# 13. Texture Mapping



# Reading

Required

- Watt, intro to Chapter 8 and intros to 8.1, 8.4, 8.6, 8.8.

Recommended

- Paul S. Heckbert. Survey of texture mapping. **IEEE Computer Graphics and Applications** 6(11): 56--67, November 1986.

Optional

- Watt, the rest of Chapter 8
- Woo, Neider, & Davis, Chapter 9
- James F. Blinn and Martin E. Newell. Texture and reflection in computer generated images. **Communications of the ACM** 19(10): 542--547, October 1976.



# Texture mapping

*Texture mapping (Woo et al., fig. 9-1)*

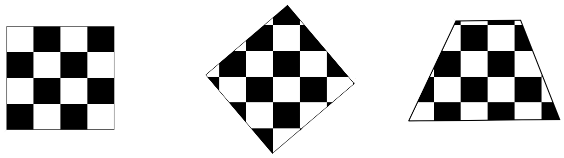
Texture mapping allows you to take a simple polygon and give it the appearance of something much more complex.

- Due to Ed Catmull, PhD thesis, 1974
- Refined by Blinn & Newell, 1976

Texture mapping ensures that "all the right things" happen as a textured polygon is transformed and rendered.



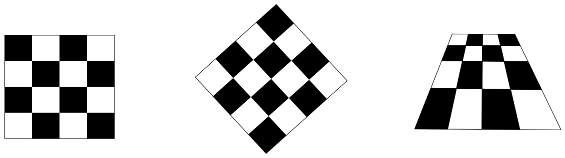
# Non-parametric texture mapping



With "non-parametric texture mapping":

- Texture size and orientation are fixed
- They are unrelated to size and orientation of polygon
- Gives cookie-cutter effect

## Parametric texture mapping



With "parametric texture mapping," texture size and orientation are tied to the polygon.

Idea:

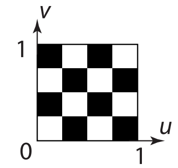
- Separate "texture space" and "screen space"
- Texture the polygon as before, but in texture space
- Deform (render) the textured polygon into screen space

A texture can modulate just about any parameter – diffuse color, specular color, specular exponent, …

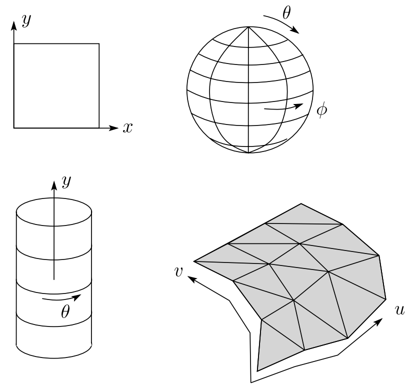


## Implementing texture mapping

A texture lives in it own image coordinates paramaterized by ($u$,$v$):




It can be wrapped around many different surfaces:




Computing ($u$,$v$) texture coordinates in a ray tracer is fairly straightforward.

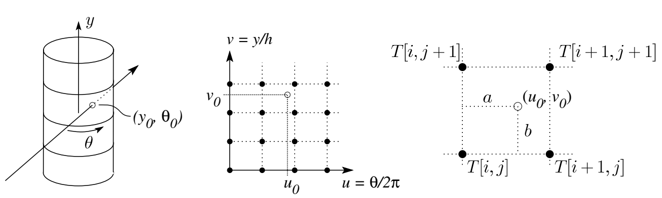
Note: if the surface moves/deforms, the texture goes with it.



## Texture resampling

The texture is usually stored as an image.

**Q**: What do you do when the texture sample you need lands between texture pixels?




Ray intersection | Mapping to texture pixels | Close-up

We need to **resample** the texture. A common choice is **bilinear resampling**:

$$T(u_0, v_0) = T(i\Delta + a, j\Delta + b)$$

$= \_\_\_\_\_\_\_\_\_\_ T[i,j] +$

$\_\_\_\_\_\_\_\_\_\_ T[i+1,j] +$

$\_\_\_\_\_\_\_\_\_\_ T[i,j+1] +$

$\_\_\_\_\_\_\_\_\_\_ T[i+1,j+1]$

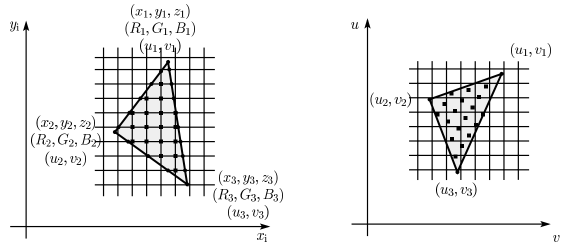


## Texture mapping and the z-buffer

Texture-mapping can also be handled in z-buffer algorithms.

