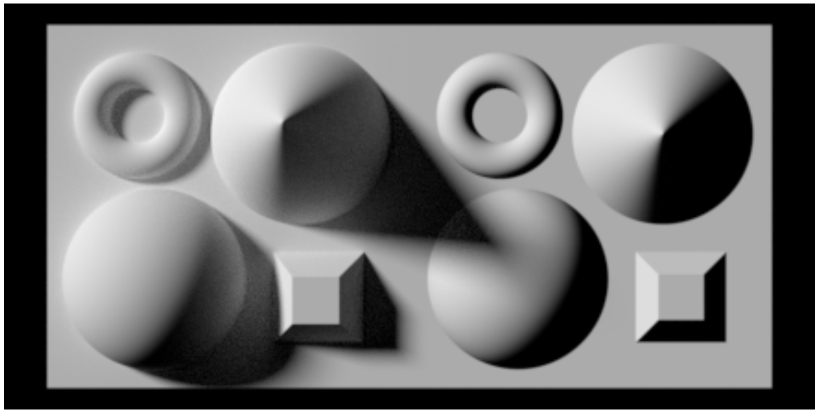
- Simulates roughness ("bumpiness") of a surface without adding geometry
- Uses a two-dimensional height field (bump map) to perturb the normal during per-fragment shading calculations
- Limitations: the mapping of texture onto the surface is unaffected; silhouette is also unaffected.

Blinn, SIGGRAPH 1978

More example of normals maps in interactive settings:
The image reacts to changes in lights directions

Just to make sure:
for depth computation (who occlude whom), we see one quads (two triangles)
For shading effects, we use the normals

Credit: wiki



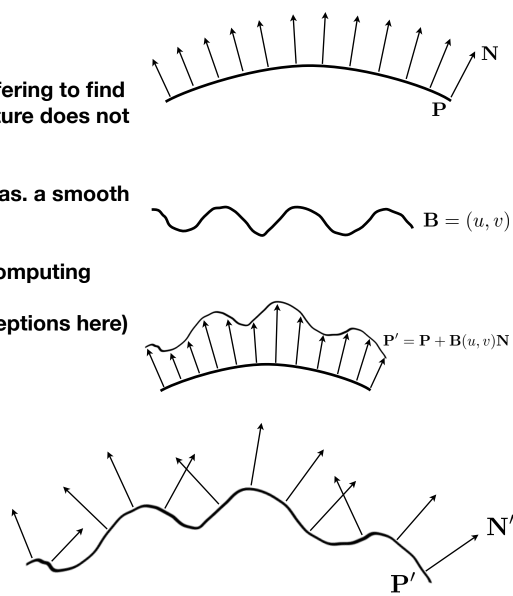
Important

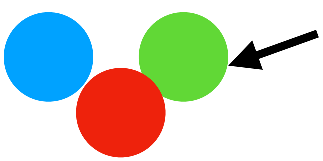
When we perform ray-tracing/z-buffering to find the first object hits by a ray, the texture does not change the answer.

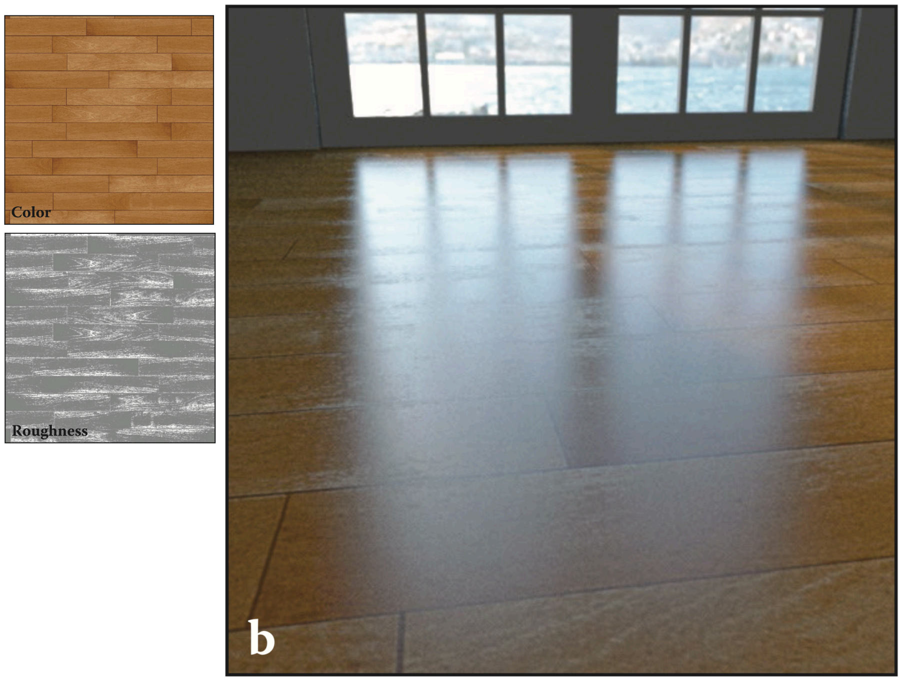
For example, we still treat a sphere as. a smooth normal sphere.

