Computer Graphics

Lecture 7

Texture Mapping, Bump-mapping, Transparency

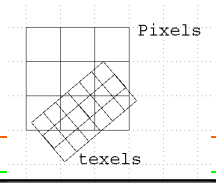
Today

- Texture mapping
  - Anti-aliasing techniques
- Bump mapping
- Transparency

Aliasing

Happens when

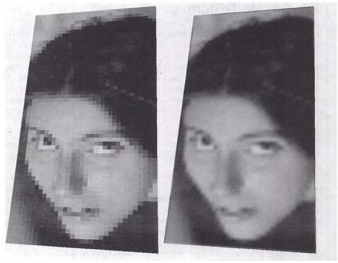
- The camera is zoomed too much into the textured surface (magnification)
- Several texels covering a pixel's cell (minification)



Texture Magnification

Zooming into a surface with a texture too much

One texel covering many pixels

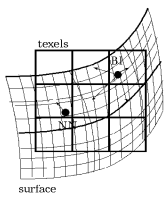


Texture Magnification

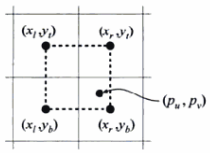
Methods to determine the color of each pixel

- Nearest neighbour (using the colour of the closest texel)
- Bilinear interpolation (linearly interpolating the colours of the surrounding texels)

NN BI



Bilinear Interpolation



$$\mathbf{b}(p_u, p_v) = (1-u')(1-v')\mathbf{t}(x_l, y_b) + u'(1-v')\mathbf{t}(x_r, y_b) + (1-u')v'\mathbf{t}(x_l, y_t) + u'v'\mathbf{t}(x_r, y_t).$$

- (pu,pv) : the pixel centre mapped into the texture space
- b(pu,pv) : the colour at point pu, pv
- t(x,y) : the texel colour at (x,y)
- u = pu – (int)pu, v = pv - (int)pv

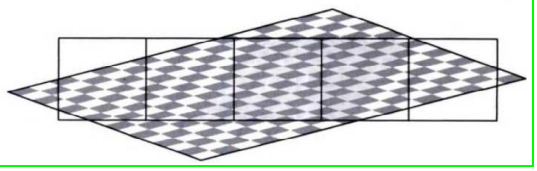
Texture Minification

Many texels covering a pixel's cell

Results in aliasing (remember Nyquist limit)

The artifacts are even more noticeable when the surface moves

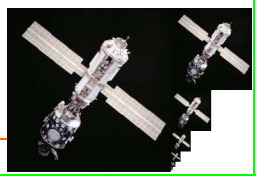
Solution
– Mipmapping



MIP map

Multum In Parvo = Many things in a small place

Produce a texture of multiple resolutions
Switch the resolution according to the number of texels in one pixel
Select a level that the ratio of the texture and the pixel is 1:1



Selecting the resolution in Mipmap

Map the pixel corners to the texture space
Find the resolution that roughly covers the mapped quadrilateral
Apply a bilinear interpolation in that resolution,
Or find the two surrounding resolutions and apply a trilinear interpolation (also along the resolution axis)

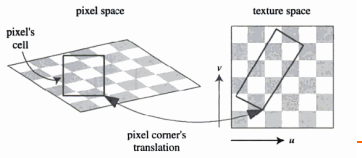
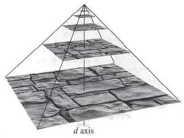
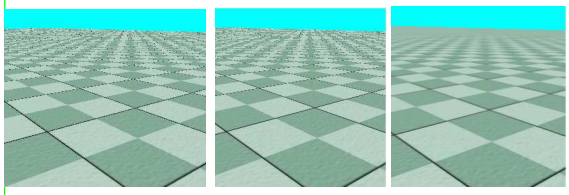


Figure 5.13. On the left is a square pixel cell and its view of a texture. On the right is the projection of the pixel cell onto the texture itself.



Texture Minification

Multiple textures in a single pixel
Solution:

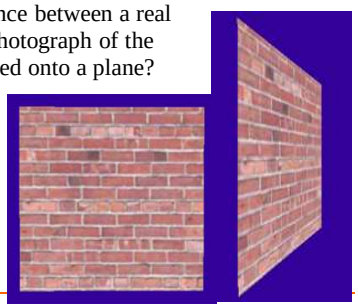


Nearest neighbour Bilinear blending Mipmapping

What's Missing?

