Textures

Datorgrafik 2006

Reading Material

MUST read

- These slides
- OH 139-172 by Magnus Bondesson
  - Texturing, mipmapping, multitexturing, environment mapping
- OH 173-174
  - Bump mapping
- OH 175-200
  - 3D-textures, Textures in Art of Illusion, Procedural textures and Perlin Noise
- OH 201
  - Rendering to texture
- OH 228-230
  - Particle systems
- OH 284-291
  - Sprites and billboards
- OH 326 and ”Från Värld till Skärm” chapter 10
  - Perspective correct texture coordinate interpolation
  - Läs översiktligt

May also read:

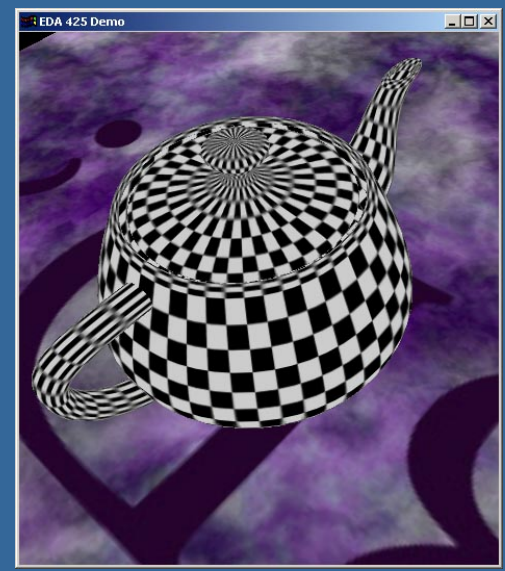
- Angel, chapter 8

Texturing: Glue n-dimensional images onto geometrical objects

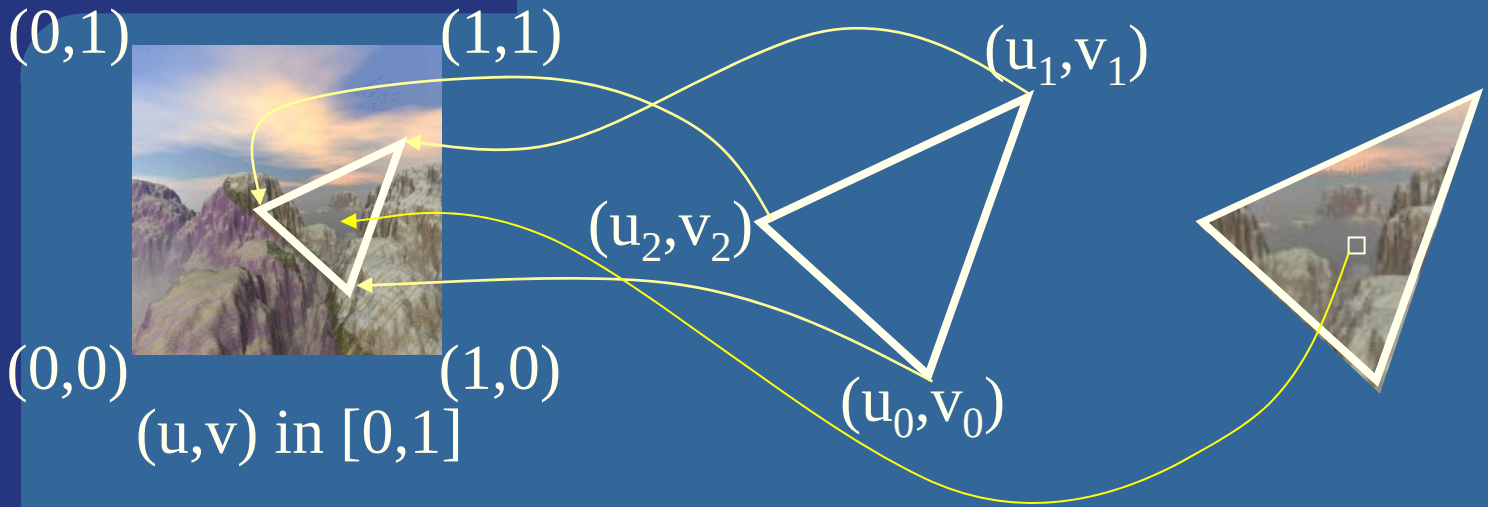
- Purpose: more realism, and this is a cheap way to do it
  - Bump mapping
  - Plus, we can do environment mapping
  - And other things

+

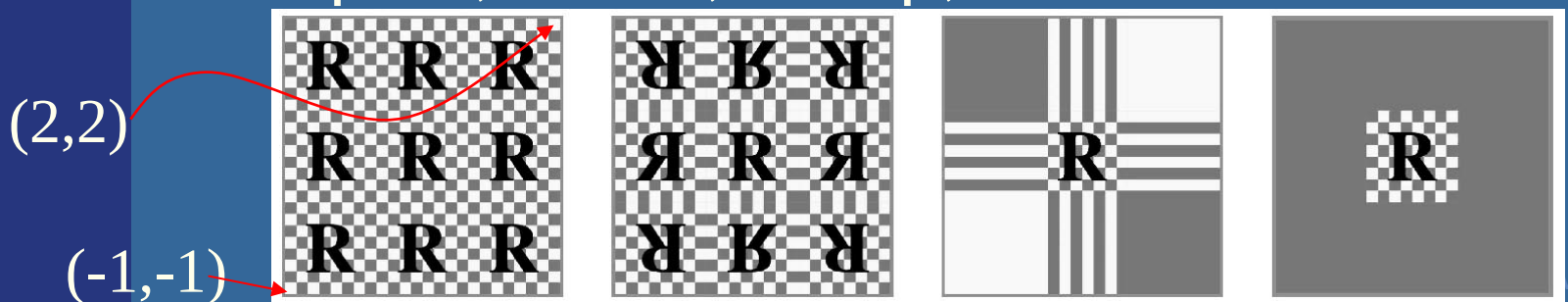
=



Texture coordinates

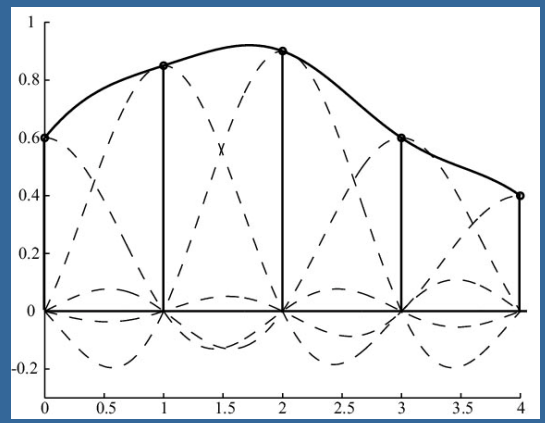


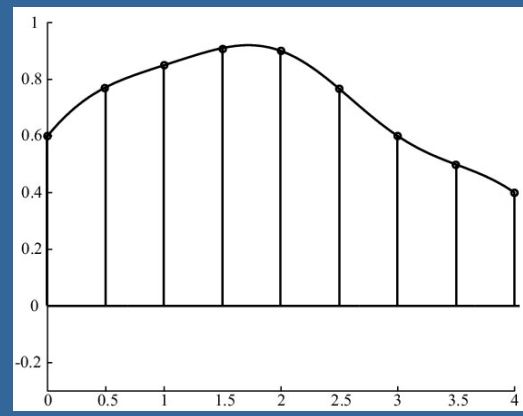
- What if (u,v) >1.0 or <0.0 ?
- To repeat textures, use just the fractional part
  - Example: 5.3 -> 0.3
- Repeat, mirror, clamp, border:



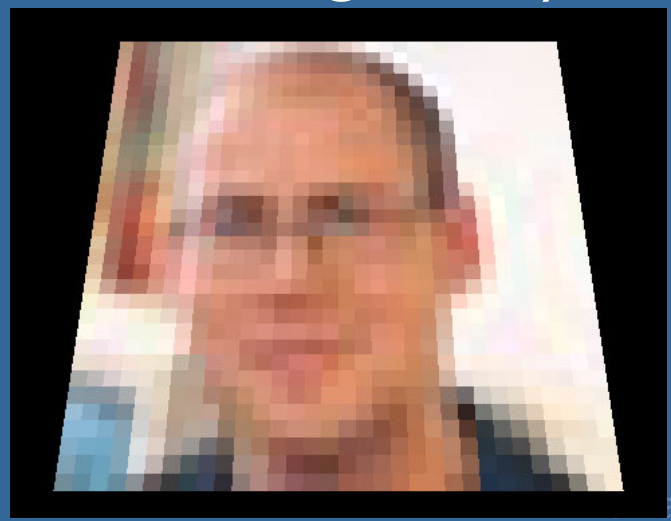
Texture magnification

- What does the theory say…



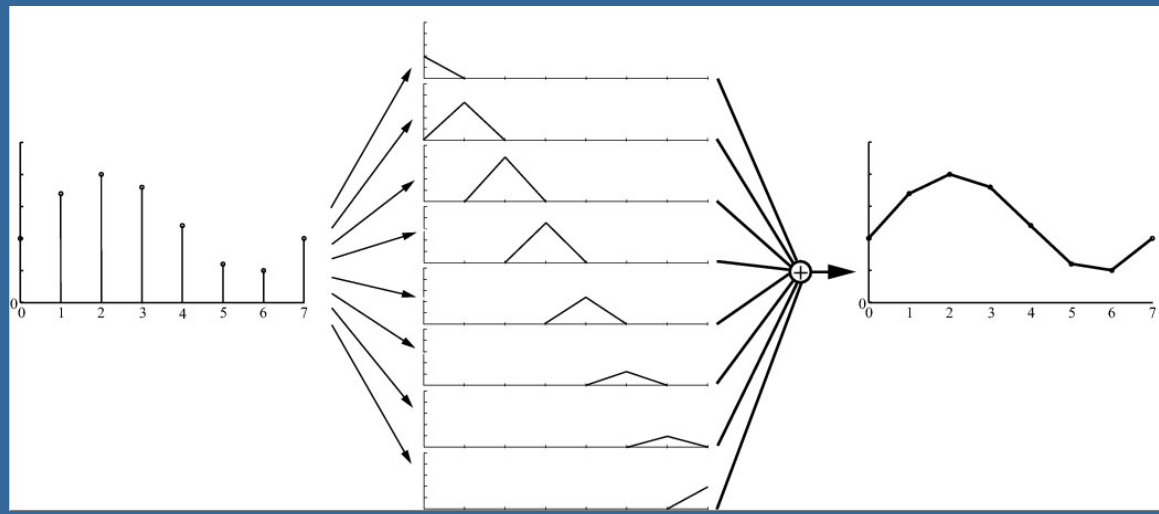


- $\mathrm{sinc}(x)$ is not feasible in real time
- Box filter (nearest-neighbor) is
- Poor quality

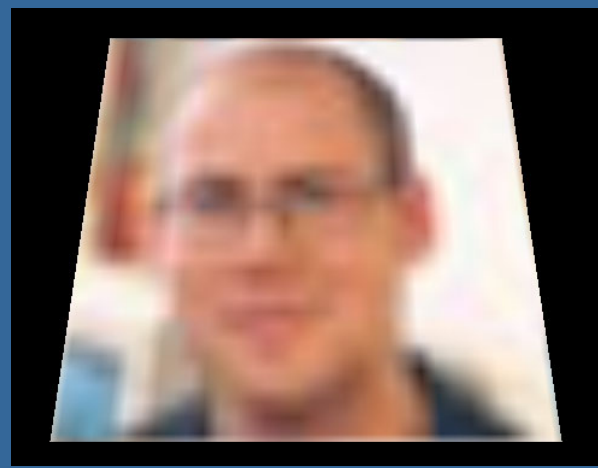


Texture magnification

- Tent filter is feasible!
- Linear interpolation

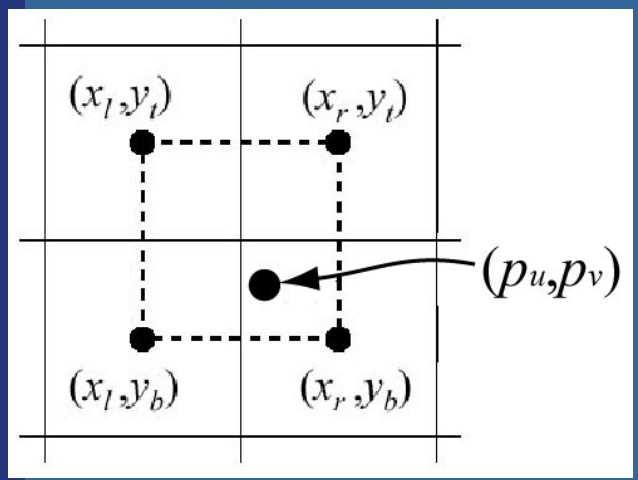


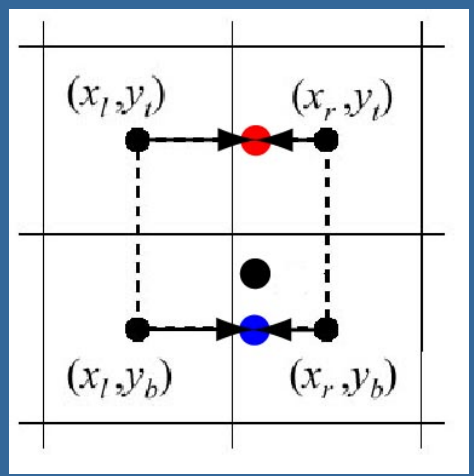
- Looks better
- Simple in 1D:
- (1-t)*color0+t*color1
- How about 2D?

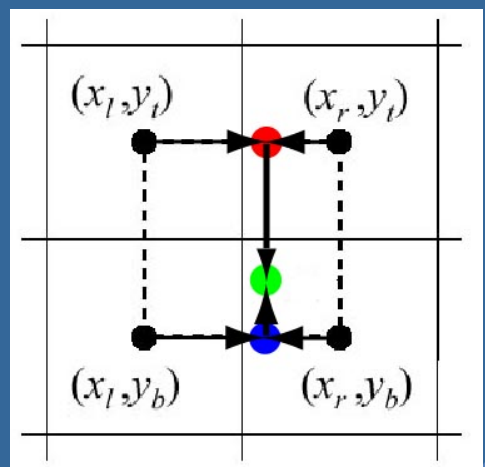


Bilinear interpolation

- Texture coordinates (p_u,p_v) in [0,1]
- Texture images size: n*m texels
- Nearest neighbor would access:
  (floor(n*u), floor(m*v))
- Interpolate 1D in x & y respectively