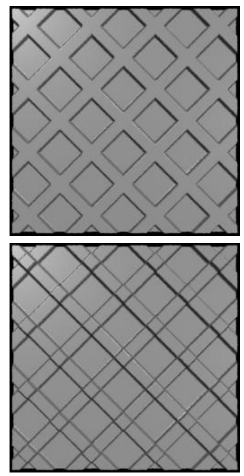
We use the bumps map only after computing

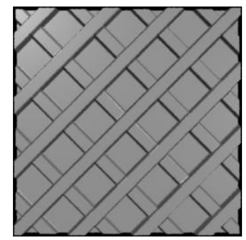
1) The first sphere hit, and
2) The hitting point (with some exceptions here)

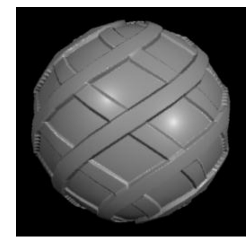






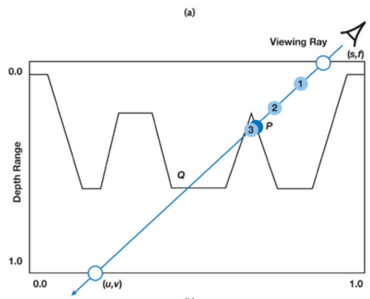






From RmanNotes
http://www.cgrg.ohiostate.edu/~smay/RManNotes/index.html

Relief mapping



- We are using depth but only comparing rays that hit the surface.
- Trace the eye ray into the bump map. A simple implementation find intersection of the ray with the tangent plane. (e.g. using z-buffer or simple ray tracing). Once this intersection points is found, step along the ray $e(v) = eye + t \cdot \vec{r}$ and simultaneously along the surface, till intersection is found (but only among the

Image from Policarpo and Oliveira (2008)

