MANCHESTER 1824

COMP27112

Computer Graphics and Image Processing

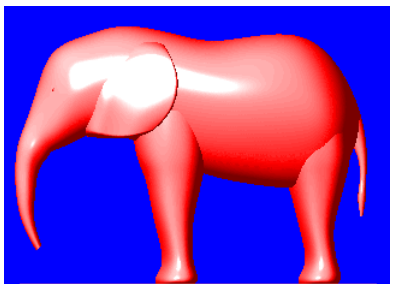
9: Rendering (3)

Toby.Howard@manchester.ac.uk

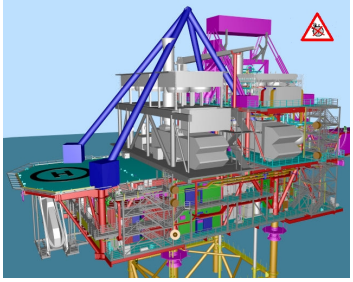
1

MANCHESTER 1824

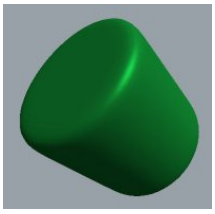
Examples of the simple local model



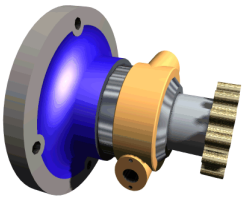
Lex Lennings



AIG



James Sinnott



Alan Watt

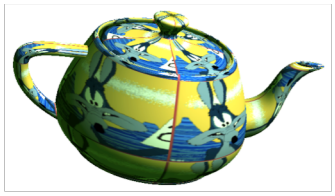
nVIDIA

2

MANCHESTER 1824

Surface detail

- We'll look at two methods for adding **surface detail** to rendered surfaces
  - texture mapping



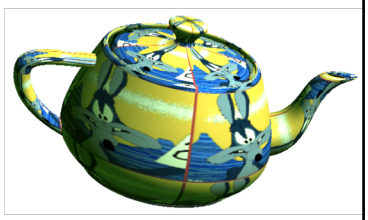
  - bump mapping

3

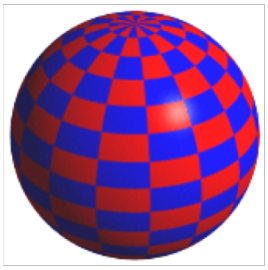
MANCHESTER 1824

Types of texture mapping

- image-based
  - the texture is defined by an image which modifies pixel colours. This is the most common approach and we'll focus on this.



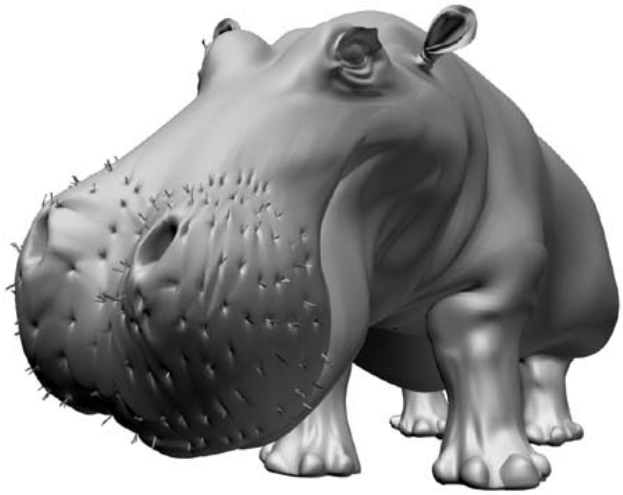
- procedural
  - a texture pattern is computed during rendering, based on algorithms/rules, which modifies pixel colour. Less common, and we won't look further at this.



4

MANCHESTER 1824

Texture mapping



Jeremy Birn

5

MANCHESTER 1824

Texture mapping

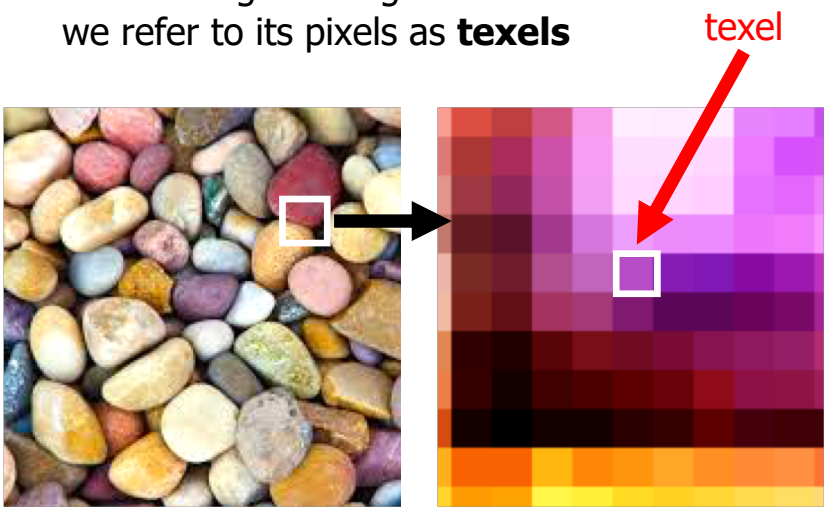
Jeremy Birn

6

MANCHESTER 1824

Defining a texture

- When using an image as a texture we refer to its pixels as **texels**

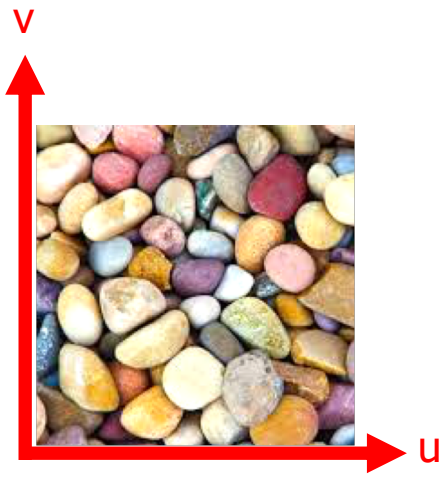


7

MANCHESTER 1824

Texture coordinates

- for convenience, a texture is defined in its own coordinate system
- this is conventionally referred to as (u,v) or (s,t)