Method:

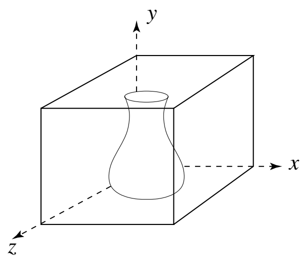
- Scan conversion is done in screen space, as usual
- Each pixel is colored according to the texture
- Texture coordinates are found by Gouraud-style interpolation




Note: Mapping is more complicated if you want to do perspective right!

# Solid textures

**Q**: What kinds of artifacts might you see from using a marble veneer instead of real marble?




One solution is to use **solid textures**:

- Use model-space coordinates to index into a 3D texture
- Like "carving" the object from the material

One difficulty of solid texturing is coming up with the textures.



# Solid textures (cont'd)

Here's an example for a vase cut from a solid marble texture:

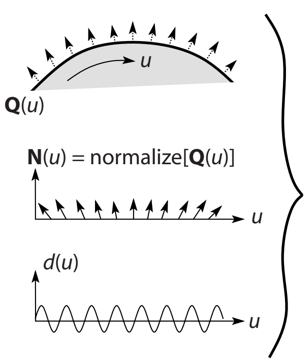
*Solid marble texture by Ken Perlin, (Foley, IV-21)*



# Displacement mapping

Textures can be used for more than just color.

In **displacement mapping**, a texture is used to perturb the surface geometry itself:




$$\tilde{\mathbf{Q}}(u) = \mathbf{Q}(u) + d(u)\mathbf{N}(u)$$

- These displacements "animate" with the surface

**Q**: Do you have to do hidden surface calculations on $\tilde{\mathbf{Q}}$?

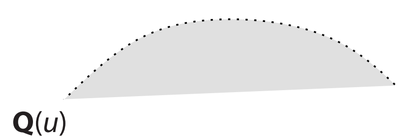


# Bump mapping

In **bump mapping**, a texture is used to perturb the normal:

- Use the original, simpler geometry, $\mathbf{Q}(u)$, for hidden surfaces
- Use the normal from the displacement map for shading:

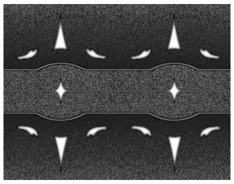
$$\tilde{\mathbf{N}} = \text{normal}[\tilde{\mathbf{Q}}(u)]$$



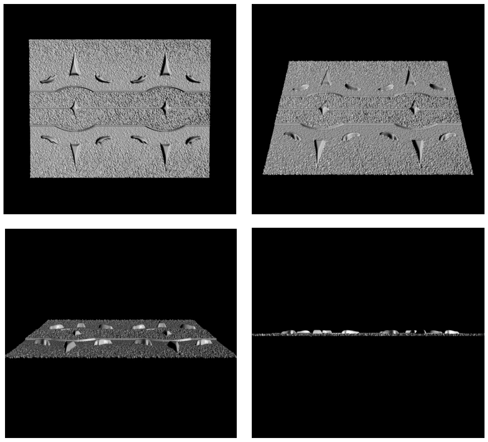

**Q**: What artifacts in the images would reveal that bump mapping is a fake?

## Displacement vs. bump mapping

Input texture



Rendered as displacement map over a rectangular surface





## Displacement vs. bump mapping (cont'd)

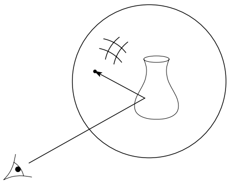
Original rendering

Rendering with bump map wrapped around a cylinder

*Bump map and rendering by Wyvern Aldinger*



## Environment mapping



In **environment mapping** (also known as **reflection mapping**), a texture is used to model an object's environment:

- Rays are bounced off objects into environment
- Color of the environment used to determine color of the illumination
- Really, a simplified form of ray tracing
- Environment mapping works well when there is just a single object – or in conjunction with ray tracing