Relief mapping

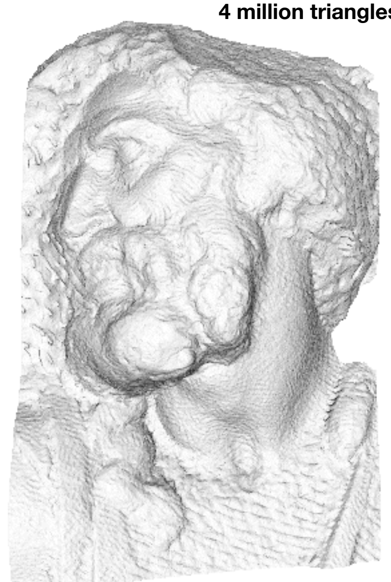
normal mapping

relief mapping

Image from Natalya Tatarchuk

Normal Map Example

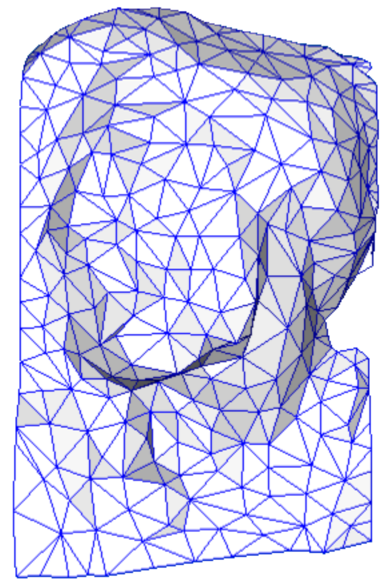
4 million triangles



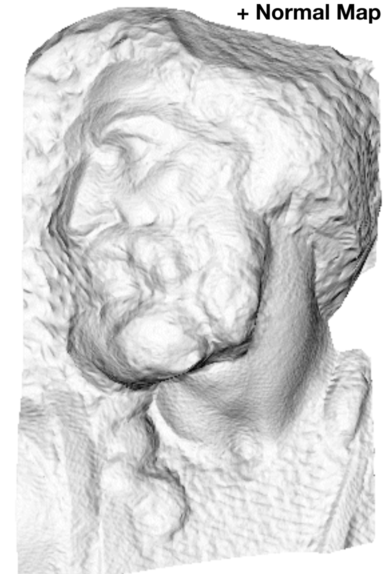
- Transfer details from high resolution mesh to normal map image

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

500 triangles



500 triangles
+ Normal Map



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

[Paul Debevec]

Relief mapping

normal mapping

relief mapping

Image from Natalya Tatarchuk

More Relief Mapping

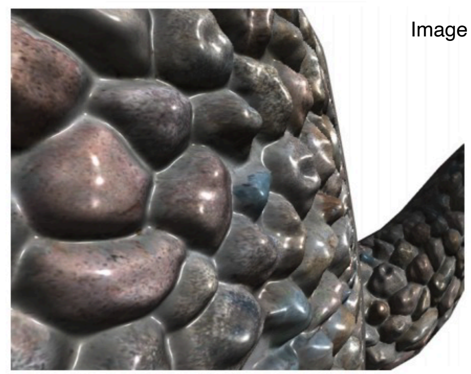
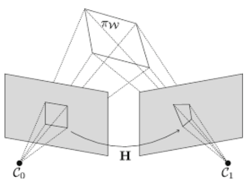


Image from Policarpo et al. (2005)

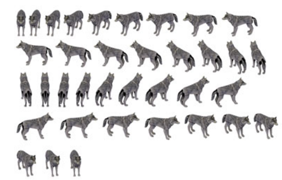
Homography



- (parallax) the transom of a planar shape, as seen 'from an angle).
- Lines → lines, but angles might change.
- Could be created by a simple matrix operation

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 the place the image of the wolf onto a rectangular billboard, how could we see the grass below the wolf ?
We cannot create one image of the wolf which is convincing from all angles, (e.g. looks but we could create a library of a small number of images, and decide which one to use based on viewer's movement

Issues of using only a single billboard

- small movement of the camera might cause self occlusion (e.g. one branch covers another. Of course, this change is not captured in a single billboard.
- Solution: Place several bboards called **slices**   Non-tree pixels are transparent.
- Each slice holds the portion of the tree which is close to its orthographic projections on the plane. In other words, no self occlusions in the slice, so it could be viewed from multiple angle

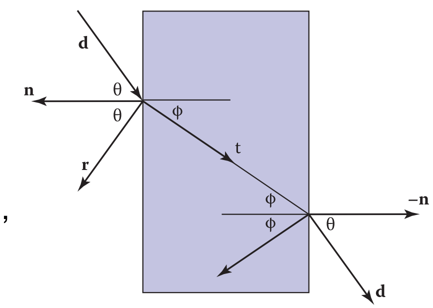
Tree decomposed into a cloud of texture-mapped planar slices

Impostors

Original

Snell's Law

- Governs the angle at which a refracted ray bends
- Computation based on refraction index of original medium, n_a, versus new index n_t (for example, $n_{air}=1$, $n_{glass}=1.7$
- $n_t \sin\theta = n_a \sin\phi$



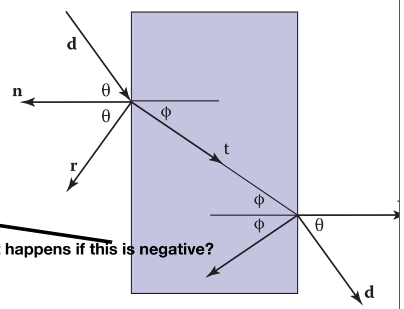
Snell's Law

- Working with cosine's are easier because we can use dot products
- Can derive the vector for the refraction direction **t** as

$$\mathbf{t} = \frac{n(\mathbf{d}+\mathbf{n}\cos\theta))}{n_t} - \mathbf{n}\cos\phi$$

$$= \frac{n(\mathbf{d}-\mathbf{n}(\mathbf{d}\cdot\mathbf{n}))}{n_t} - \mathbf{n}\sqrt{1-\frac{n^2\left(1-(\mathbf{d}\cdot\mathbf{n})^2\right)}{n_t^2}}$$