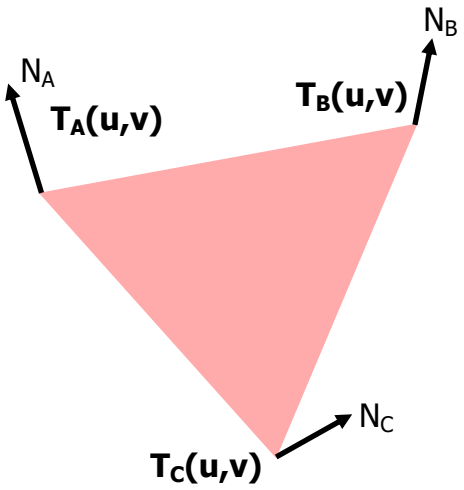


8

MANCHESTER 1824

Mapping texture per-polygon

- We associate (u,v) texture coordinates with each (x,y,z) vertex of a polygon
- And interpolate the texture coordinates during scan-conversion
- Then we **blend** the pixel colour with the texture colour

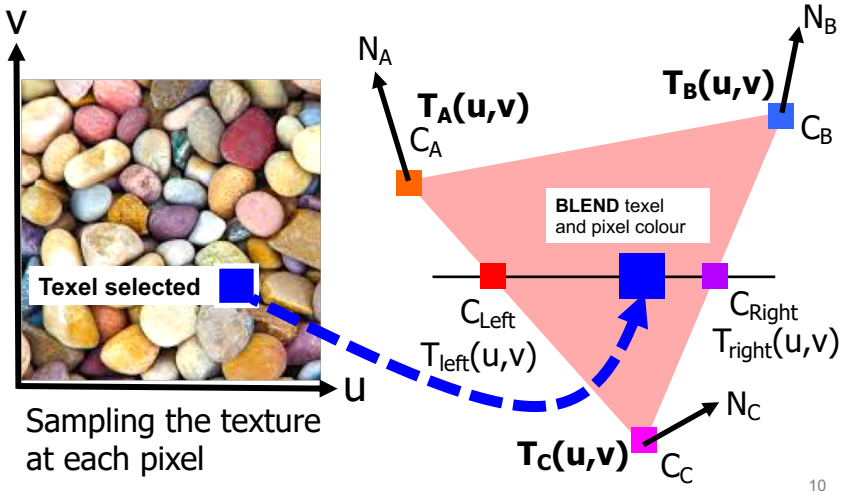


9

MANCHESTER 1824

Performing texture mapping

- We do texture mapping as part of rasterisation



Sampling the texture at each pixel

10

MANCHESTER 1824

Blending texture and pixel colours

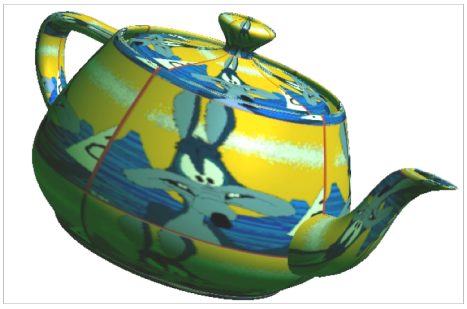
- Programmer has fine control of blending
- Examples: ADD, MODULATE, DECAL, BLEND, REPLACE, COMBINE
- See Coursework 3
- See ex11.c in OpenGL manual
- Details are not examinable
- https://www.khronos.org/registry/OpenGL-Refpages/gl2.1/xhtml/glTexEnv.xml

11

MANCHESTER 1824

Meshes and seams

- In order to realistically texture a mesh, we often have to use multiple textures in different places, which can give rise to ugly "seams"



- One solution is to use textures that are **seamless**, so the edge of one exactly matches the edge of another
- Creating textures is an art!

MANCHESTER 1824

Resolution mismatches

Texture image

rendered triangle

In this particular example, **texture resolution** happens to match **pixel resolution** (this is lucky, and very unusual)

13

MANCHESTER 1824

Resolution mismatches

Texture image

But now the camera has changed...

...so texture resolution no longer matches pixel resolution – this is almost always the situation we have to deal with

14

MANCHESTER 1824

Resolution mismatches

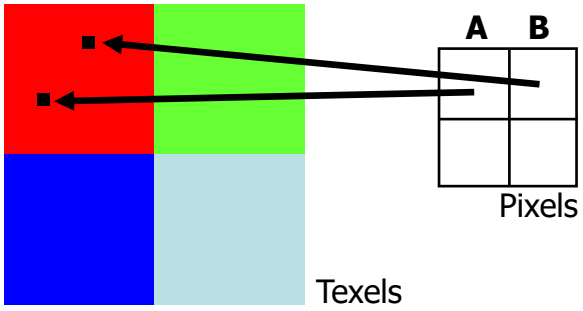
- There are two cases:
  - pixel resolution **>** texture resolution (i.e. pixels are smaller than texels).
  - pixel resolution **<** texture resolution (i.e. pixels are bigger than texels)
- In each case we need to filter, aka sample
- Each case requires a different approach