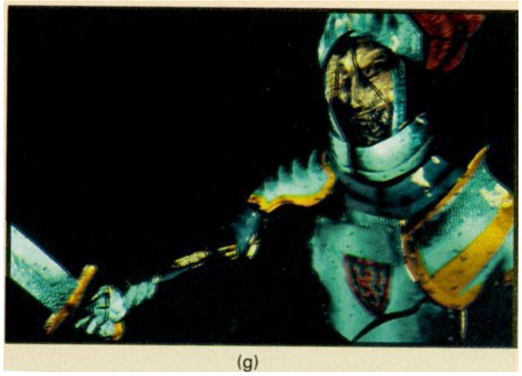
Under simplifying assumptions, environment mapping can be implemented in hardware.

With a ray tracer, the concept is easily extended to handle refraction as well as reflection.



## Combining texture maps

Using texture maps in combination gives even better effects, as *Young Sherlock Holmes* demonstrated …




*Construction of the glass knight, (Foley, IV-24)*

## Combining texture maps (cont'd)

Phong lighting with diffuse texture

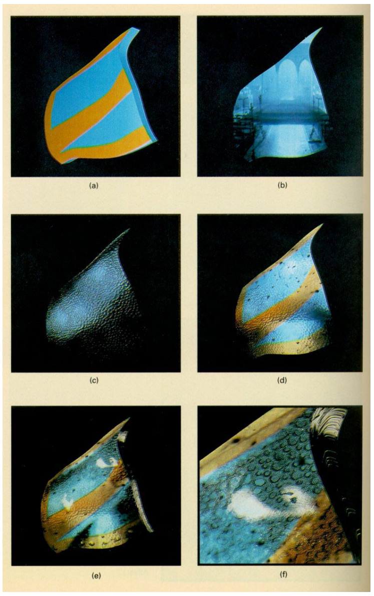
Environment-mapped mirror reflection

Bump mapping + Glossy reflection

Combine textures and add dirt

Rivet stains + Shinier reflections

Close-up

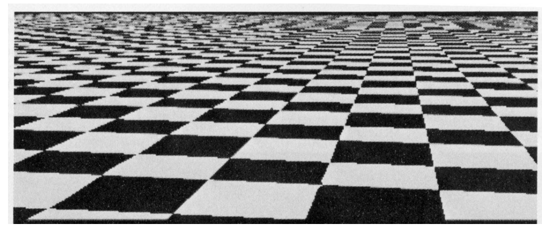



*Construction of the glass knight, (Foley, IV-24)*



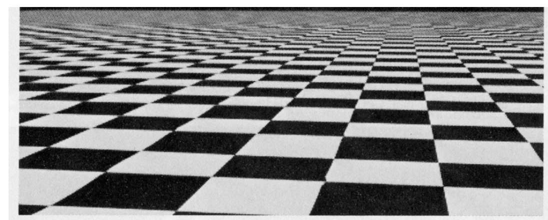
## Antialiasing

If you point-sample the texture map, you get aliasing:



*From Crow, SIGGRAPH '84*

Proper antialiasing requires area averaging in the texture:



*From Crow, SIGGRAPH '84*



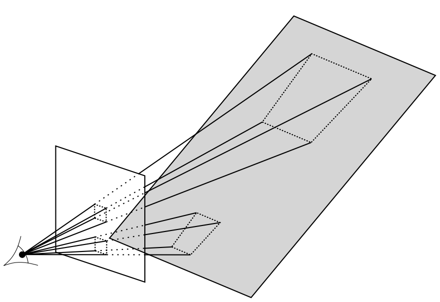
## Computing the average color

The computationally difficult part is summing over the covered pixels.

Several methods have been used:

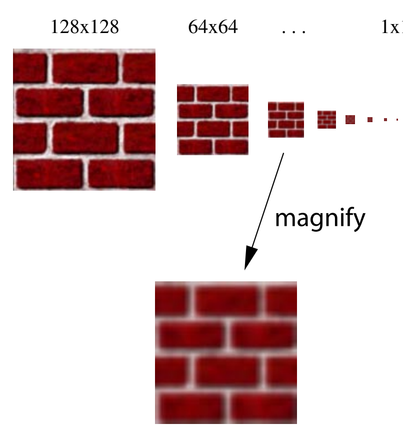
The simplest is **brute force**:

- Figure out which texels are covered and add up their colors to compute the average.