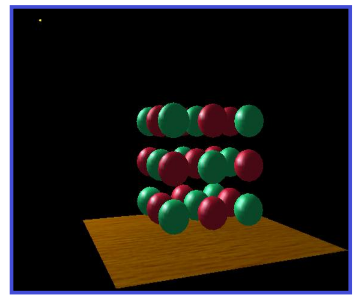
What happens if this is negative?

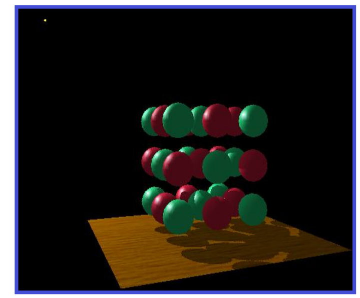


Notice that (unfortunately), the letter 'n' is used in this equation to denote three (3) different terms:

1. $\mathbf{n} = \vec{\mathbf{n}}$ is the vector which is normal to the sufrace, pointing toward the source of the ray.
2. $n = n_{from}$ is the medium coefficient in the medium the ray is coming from. For example, if the ray entering from air $\rightarrow water$, the n=0
3. $n_t = n_{to}$ is the medium coefficient in the medium the ray is coming from. For example, if the ray entering from air$air \rightarrow water$, then $n_t \approx 1.8$

Shadows

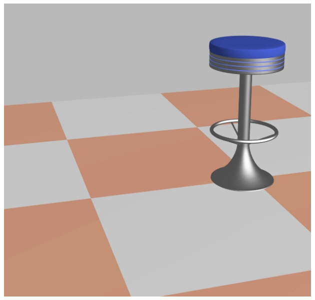


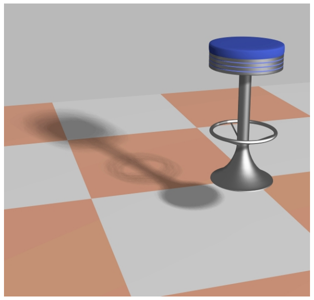


- Valuable cue of spatial relationships
- Increases realism

Mark J. Kilgard

Shadows

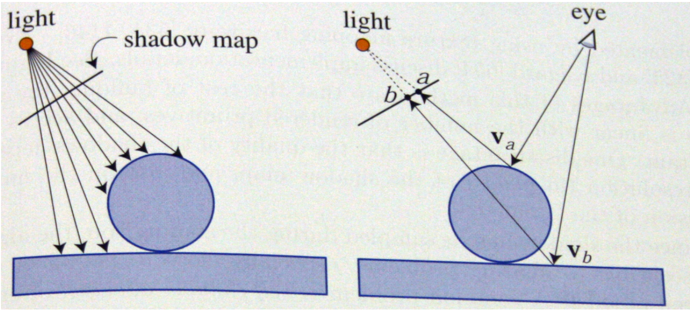




- Valuable cue of spatial relationships
- Increases realism

Akenine-Moeller and Haines

Shadow mapping



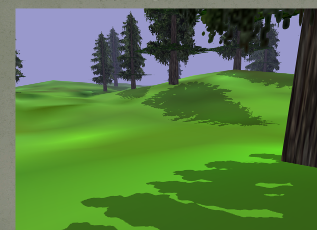
- First pass: render the scene from the viewpoint of the light, store depth buffer as texture (shadow map) much easier to do if the projected object is planar (triangle/quad)
- Second pass: project vertices into shadow map and compare depth values

Akenine-Moeller et al., Real-Time Rendering

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!