15

MANCHESTER 1824

pixel resolution > texel resolution

- Pixels **A** and **B** happen to map to the same texel, because pixel resolution > texel resolution



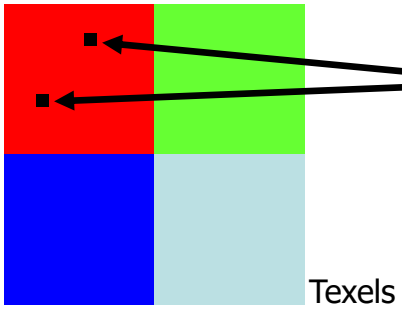
- We' ll look at two approaches (there are more)
  1. no filter
  2. bilinear interpolation filter

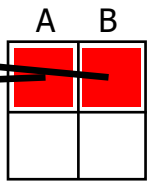
16

MANCHESTER 1824

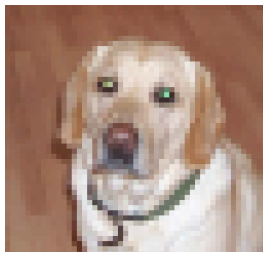
1. No filter

- We simply select the texel to which the pixel maps





Texels



- The resulting rendering looks blocky

17

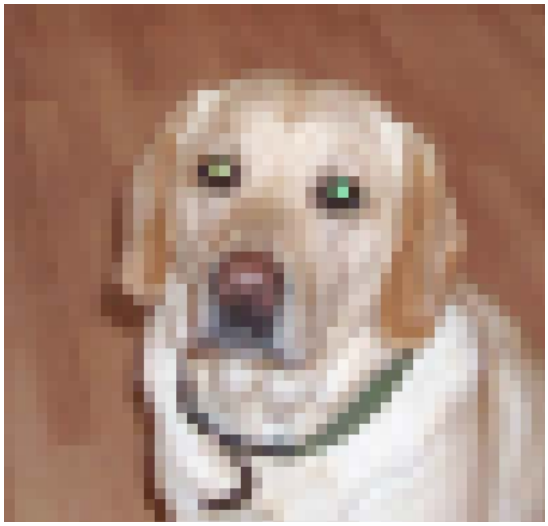
MANCHESTER 1824

Pixel resolution == Texel resolution

18

MANCHESTER 1824

Pixel resolution > Texel resolution: no filter

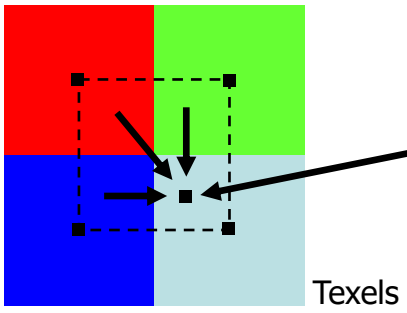


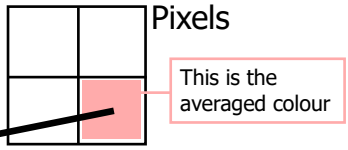
19

MANCHESTER 1824

2. Bilinear interpolation filter

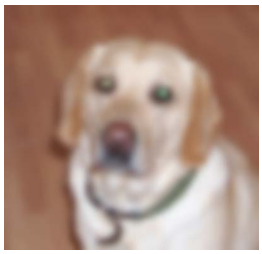
- We compute a texel colour from adjacent texels, **averaging** horizontally and vertically





Texels

- The resulting pixel image looks smoother



20

MANCHESTER 1824

Pixel resolution == Texel resolution

21

MANCHESTER 1824

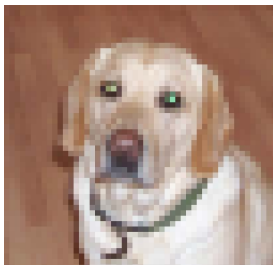
Pixel resolution > Texel resolution: bilinear filter

22

MANCHESTER 1824

Texture filtering: comparison

Unfiltered texture (1 to 1 pixel correspondence)



No filter

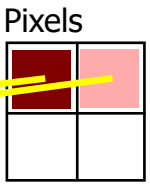
Bilinear interpolation filter

23

MANCHESTER 1824

pixel resolution < texel resolution

- If pixel resolution **<** texel resolution, **adjacent** pixels may map to texels **far apart** in the texture, leading to missing detail, aka aliasing
- In animated sequences especially, we see unpleasant aliasing effects, as pixels "pop" on and off, or change colour unexpectedly in each frame

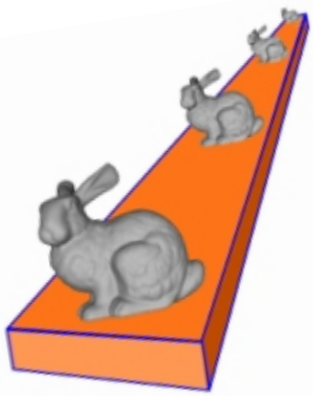


24

MANCHESTER 1824

Mipmap filtering

- One technique to minimise this effect is **mipmapping**
- **mip** = *multum in parvo* (Latin), meaning "many things in a small place"
- The idea is simple: the further away from the viewpoint, the less detail we need
- So, we use a **set** of texture maps, and **select** which map to use, according to the **distance** of a pixel from the viewer



25

MANCHESTER 1824

Creating a mipmap

t0

256 x 256

t0 is the original full-resolution texture

t1

128 x 128

t1 is ½ res

t2

64 x 64

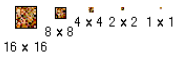
t2 is ¼ res

t3

32 x 32

t3 is 1/8 res (etc)

t4 (etc)