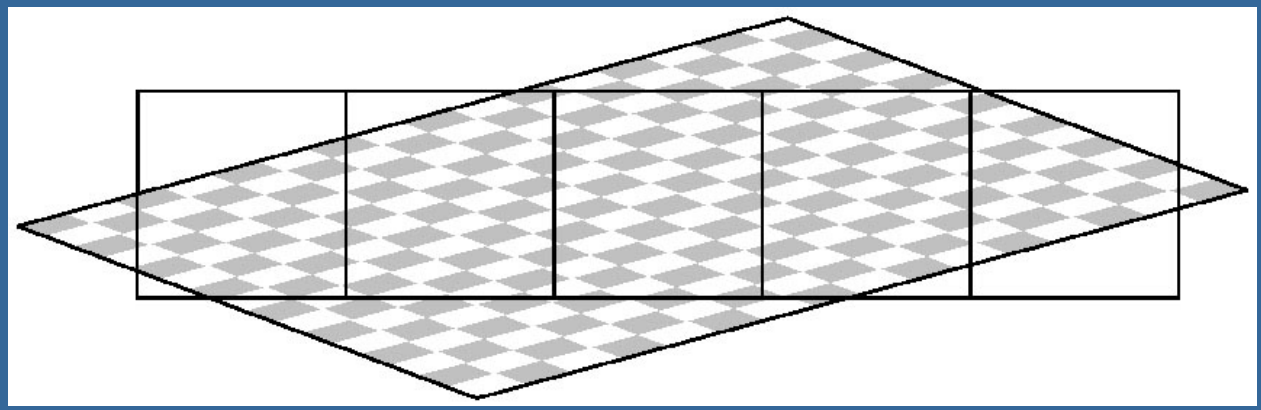
Bilinear interpolation

- Check out this formula at home
- **t**(*u*,*v*) accesses the texture map
- **b**(*u*,*v*) filtered texel

$$(u', v') = (p_u - \lfloor p_u \rfloor, p_v - \lfloor p_v \rfloor).$$

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

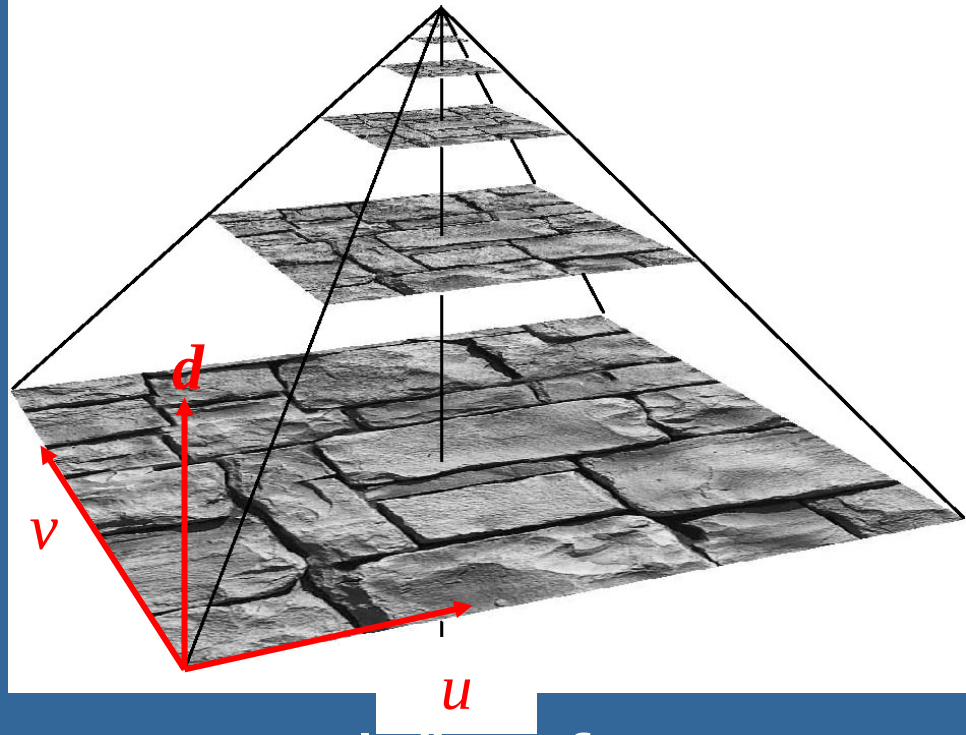
Texture minification
What does a pixel ”see”?



- Theory (sinc) is too expensive
- Cheaper: average of texel inside a pixel
- Still too expensive, actually

- Mipmaps – another level of approximation
  - Prefilter texture maps as shown on next slide

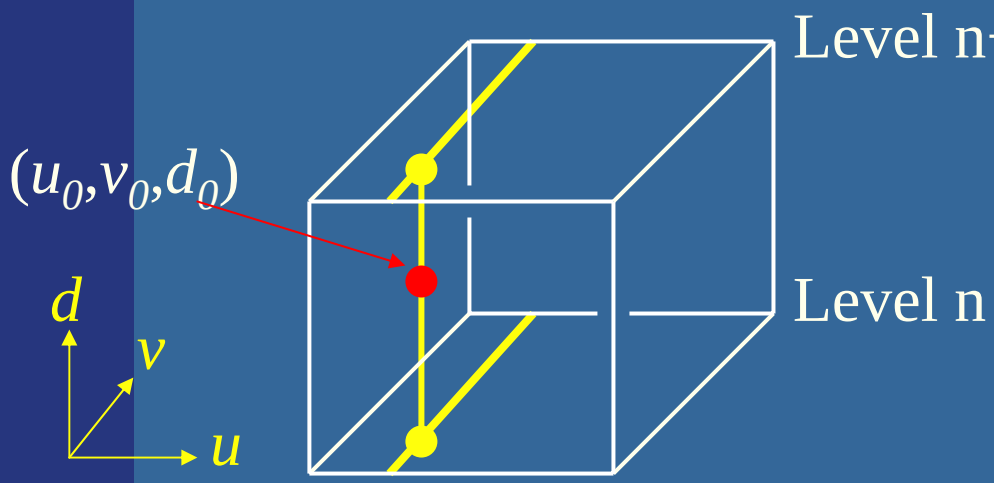
Mipmapping



- Image pyramid
- Half width and height when going upwards
- Average over 4 ”parent texels” to form ”child texel”
- Depending on amount of minification, determine which image to fetch from
- Compute *d* first, gives two images
  - Bilinear interpolation in each

Mipmapping

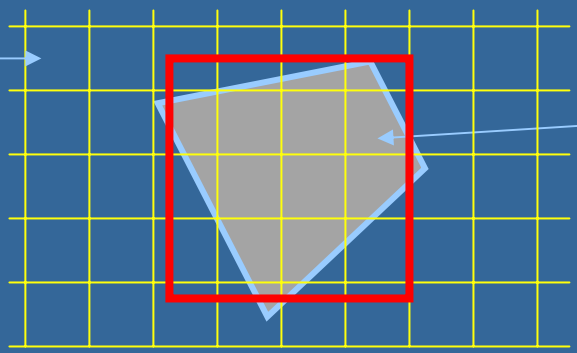
- Interpolate between those bilinear values
  - Gives trilinear interpolation



- Constant time filtering: 8 texel accesses

- How to compute d?