Shadow Mapping



Two possible tasks (different difficulties)

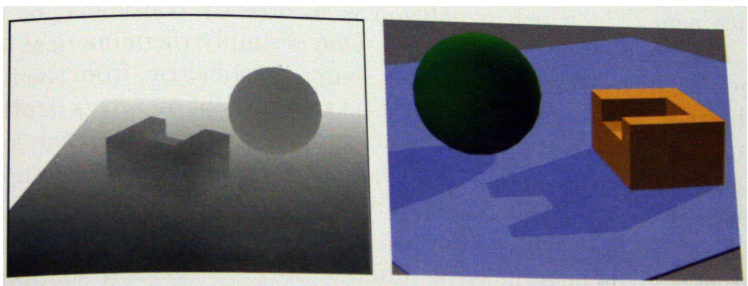
1) case shadow by changing the texture (lighting) of the image of the grass
   1) Does not effected by the location of viewer

   Used to determine if a pixel of the grass is exposed to the sun during the rendering of the grass (applying e.g. for Phong Specular Shading)

   Need to be able to determine FAST in a point of the grass is visible to the sun

   For this we use the depth buffer D[1..n, 1..n] from the first path as texture for other second pass

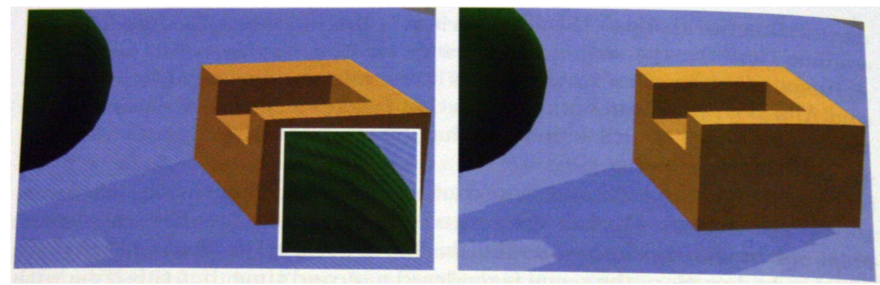
Shadow mapping



- First pass details: can disable all rendering features that do not affect depth map.
- Second pass details: For each fragment, use the light's modelview and projection transforms to obtains (u,v) coordinates in the shadow map and the depth w of the vertex.
- Compare w with value w' stored in (u,v) in the shadow map. If $w \leq w'$, perform lighting calculations with this light. Otherwise, do not.

Akenine-Moeller et al., Real-Time Rendering

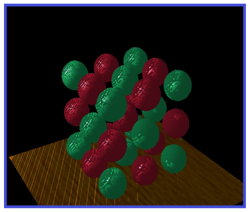
Bias



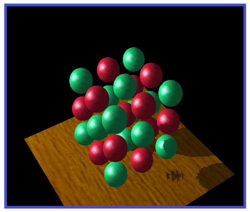
- Numerical imprecision leads to self-shadowing
- Solution: add a bias ε. Change comparison from $w \leq w'$ to $w \leq w' + \varepsilon$
- Can use glPolygonOffset

Akenine-Moeller et al., Real-Time Rendering

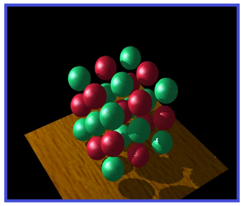
Setting the bias



Too little



Too much



Just right

- Numerical imprecision leads to self-shadowing
- Solution: add a bias ε. Change comparison from $w \leq w'$ to $w \leq w' + \varepsilon$
- Can use glPolygonOffset

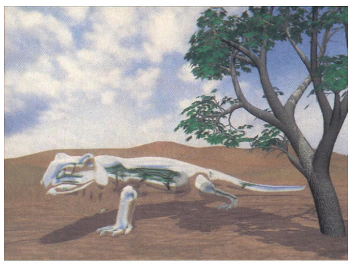
Mark J. Kilgard

Reflection mapping

- Render the scene from a single point inside the reflective object. Store rendered images as textures.
- Map textures onto object. Determine texture coordinates by reflecting view ray about the normal.