What's the difference between a real brick wall and a photograph of the wall texture-mapped onto a plane?

What happens if we change the lighting or the camera position?



Bump Mapping

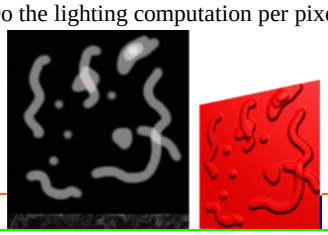
Use textures to alter the surface normal
Does not change the actual shape of the surface
Just shaded as if it were a different shape

Bump Mapping

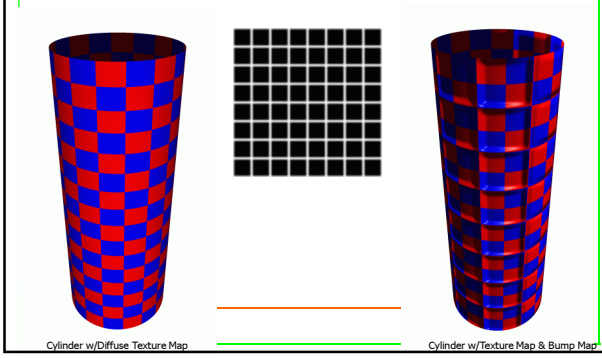
Treat the texture as a single-valued height function

Compute the normal from the partial derivatives in the texture

Do the lighting computation per pixel

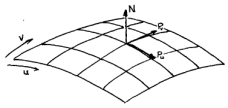


Another Bump Map Example

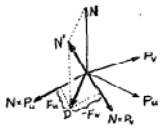


Computing the normals

$$\mathbf{n}' = \mathbf{n} + \frac{F_u(\mathbf{n} \times \mathbf{P_v}) - F_v(\mathbf{n} \times \mathbf{P_u})}{||\mathbf{n}||}$$



- **n** the normal vector at the surface
- **n'** the updated normal vector
- **Pu**, **Pv** are partial derivatives of the surface in the u and v direction
- Fu, Fv are the gradients of the bump map along the u and v axes in the bump texture



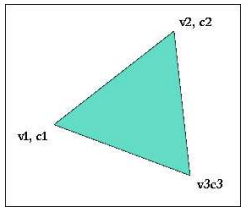
Computing **Pu** and **Pv**

- Do this for every triangle:

$$\Delta \mathbf{v2v1} = \mathbf{v2} - \mathbf{v1}, \quad \Delta \mathbf{v3v1} = \mathbf{v3} - \mathbf{v1}$$
$$\Delta \mathbf{c2c1} = \mathbf{c2} - \mathbf{c1}, \quad \Delta \mathbf{c3c1} = \mathbf{c3} - \mathbf{c1}$$

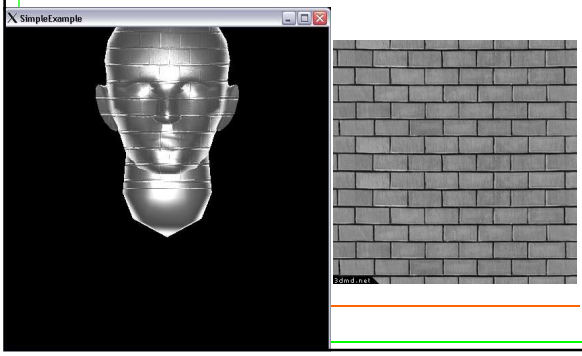
$$\begin{bmatrix} \mathbf{Pu} \\ \mathbf{Pv} \end{bmatrix} = \begin{bmatrix} \frac{\Delta \mathbf{v2v1}\Delta \mathbf{c3c1}_v - \Delta \mathbf{v3v1}\Delta c2\mathbf{c1}_v}{M} \\ \frac{\Delta \mathbf{v3v1}\Delta 12\mathbf{c1}_u - \Delta \mathbf{v2v1}\Delta c3\mathbf{c1}_v}{M} \end{bmatrix}$$

$$M = \Delta \mathbf{c2c1}_u \Delta \mathbf{c3c1}_v - \Delta \mathbf{c3c1}_u \Delta \mathbf{c2c1}_v$$