16 x 16 8 x 8 4 x 4 2 x 2 1 x 1

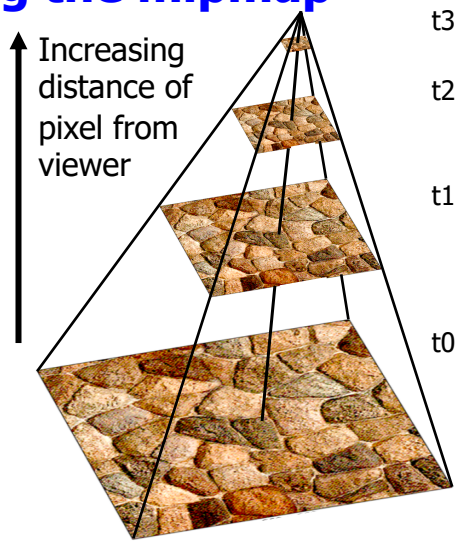
Starting from the original texture **t0**, we repeatedly create smaller versions of it (**t1**, **t2** etc), **downsampling** (i.e., averaging) by ½ each time, until we reach a 1x1 texture. This is a **pre-processing** procedure, and we store each texture in memory.

26

MANCHESTER 1824

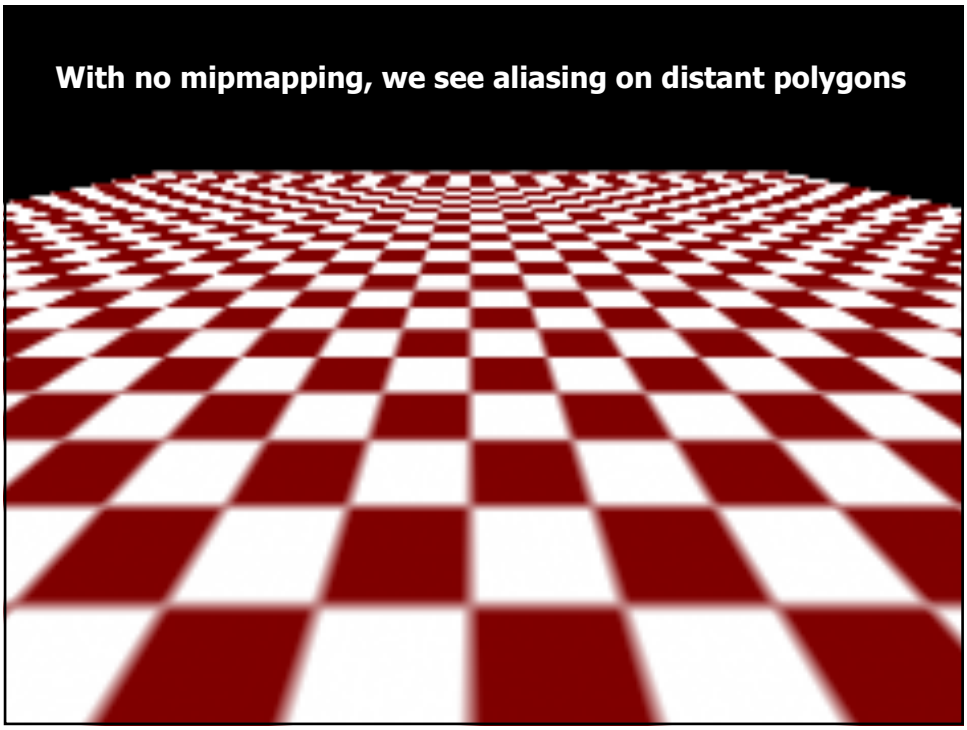
Rendering using the mipmap

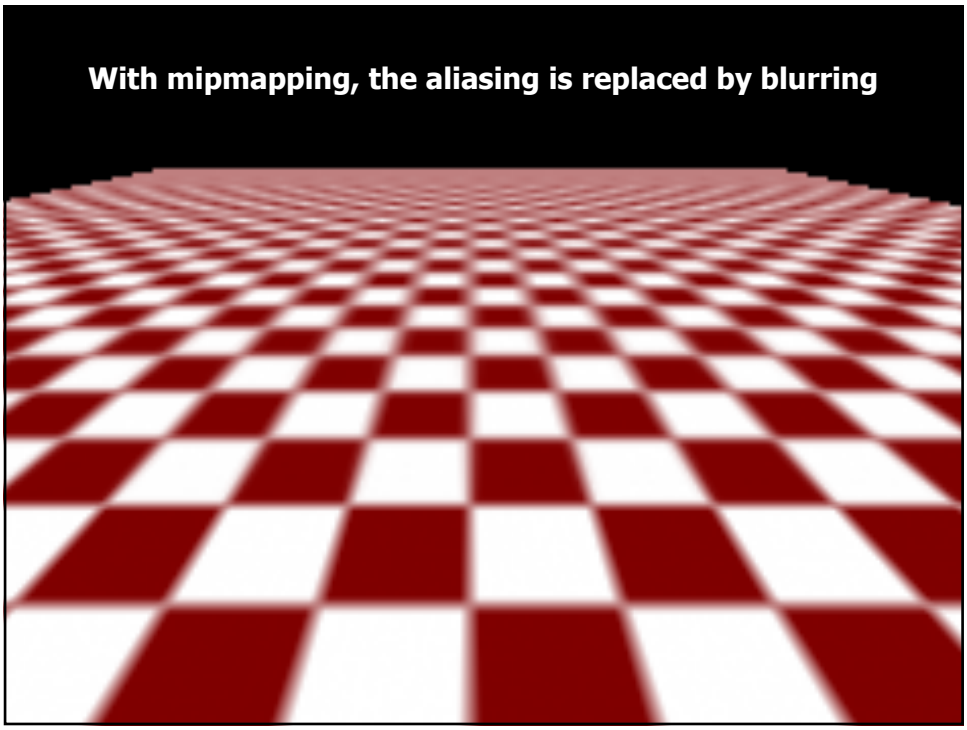
- When rendering, we select one of the textures according to the **distance** of the pixel from the viewpoint
- Or, we can also choose the **two closest** textures, and do bilinear interpolation – for extra smoothness