Terminator 2

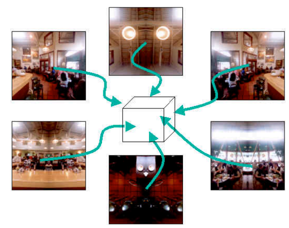
Cube mapping

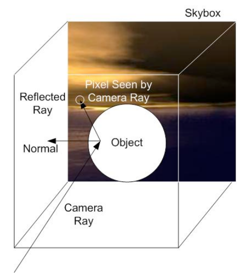


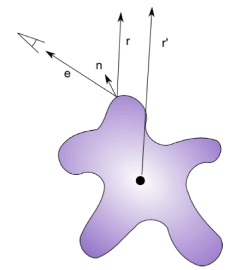
- Render the scene six times, through six faces of a cube, with 90-degree field-of-view for each image.
- Store images in six textures, which represent an omni-directional view of the environment

Greene, 1986

Cube mapping







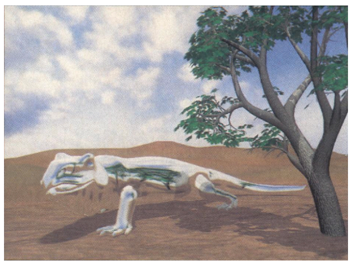
- To compute texture coordinates, reflect the view vector **v** about the normal **n**:

$$\mathbf{r} = 2(\mathbf{v} \cdot \mathbf{n})\mathbf{n} - \mathbf{v}$$

- The highest (in absolute value) coordinate of r identifies which of the six maps we need. The texture coordinates in this map are obtained by normalizing the other two coordinates of r.

http://developer.nvidia.com/object/cube_map_ogl_tutorial.html; TopherTG (Wikipedia)

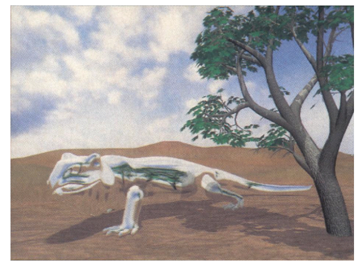
Sphere mapping



- Cube maps require maintaining six texture in memory
- Sphere mapping uses a single viewpoint-specific environment map, updated every frame
- Map depicts a perfectly reflective sphere viewed orthographically

Greene, 1986

Sphere mapping



- Cube maps require maintaining six texture in memory
- Sphere mapping uses a single viewpoint-specific environment map, updated every frame
- Map depicts a perfectly reflective sphere viewed orthographically

Greene, 1986

Rendering Large Environments

(in real time)