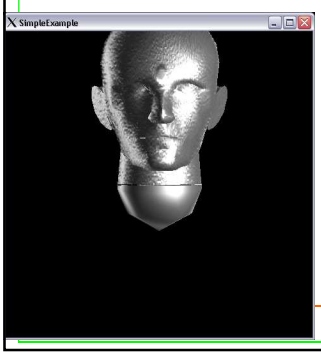


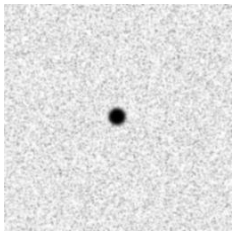
- v1,v2,v3 : 3D coordinates
- c1,c2,c3 : texture coordinates
- http://www.blacksmith-studios.dk/projects/downloads/tangent_matrix_derivation.php

Some more examples

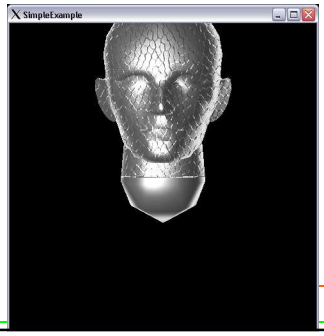


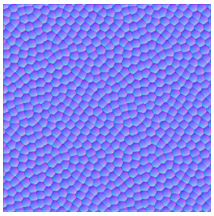
Some more examples





Some more examples

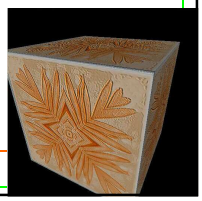




Emboss Bump Mapping

Real bump mapping uses per-pixel lighting
- Lighting calculation at each pixel based on perturbed normal vectors
- Computationally expensive

Emboss bump mapping is a hack
- Diffuse lighting only, no specular component
- Can use per vertex lighting
- Less computation



Diffuse Lighting Calculation

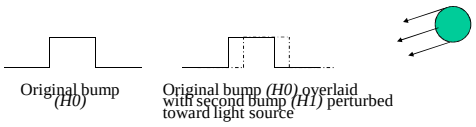
$C = (L \bullet N) \times Dl \times Dm$

- *L* is light vector
- *N* is normal vector
- *Dl* is light diffuse color
- *Dm* is material diffuse color
- Bump mapping changes *N* per pixel
- Emboss bump mapping approximates *(L•N)*

Approximate diffuse factor L•N

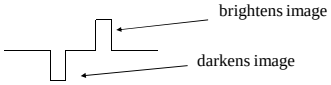
Texture map represent height field
- [0,1] height represents range of bump function
- First derivative represents slope *m*
- *m* increases/decreases base diffuse factor *Fd*
- *(Fd+m)* approximates *(L•N)* per pixel

Compute the Bump



Original bump *(H0)*

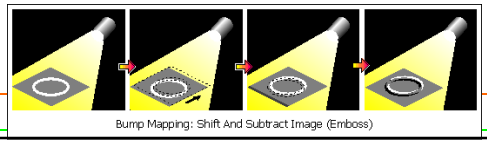
Original bump *(H0)* overlaid with second bump *(H1)* perturbed toward light source



Subtract original bump from second *(H1-H0)*

Approximate derivative

Embossing approximates derivative
- Lookup height *H0* at point *(s,t)*
- Lookup height *H1* at point slightly perturbed toward light source *(s+Δs, t+Δt)*
- subtract original height *H0* from perturbed height *H1*
- difference represents instantaneous slope *m=H1-H0*



Compute the Lighting

Evaluate fragment color Cf

Cf = (L•N) × Dl × Dm

(L•N) ≅ (Fd + (H1-H0))

Dm × Dl encoded in surface texture color *Ct*

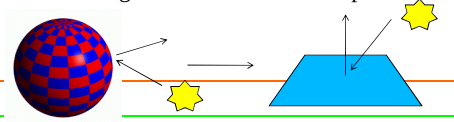
Cf = (Fd + (H1-H0)) × Ct

Required Operations

Calculate texture coordinate offsets *Δs, Δt*
Calculate diffuse factor *Fd*
Both are derived from normal *N* and light vector *L*
Only done per vertex
Computation of H1-H0 done per pixel

Calculate Texture Offsets

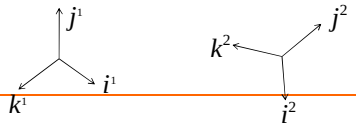
Rotate light vector into normal space
- Need Normal coordinate system
- Derive coordinate system from normal and "up" vector
  - Normal is z-axis
  - Cross product is x-axis
  - Throw away up vector, derive y-axis as cross product of x- and z-axes
- Build 3x3 matrix from axes
- Transform light vector into Normal space