Computing *d* for mipmapping



$$A = \text{approximative area of quadrilateral}$$

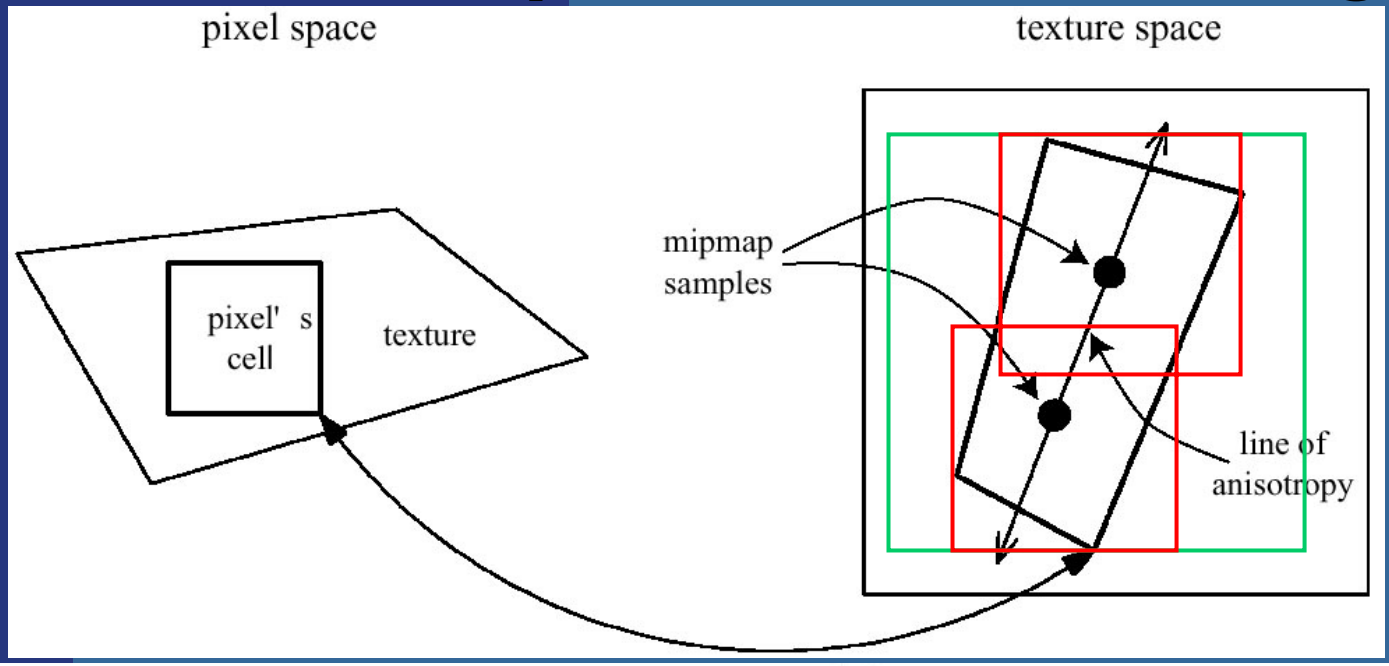
$$b = \sqrt{A}$$

$$d = \log_2 b$$

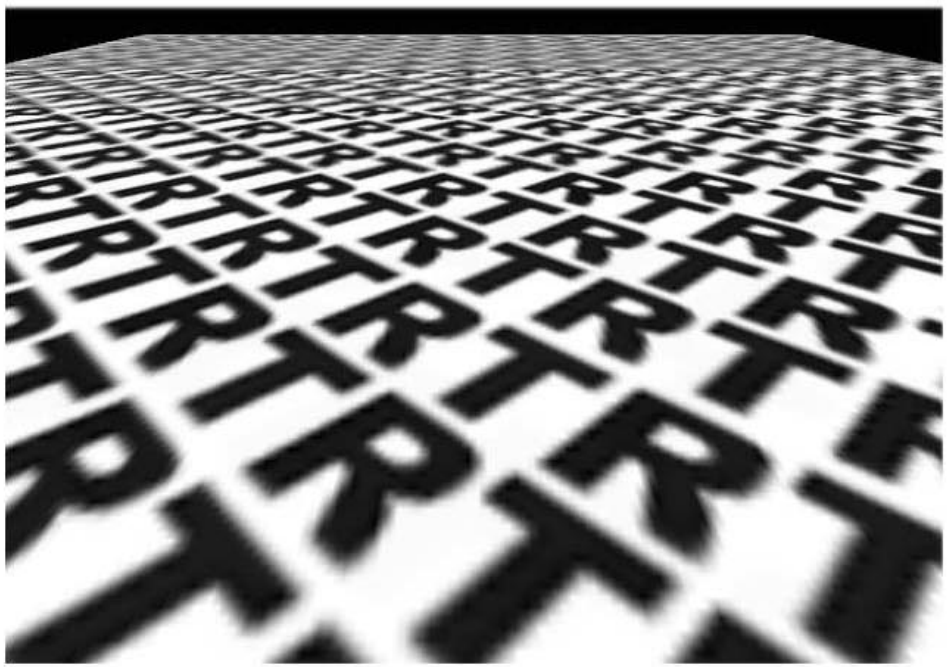
- Approximate quad with square
- Gives overblur!

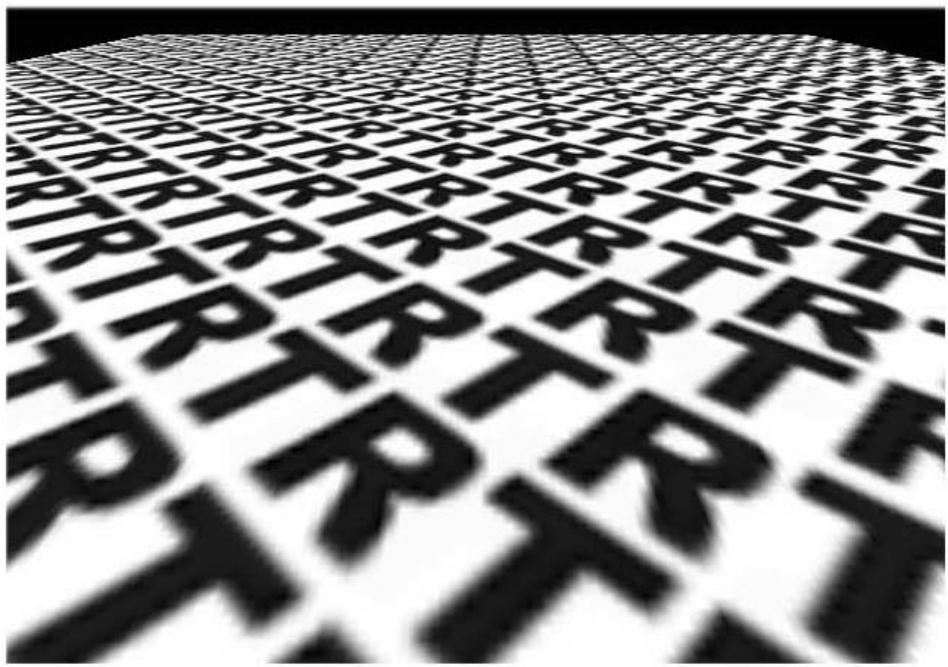
- Even better: anisotropic texture filtering
  - Approximate quad with several smaller mipmap samples