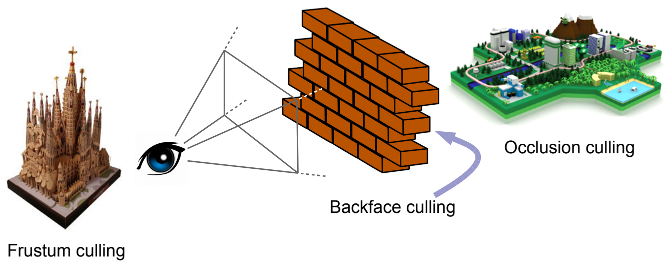
Siddhartha Chaudhuri

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



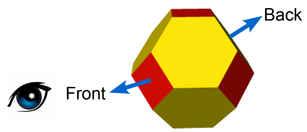
Difficulty

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);
```

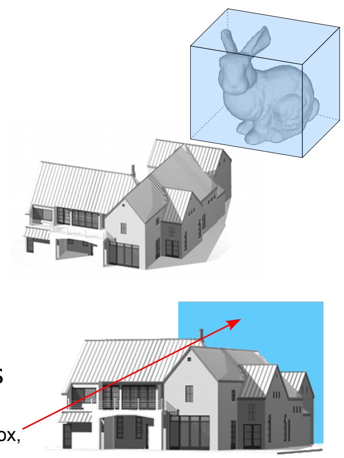


Frustum Culling

- 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

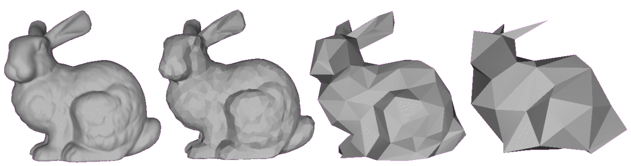
- Suitable for indoor environments
- Divide environment into ***cells***, connected by simple polygonal ***portals*** (doors/windows/...)
- Render:
  - Neighboring cells with visible portals (check if projected polygon is within screen limits)
  - Neighbors-of-neighbors with portals visible through the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

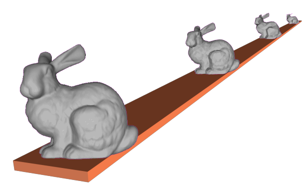
Levels of Detail (LOD)

- **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)



Instead of storing **only** the highly-detailed model, we store:

1. The highly-detailed model, $\approx 70K$ triangles, and
2. a less-detailed simplification of the same model model, and
3. a very coarse copy etc.

- The decision of which of these copies to use for rendering is made in real time, depending on viewer positions.

- What is the overhead of storing multiple copies?
- A very pessimistic upper bound.

1. Assume that the we start with a model with n triangles
2. In the coarser model, we store $\leq n/2$
3. in the coarser model, we store half this number, so $\leq n/4$,
4. next, $n/8$...

- So in total at most $n(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots) \leq 2n$, and the overhead is at most factor 2. Neglectable, comparing to the speedup in rendering time.

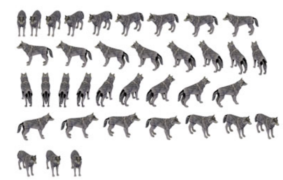
- In practice, the actually overhead is much smaller

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

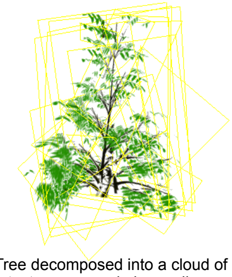
Image-Based Rendering

- 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