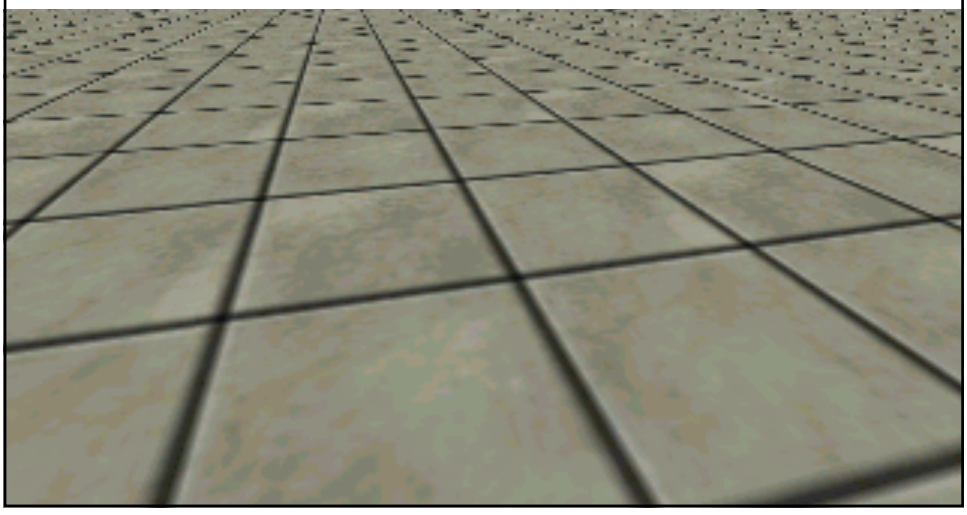
27

With no mipmapping, we see aliasing on distant polygons

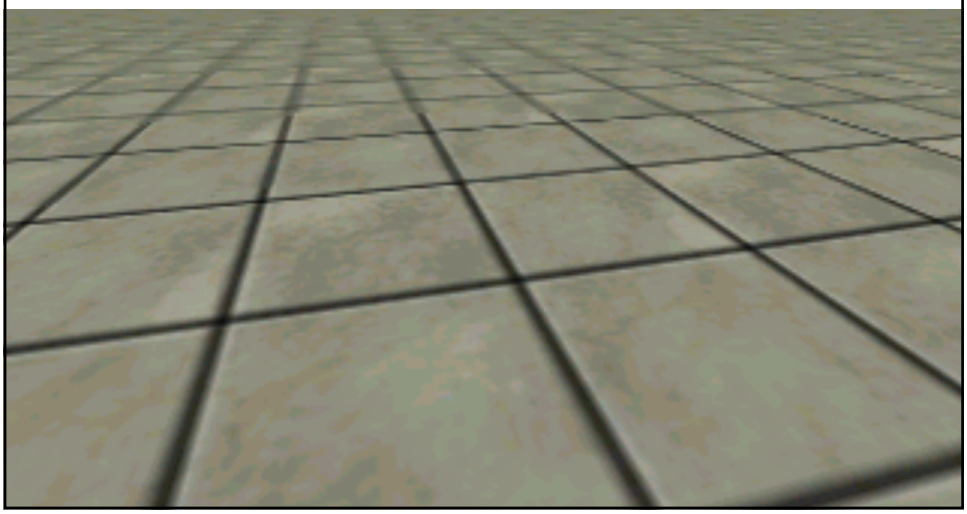




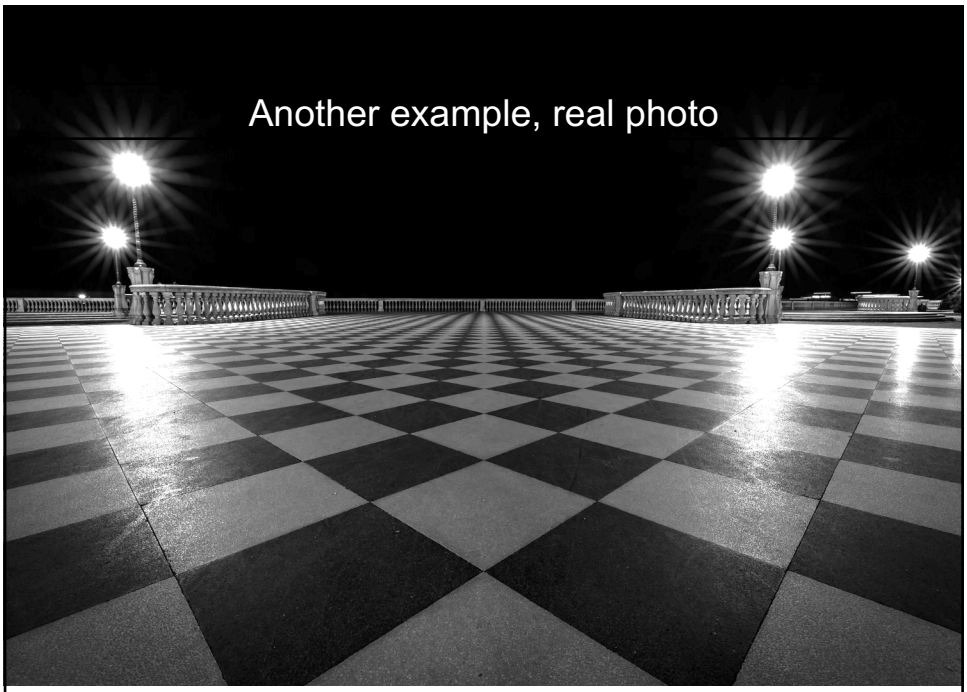
Another example, without mip-mapping



Another example, with mip-mapping



Another example, real photo

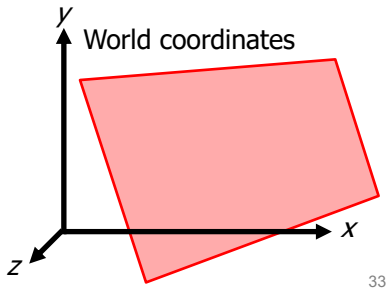


Nico Angeli (https://www.flickr.com/photos/135915437@N08/)

MANCHESTER 1824

Textures in OpenGL

```
glBegin(GL_QUADS);
 glVertex3f(x0, y0, z0);
 glVertex3f(x1, y1, z1);
 glVertex3f(x2, y2, z2);
 glVertex3f(x3, y3, z3);
glEnd();
```

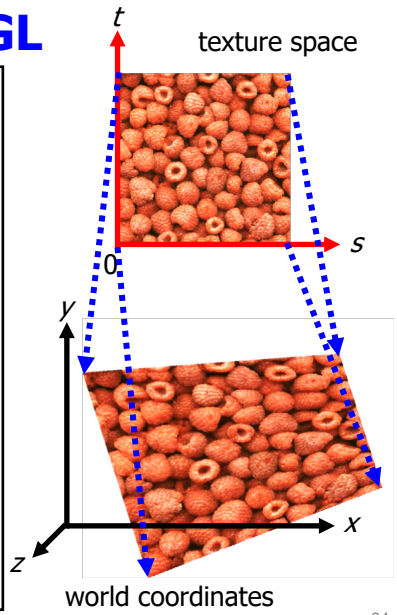


33

MANCHESTER 1824

Textures in OpenGL

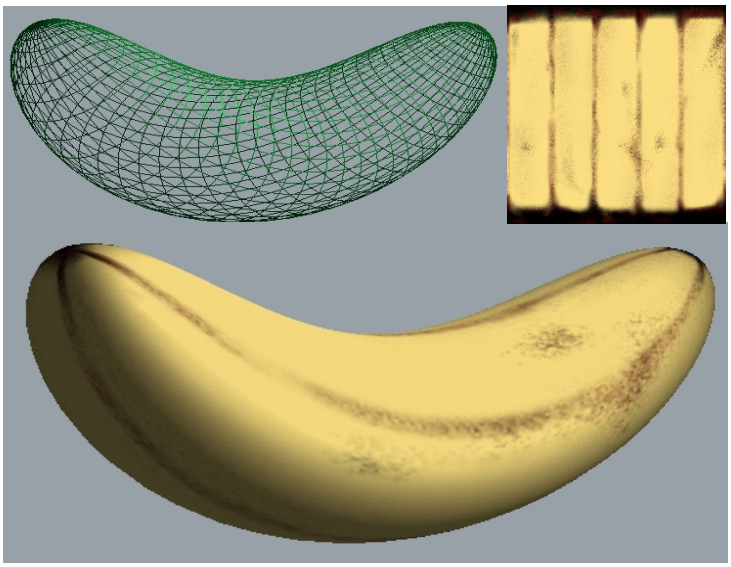
```
/* First call functions to
   read texture from
   image file, and to set
   its properties – how
   it BLENDS with pixels
*/

glBegin(GL_QUADS);
 glTexCoord(0.0, 0.0);
   glVertex3f(x0, y0, z0);
 glTexCoord(1.0, 0.0);
   glVertex3f(x1, y1, z1);
 glTexCoord(1.0, 1.0);
   glVertex3f(x2, y2, z2);
 glTexCoord(0.0, 1.0);
   glVertex3f(x3, y3, z3);
glEnd();
```



34

MANCHESTER 1824

Texture example



James Sinnott

35

MANCHESTER 1824

Textures as illumination

- Textures can be used to add accurate illumination to a real-time scene
- Off-line, pre-compute accurate diffuse illumination of the scene, using a global model (e.g., radiosity)
- Save rendered surfaces as textures – called **lightmaps**
- In real-time, apply lightmap textures, and also actual surface textures.