Anisotropic texture filtering



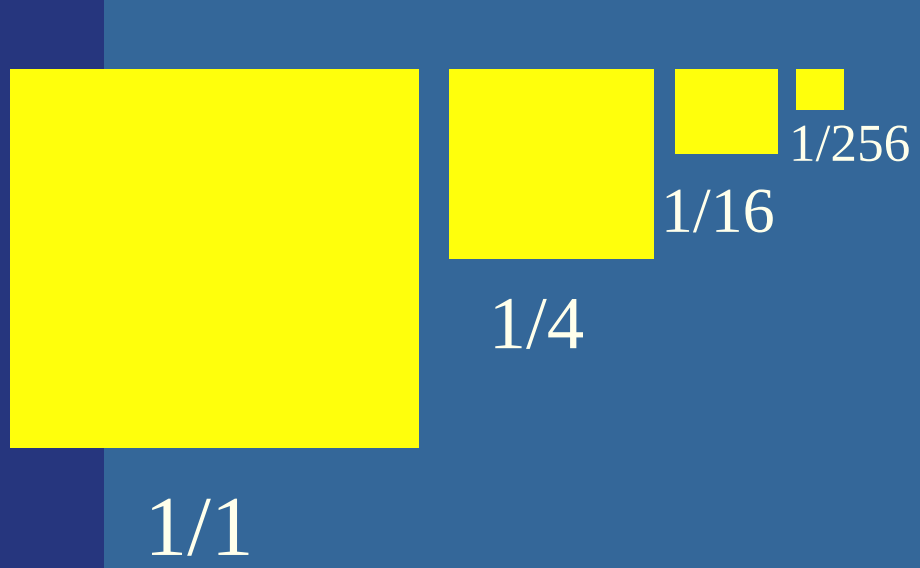
16 samples

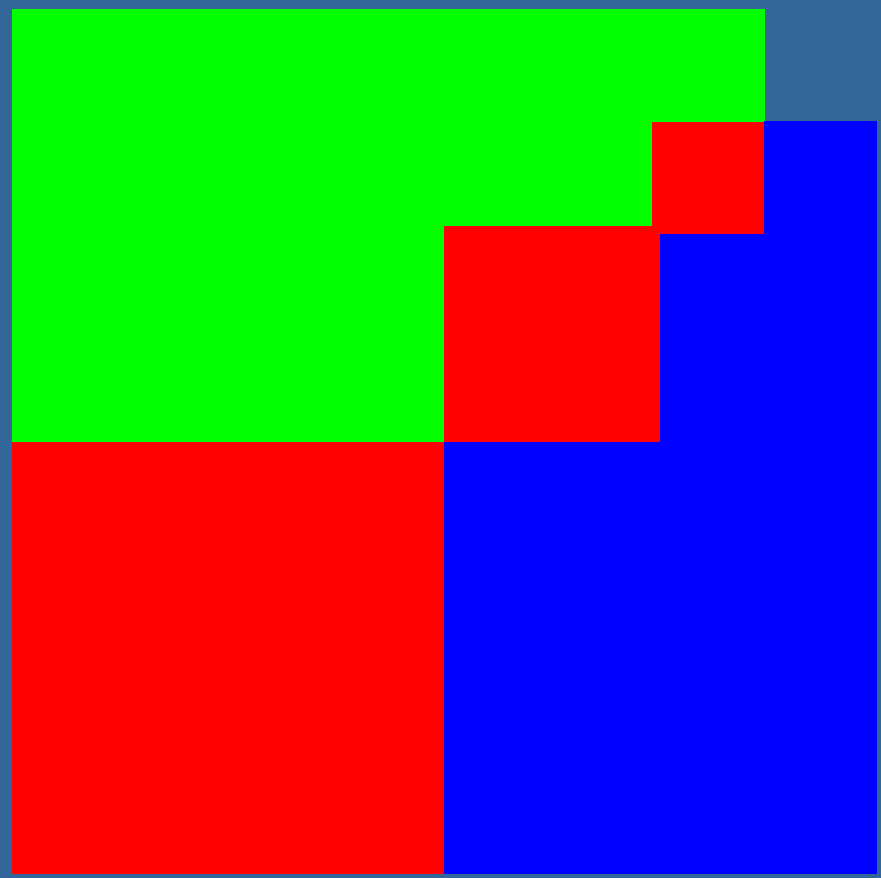




Mipmapping: Memory requirements

- Not twice the number of bytes…!



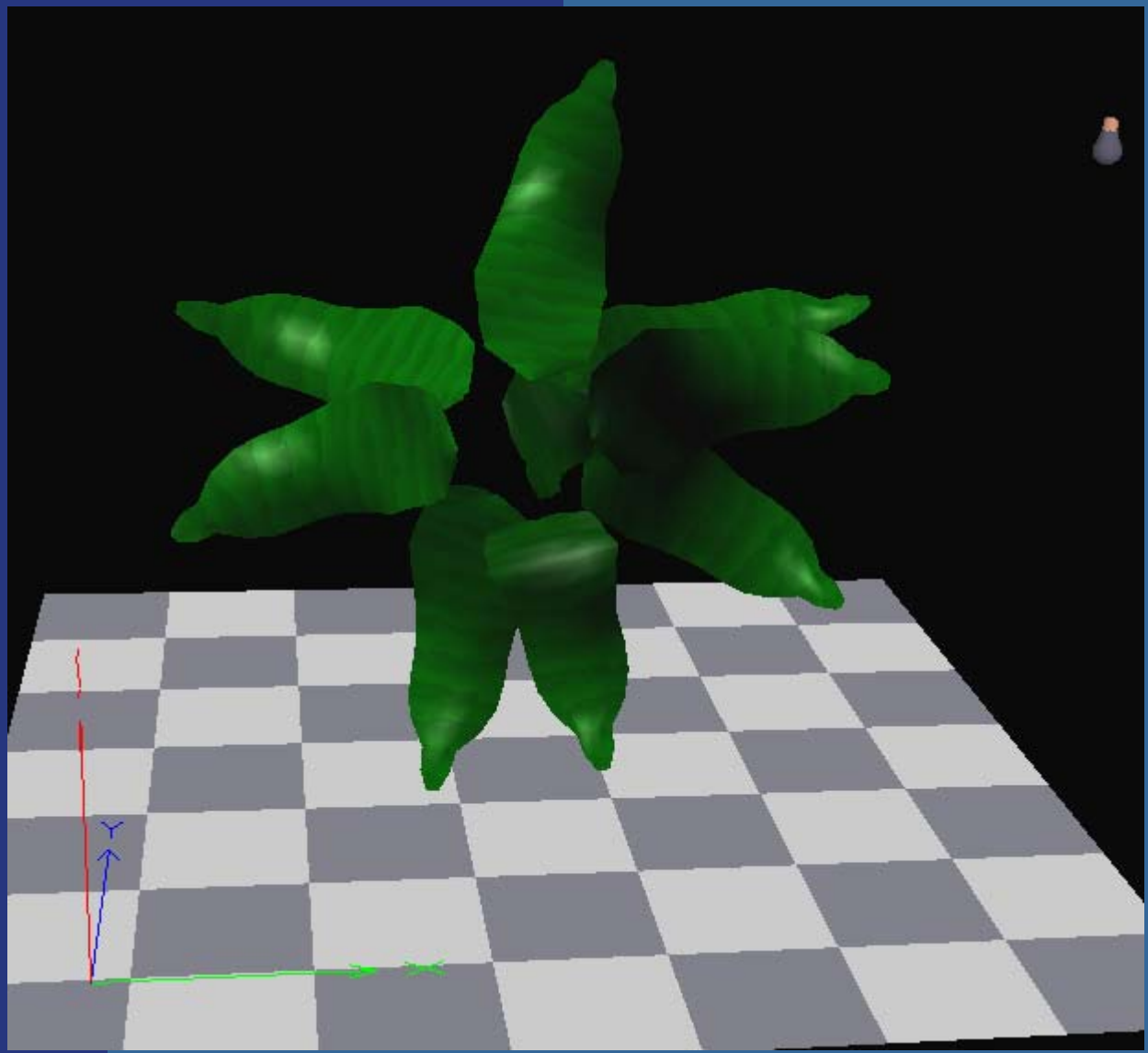


- Rather 33% more – not that much

Miscellaneous

- How to apply texturing:
  - Modulate (multiply texture with lighting)
  - Replace (just use texture color)
  - Add, sub, etc (on newer hardware)
  - More in the manuals, e.g. www.msdn.com

Modulate



Using textures in OpenGL

Do once when loading texture:

```
GLuint texID;
glGenTextures(1, &texID);
glBindTexture(GL_TEXTURE_2D, texID);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
        GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
        GL_LINEAR_MIPMAP_LINEAR); // this is the nicest available mipmap

// specify whether to modulate color from lighting or replacing color, by setting
// GL_MODULATE or GL_DECAL. See OH 152 and MSDN-help for more options
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);

// Using glu to autogenerate mipmaps  (image is pointer to raw rgb-data)
gluBuild2DMipmaps(GL_TEXTURE_2D, 3, width,
                  height, GL_RGB, GL_UNSIGNED_BYTE, image);
```