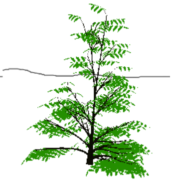
Problem: If the place the image of the wolf onto a rectangular billboard, how could we see the grass below the wolf ?

Image-Based Rendering



Tree decomposed into a cloud of texture-mapped planar slices

Impostors

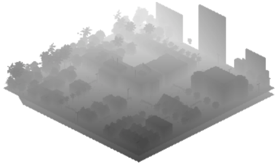


Original

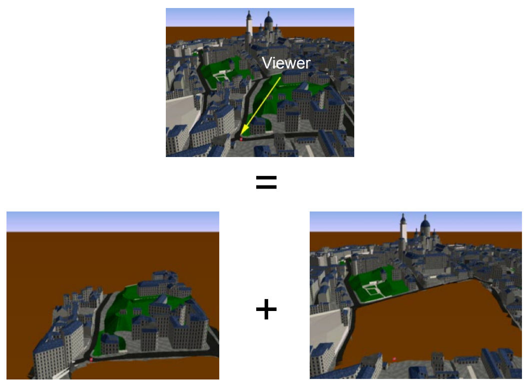
Décoret, Sillion, Durand and Dorsey 2002

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?



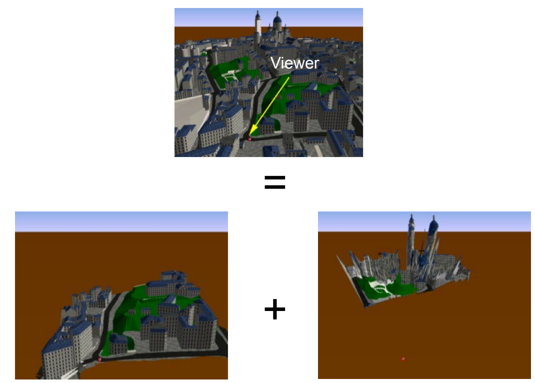
Images + Geometry



Foreground

Background

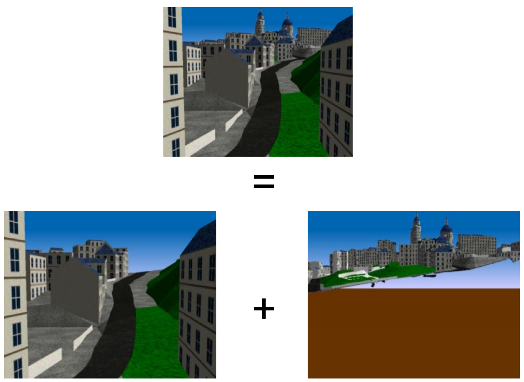
Images + Geometry



Foreground

Background Impostor

Images + Geometry (Rendered View)

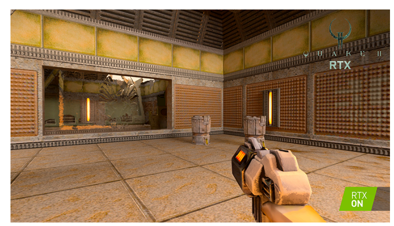


Foreground

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!)



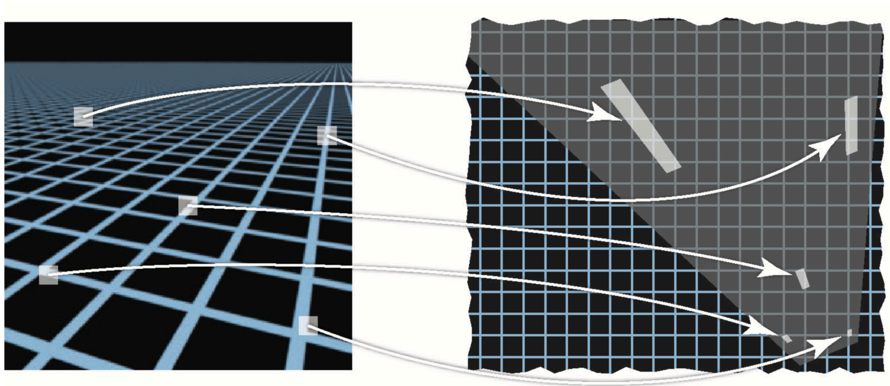
Antialiasing and Mipmaps

Problem: Sampling Textures Can Lead to Aliasing

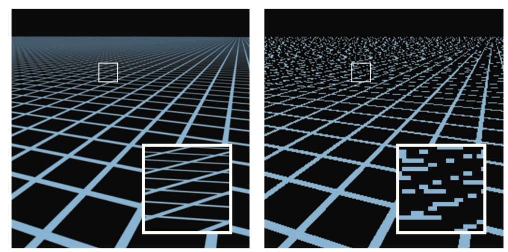
- Just as we've seen with image processing and raytracing applications, if details are not captured with sufficient samples we can see noticeable artifacts
- Solution: use a better sampling/reconstruction

Pixel Footprints

- Can vary in size, shape, and orientation relative to the texture
- Problem: Which of the texture pixels show we pick for each image pixel ? (blue or black)



Answer: neither blue nor black is correct. We need to average them.

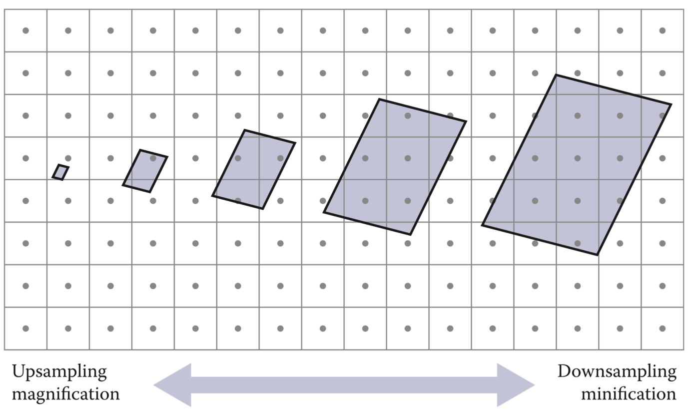


High-resolution image   Point sampling

To resolve the aliasing problem: For each rendered image pixel, we need to average multiple texture pixels. Their number might be large.

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

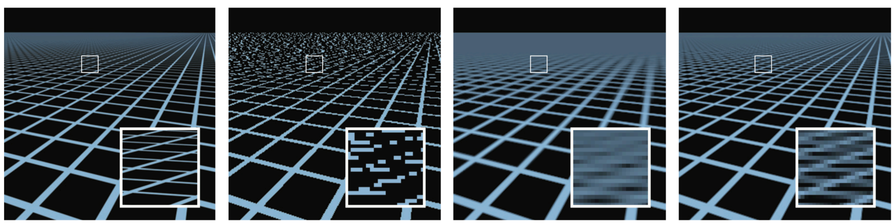


Mipmap

- More or less the same idea as "level of details"
- Antialiasing is only one of the applications of 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



High-resolution image　　Point sampling