Transforming the coordinates

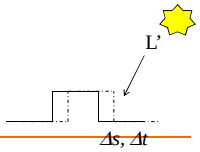
$$R_1 = \begin{pmatrix} i^1_x & j^1_x & k^1_x \\ i^1_y & j^1_y & k^1_y \\ i^1_z & j^1_z & k^1_z \end{pmatrix}, R_2 = \begin{pmatrix} i^2_x & j^2_x & k^2_x \\ i^2_y & j^2_y & k^2_y \\ i^2_z & j^2_z & k^2_z \end{pmatrix}$$

$$R_{1\to 2} = R_2 R_1^{-1}$$



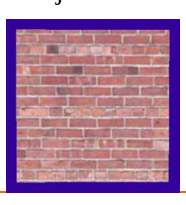
Calc Texture Offsets (cont'd)

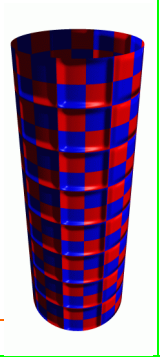
Use normal-space light vector for offsets
- L' = T(*L*) : T is the transformation
- Use L'x, L'y for *Δs, Δt*
- *Use L'z for diffuse factor (Fd)*
  - *If light vector is near normal, L'x, L'y are small*
  - *If light vector is near tangent plane, L'x and L'y are large*



What's Missing?

There are no bumps on the silhouette of a bump-mapped object

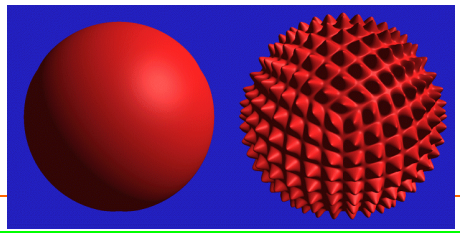




Displacement Mapping

Use the texture map to actually move the surface point

The geometry must be displaced before visibility is determined



Transparency

Sometimes we want to render transparent objects

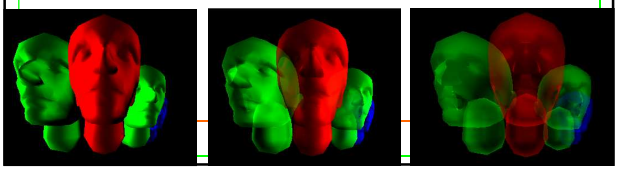
We blend the colour of the objects along the same ray

Apply alpha blending

Alpha

Another variable called alpha is defined here
This describes the opacity
Alpha = 1.0 means fully opaque
Alpha = 0.0 means fully transparent

α = 1 α = 0.5 α = 0.2



Sorting by the depth

First, you need to save the depth and colour of all the fragments that will be projected onto the same pixel in a list

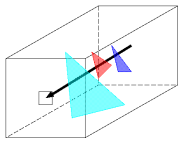
Then blend the colour from back towards the front

The colours of overlapping fragments are blended as follows:

$C_o = \alpha C_s + (1-\alpha) C_d$

C_s : colour of the transparent obje colour before blending, C_o is th result of blending

Do this for all the pixels



Sorting the fragment data by the depth – use stl::sort

```
#include <algorithm>

struct FragInfo
{
      float z;
      float color[3];
};

bool PixelInfoSortPredicate(const PixelInfo * d1, const PixelInfo * d2)
{
       return d1->z < d2->z;
}


main()
{
      FragInfo f1,f2,f3;

      f1.z = 1; f2.z = -2; f3.z = -5
      vector <FragInfo> flist ;

      flist.push_back(f1); flist.push_back(f2) ; flist.push_back(f3);

      std::sort(flist.begin(), flist.end(), PixelInfoSortPredicate);

}
```


Readings

- Blinn, "Simulation of Wrinkled Surfaces", Computer Graphics, (Proc. Siggraph), Vol. 12, No. 3, August 1978, pp. 286-292.
- Real-time Rendering, Chapter 5,1-5.2
- http://www.blacksmith-studios.dk/projects/downloads/tangent_matrix_derivation.php
- http://developer.nvidia.com/object/emboss_bump_mapping.html