Do every time you want to use this texture when drawing:

```
glBindTexture(GL_TEXTURE_2D, texID);
glEnable(GL_TEXTURE_2D);
// Now, draw your triangles with texture coordinates specified
```

Automatic Texture Generation

- Instead of manually specifying texcoords
  - e.g. by using glTexCoord()

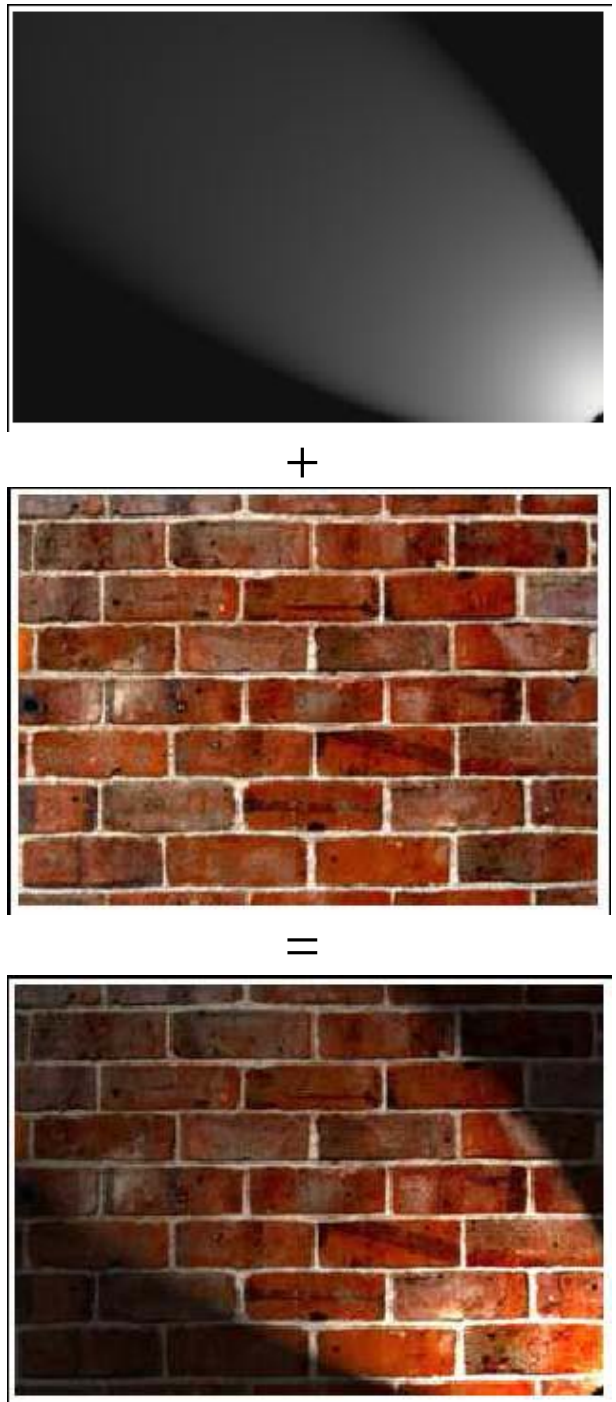
  Do:

```
glTexGeni(  GL_S, GL_TEXTURE_GEN_MODE,    GL_OBJECT_LINEAR)
            GL_T,                         GL_EYE_LINEAR
            GL_R,                         GL_SPHERE_MAP
                                          GL_NORMAL_MAP_ARB
                                          GL_REFLECTION_MAP_ARB
```

- Particularly useful for sphere mapping

Light Maps

- Often used in games
- Can use multitexturing, or
  - render wall using brick texture
  - render wall using light texture and blending to the frame buffer

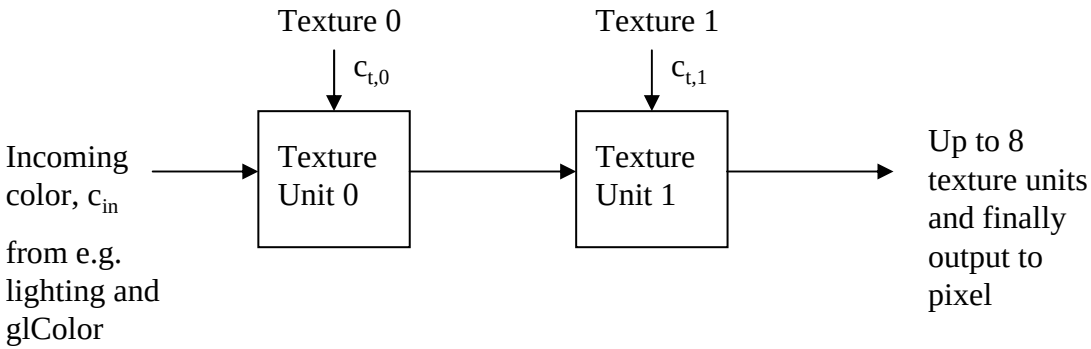


Light Maps – example code

```
// Först ritar vi kvadraten med tegel utan ljus
// Talet 5 ger repetition av texturen 5 ggr över ytan
glBindTexture(GL_TEXTURE_2D, texName[tegelnr]);
glColor3f(1.0, 1.0, 1.0); //Bara om vi använder GL_MODULATE,
//lägrevärden skulle förmörka
glBegin(GL_QUADS);
	glTexCoord2f(0, 0); glVertex2f(-1, -1);
	glTexCoord2f(5, 0); glVertex2f(1, -1);
	glTexCoord2f(5, 5); glVertex2f(1, 1);
	glTexCoordf(0, 5); glVertex2f(-1, 1);
glEnd();
// Sedan är det dags att byta till ljuskartan och se till att
// värdena blandas på lämpligt sätt med det redan ritade
glEnable(GL_BLEND);
glBlendFunc(GL_ZERO, GL_SRC_COLOR); // Ger tegel*ljuskarta
glDepthFunc(GL_LEQUAL); // Ser till att det ritas även om
// vi redan ritat på just det avståndet; finns andra sätt
glBindTexture(GL_TEXTURE_2D, texName[ljusnr]);

//Och så ritar vi kvadraten med enbart ljuskartan
glColor3f(1.0, 1.0, 1.0); // Onödig upprepning
glBegin(GL_QUADS);
	glTexCoord2f(0, 0); glVertex2f(-1, -1);
	glTexCoord2f(1, 0); glVertex2f(1, -1);
	glTexCoord2f(1, 1); glVertex2f(1, 1);
	glTexCoord2f(0, 1); glVertex2f(-1, 1);
glEnd();
```

Multitextures

- OpenGL 1.3 (aug 2001)