36

MANCHESTER 1824

Lightmaps only

37

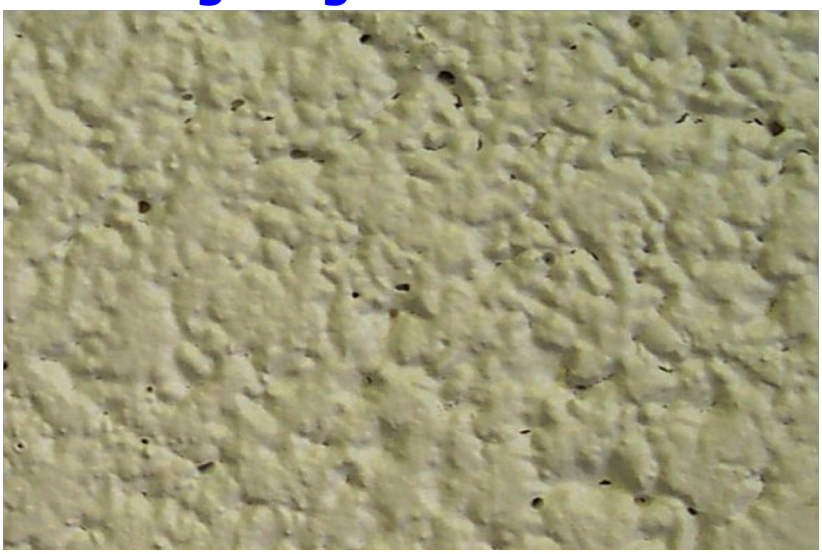
MANCHESTER 1824

Lightmaps blended with other textures

38

MANCHESTER 1824

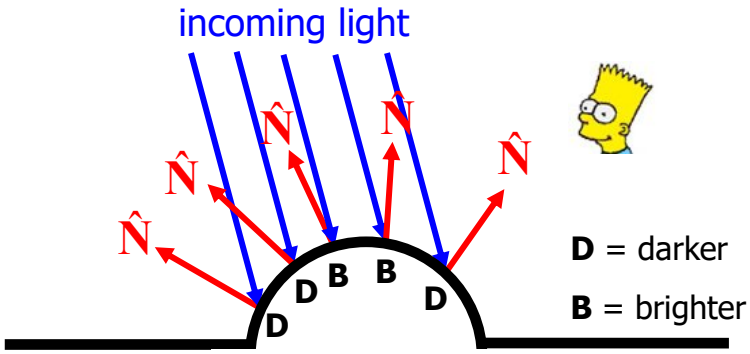
Modelling rough surfaces



39

MANCHESTER 1824

Bump mapping



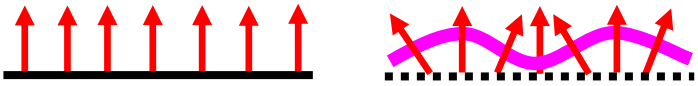
- Why do rough surfaces look rough?
- The **surface normals** change across the bumps, so the top of the bumps appear brighter than the sides

40

MANCHESTER 1824

Bump mapping

- Rather than alter the surface **colour**, as in texturing, we can alter the **surface normal**
- this has the same effect on the illumination model as if the surface were really geometrically different (but it isn't)

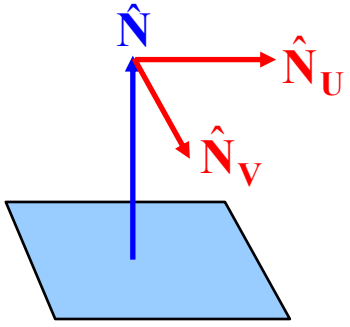


original normals altered normals

41

MANCHESTER 1824

Altering a surface normal

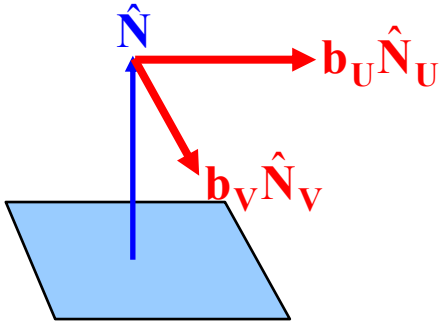


- Step 1: Introduce two **unit** vectors $\hat{N}_U$ and $\hat{N}_V$ each orthogonal to the surface normal

42

MANCHESTER 1824

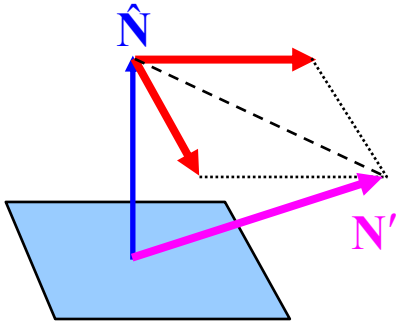
Altering a surface normal



- Step 2: Scale $\hat{\mathbf{N}}_U$ and $\hat{\mathbf{N}}_V$ by $\mathbf{b}_U$ and $\mathbf{b}_V$
- To give the "bump vectors"

MANCHESTER 1824

Altering a surface normal

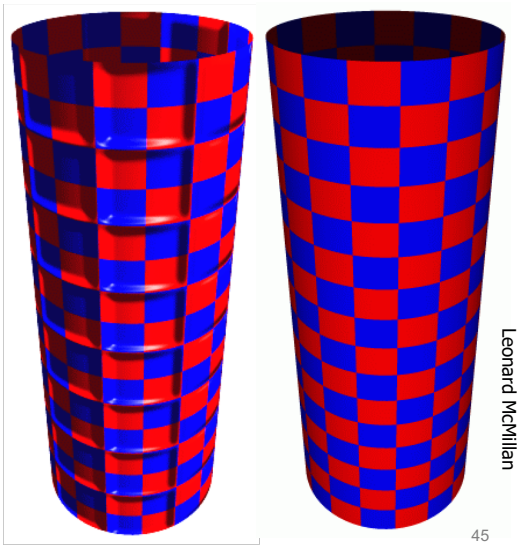


- Step 3: Add the bump vectors to the original surface normal to get an altered surface normal
- $\mathbf{N}' = \hat{\mathbf{N}} + (\mathbf{b}_U\hat{\mathbf{N}}_U + \mathbf{b}_V\hat{\mathbf{N}}_V)$

MANCHESTER 1824

Bump mapping example

- The altered surface normals make the mesh look bumpy
- But only the normals are changing during rendering
- The definition of the mesh does not change



Leonard McMillan

45

MANCHESTER 1824

Where do we get b_U and b_V ?