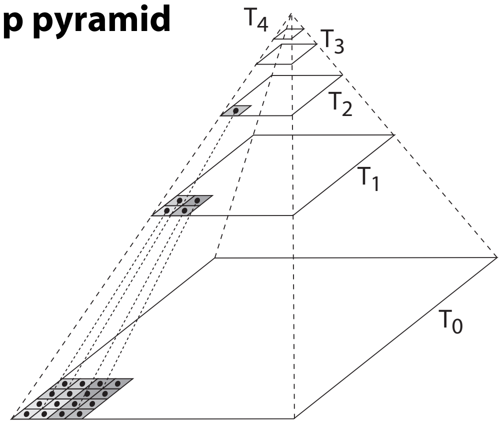
## Mip maps




A faster method is **mip maps** developed by Lance Williams in 1983:

- Stands for “multum in parvo” – many things in a small place
- Keep textures prefiltered at multiple resolutions
- Has become the graphics hardware standard

## Mip map pyramid



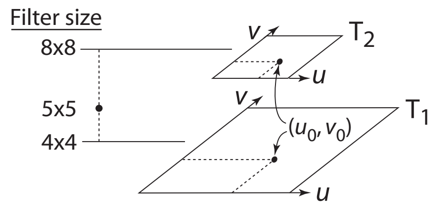

The mip map hierarchy can be thought of as an image pyramid:

- Level 0 ($T_0$[i,j]) is the original image.
- Level 1 ($T_1$[i,j]) averages over 2x2 neighborhoods of original.
- Level 2 ($T_2$[i,j]) averages over 4x4 neighborhoods of original
- Level 3 ($T_3$[i,j]) averages over 8x8 neighborhoods of original

What's a fast way to pre-compute the texture map for each level?



## Mip map resampling




What would the mip-map return for an average over a 5x5 neighborhood at location $(u_0, v_0)$?

How do we measure the fractional distance between levels?

What if you need to average over a non-square region?



## Summed area tables

A more accurate method than mip maps is **summed area tables** invented by Frank Crow in 1984.

Recall from calculus:

$$\int_a^b f(x)dx = \int_{-\infty}^b f(x)dx - \int_{-\infty}^a f(x)dx$$

In discrete form:

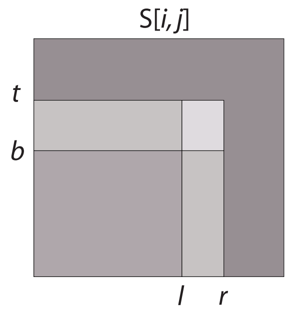
$$\sum_{i=k}^{m} f[i] = \sum_{i=0}^{m} f[i] - \sum_{i=0}^{k} f[i]$$

**Q**: If we wanted to do this real fast, what might we pre-compute?



## Summed area tables (cont'd)

We can extend this idea to 2D by creating a table, S[*i*,*j*], that contains the sum of everything below and to the left.




**Q**: How do we compute the average over a region from (*l*, *b*) to (*r*, *t*)?

Characteristics:

- Requires more memory
- Gives less blurry textures

# Comparison of techniques

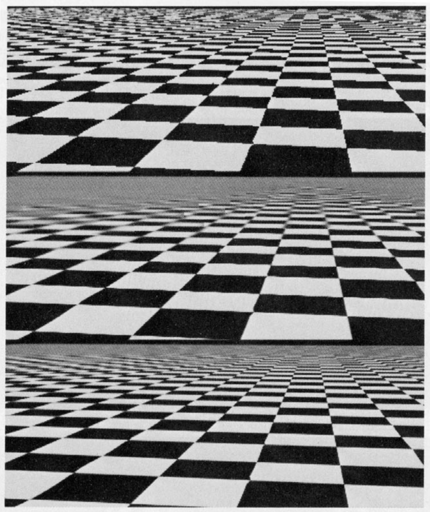



Figure 5: CheckerBoards mapped onto a square showing vertically compressed texture.

*From Crow, SIGGRAPH '84*



# Summary

What to take home from this lecture:

1. The meaning of the boldfaced terms.
2. Familiarity with the various kinds of texture mapping, including their strengths and limitations.
3. Understanding of the various approaches to antialiased texture mapping:
   - Brute force
   - Mip maps
   - Summed area tables