$c_{out,0} = c_{in} * c_{t,0}$

for GL_MODULATE

Light Maps – multitexture version

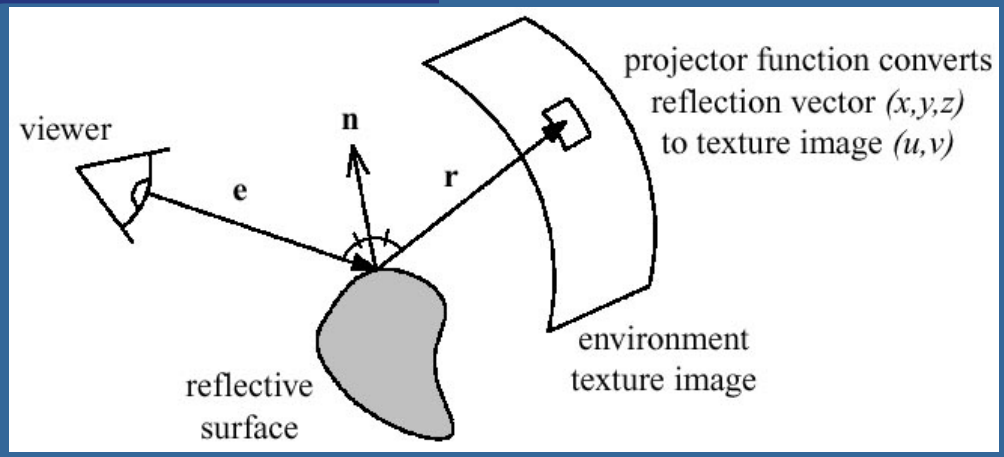
```
void wall() {
    glBegin(GL_QUADS);
    // Första hörnet, ett texturkoordinatpar för tegelväggen (texturenhet 0),
    // ett annat för ljuskartan (texturenhet 1)
        glMultiTexCoord2fARB(GL_TEXTURE0_ARB,0, 0);
        glMultiTexCoord2fARB(GL_TEXTURE1_ARB,0, 0);
        glVertex2f(-1, -1);
    // Övriga hörn. Siffran 5 ger upprepning av tegelmönstret
        glMultiTexCoord2fARB(GL_TEXTURE0_ARB,5, 0);
        glMultiTexCoord2fARB(GL_TEXTURE1_ARB,1, 0);
        glVertex2f(1, -1);
        glMultiTexCoord2fARB(GL_TEXTURE0_ARB,5, 5);
        glMultiTexCoord2fARB(GL_TEXTURE1_ARB,1, 1);
        glVertex2f(1, 1);
        glMultiTexCoord2fARB(GL_TEXTURE0_ARB,0, 5);
        glMultiTexCoord2fARB(GL_TEXTURE1_ARB,0, 1);
        glVertex2f(-1, 1);
    glEnd();
}
```

Light Maps – multitexture version

Och i omritningsproceduren display():

```
// Med glActiveTexture bestämmer vi vilken texturenhet
// vars tillstånd skall förändras
// Först texturenhet 0 som får ha hand om tegelmönstret
glActiveTextureARB(GL_TEXTURE0_ARB);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
glBindTexture(GL_TEXTURE_2D, texName[tegelnummer]);
glEnable(GL_TEXTURE_2D);
// Sedan texturenhet 1 som hanterar ljuskartan
glActiveTextureARB(GL_TEXTURE1_ARB);
glTexEnvi(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_MODULATE);
glBindTexture(GL_TEXTURE_2D, texName[ljusnr]);
glEnable(GL_TEXTURE_2D);

// vi slänger in exempel på hur man kan använda texturmatrisen för att translatera texturen
glMatrixMode(GL_TEXTURE);
glPushMatrix();
glTranslatef(tx, ty, tz);
wall();
glPopMatrix(); // texturmatrisen
glDisable(GL_TEXTURE_2D);
glActiveTextureARB(GL_TEXTURE0_ARB);
glDisable(GL_TEXTURE_2D);
```

Environment mapping



- Assumes the environment is infinitely far away
- Sphere mapping
  - For details, see OH 166-169
- Cube mapping is the norm nowadays
  - Advantages: no singularities as in sphere map
  - Much less distortion
  - Gives better result
  - Not dependent on a view position

Sphere map

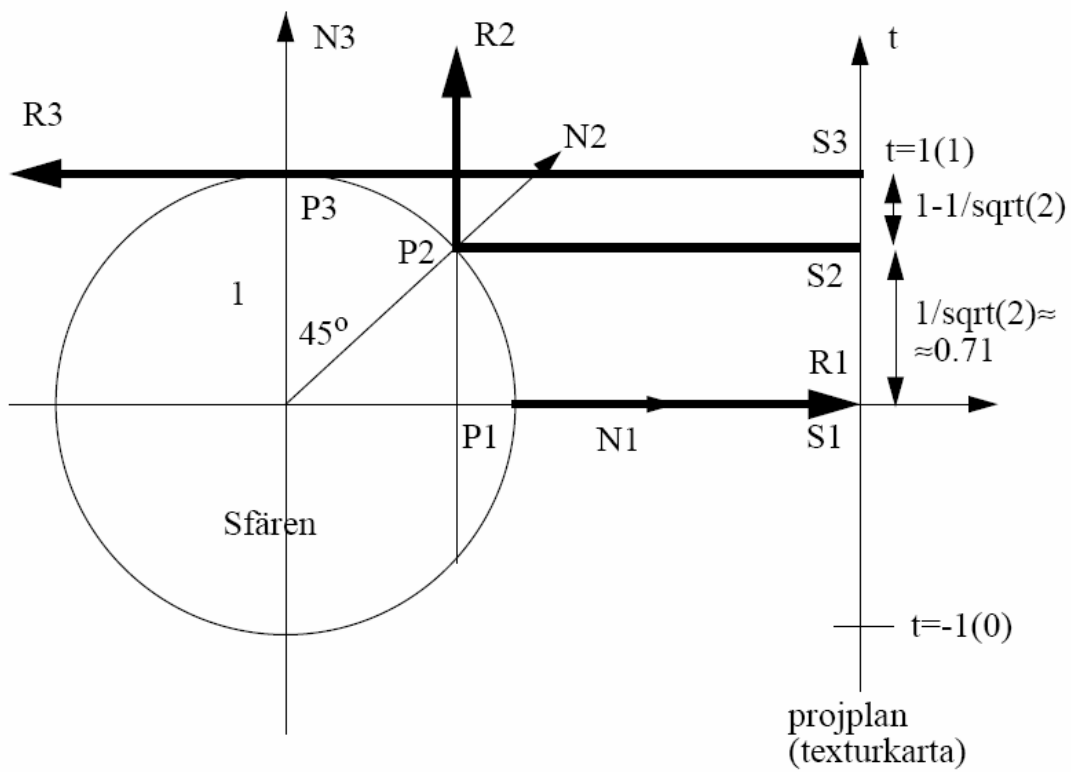
- example

Sphere map
(texture)

Sphere map
applied on torus

Sphere Map

- Infinitesimally small reflective sphere (infinitely far away)
  - i.e., orthographic view of a reflective unit sphere
- Create by:
  - Photographing metal sphere
  - Ray tracing
  - Transforming cube map to sphere map



Sphere Map

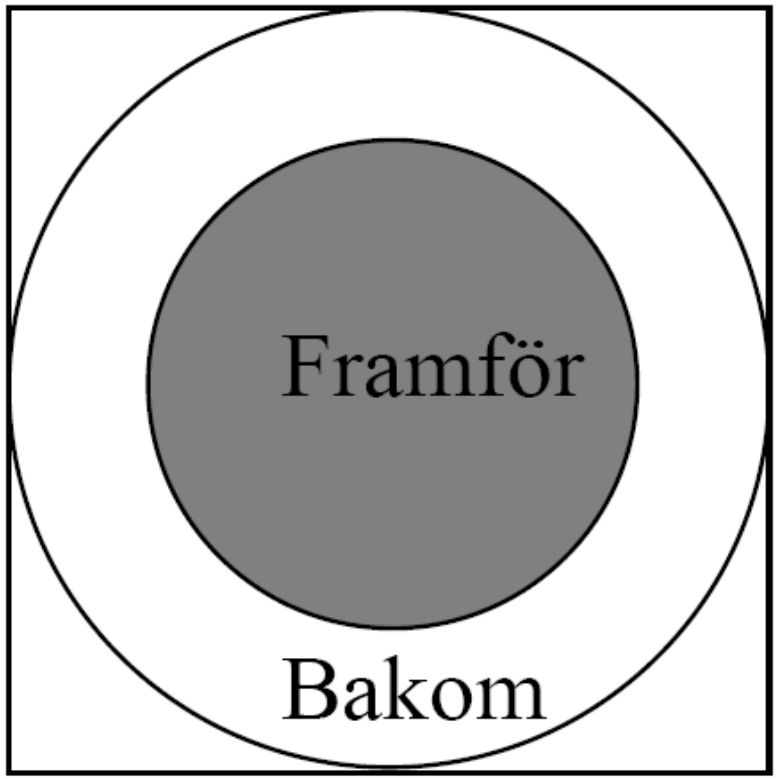
- Assume surface normals are available
- Then OpenGL can compute reflection vector at each pixel
- The texture coordinates s,t are given by:
  - (see OH 169 for details)