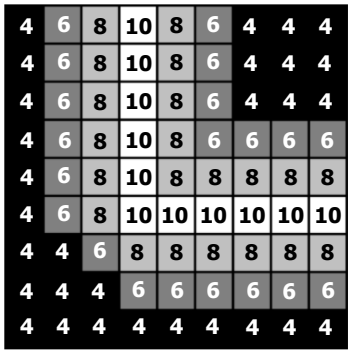
- One way is to use a texture as a **bump map** to derive b_U and b_V
- We use the texel colours to encode bumpiness

46

MANCHESTER 1824

Format of a bump map

- We treat the value of each texel as a "height" which will control the bumpiness of the surface
- We can say that brighter texels are "higher" than darker texels



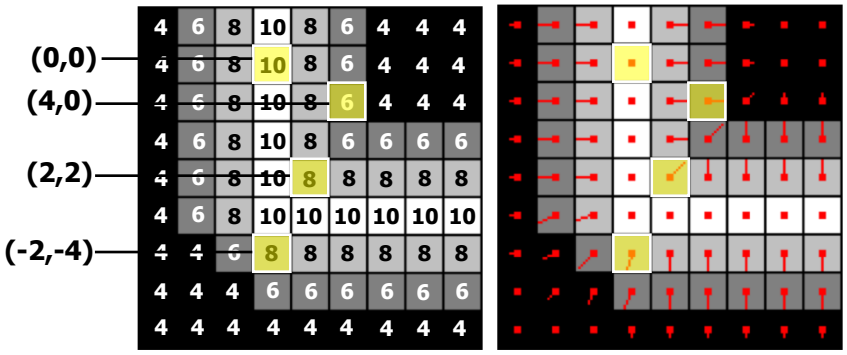
Hugo Elias

47

MANCHESTER 1824

Deriving b_U and b_V

- For each texel T, we compute its x and y brightness **gradients** and use them as b_U and b_V:
  - $b_U = T(x-1, y) - T(x+1, y)$
  - $b_V = T(x, y-1) - T(x, y+1)$

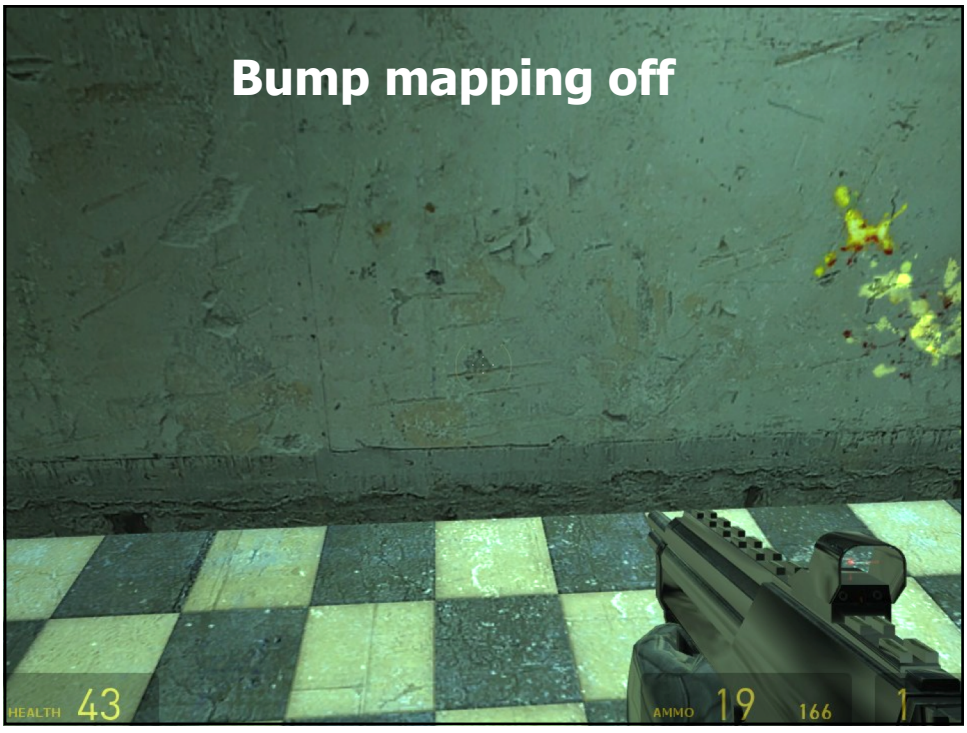


MANCHESTER 1824

Bump mapping example

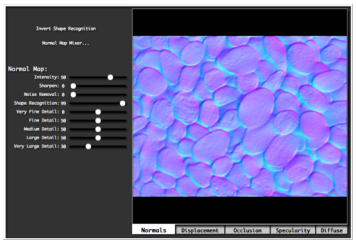
NVIDIA

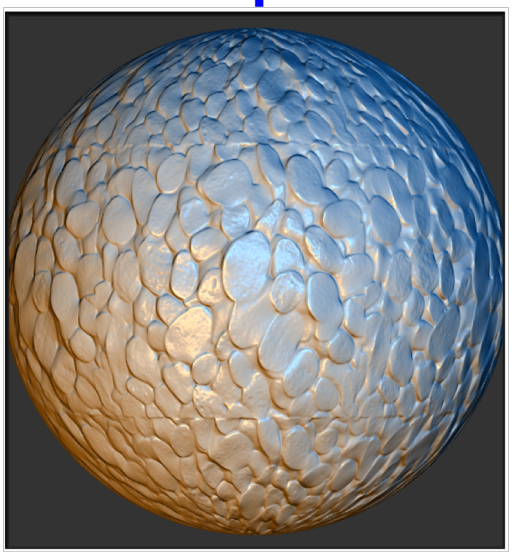
49



MANCHESTER 1824

Crazybump – do some experiments





www.crazybump.com (mac/win)

52

MANCHESTER 1824

What we've covered in Part 1 of the course