$$L = \sqrt{R_x{}^2 + R_y{}^2 + (R_z + 1)^2}$$

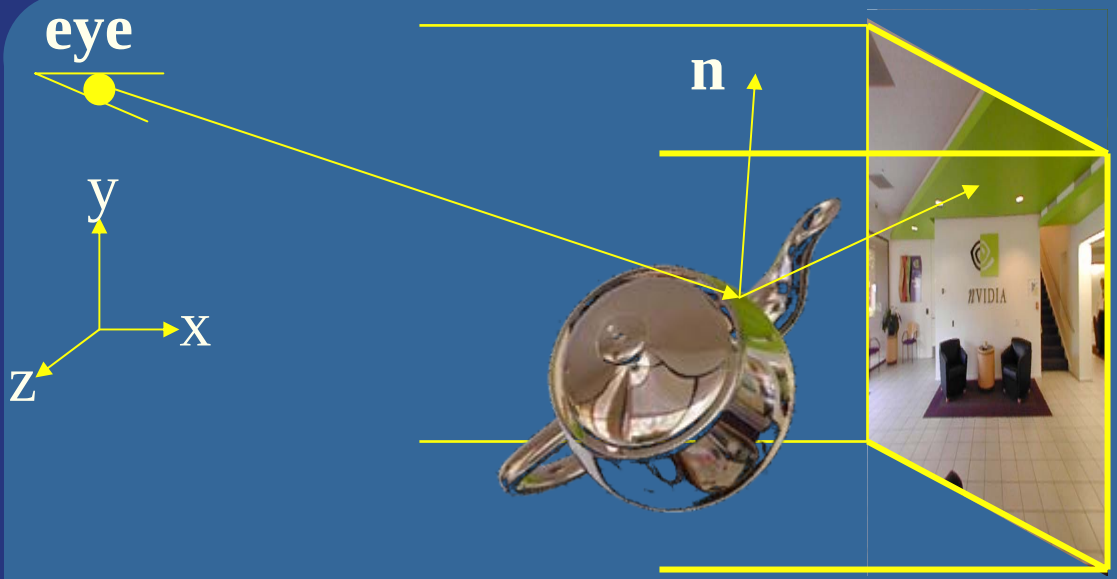
$$s = 0.5\left(\frac{R_x}{L} + 1\right)$$

$$t = 0.5\left(\frac{R_y}{L} + 1\right)$$

Sphere Map

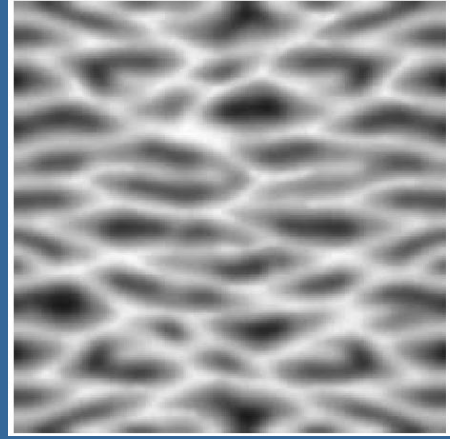


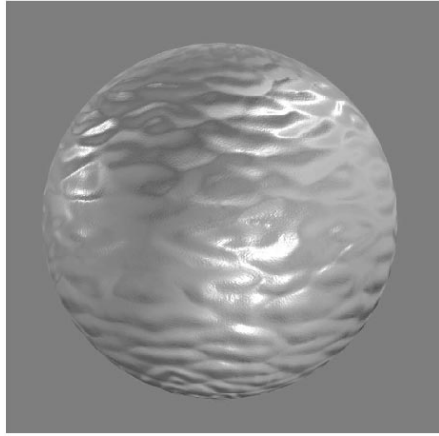
Cube mapping



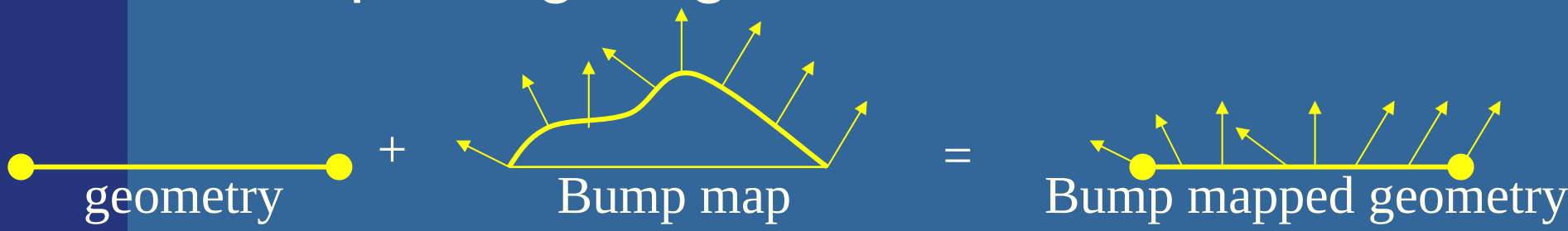
- Simple math: compute reflection vector, **r**
- Largest abs-value of component, determines which cube face.
  - Example: **r**=(5,-1,2) gives POS_X face
- Divide **r** by 5 gives (*u*,*v*)=(-1/5,2/5)
- If your hardware has this feature, then it does all the work

Bump mapping





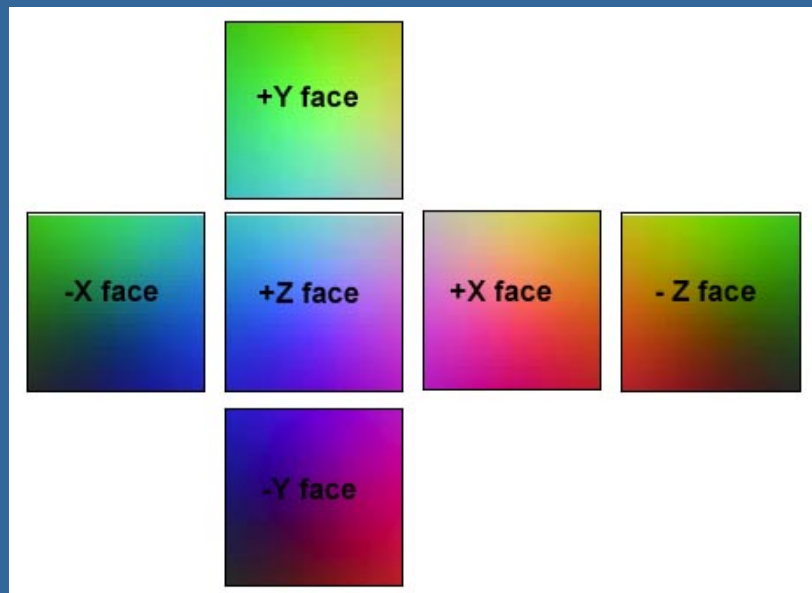
- by Blinn in 1978
- Inexpensive way of simulating wrinkles and bumps on geometry
  - Too expensive to model these geometrically
- Instead let a texture modify the normal at each pixel, and then use this normal to compute lighting



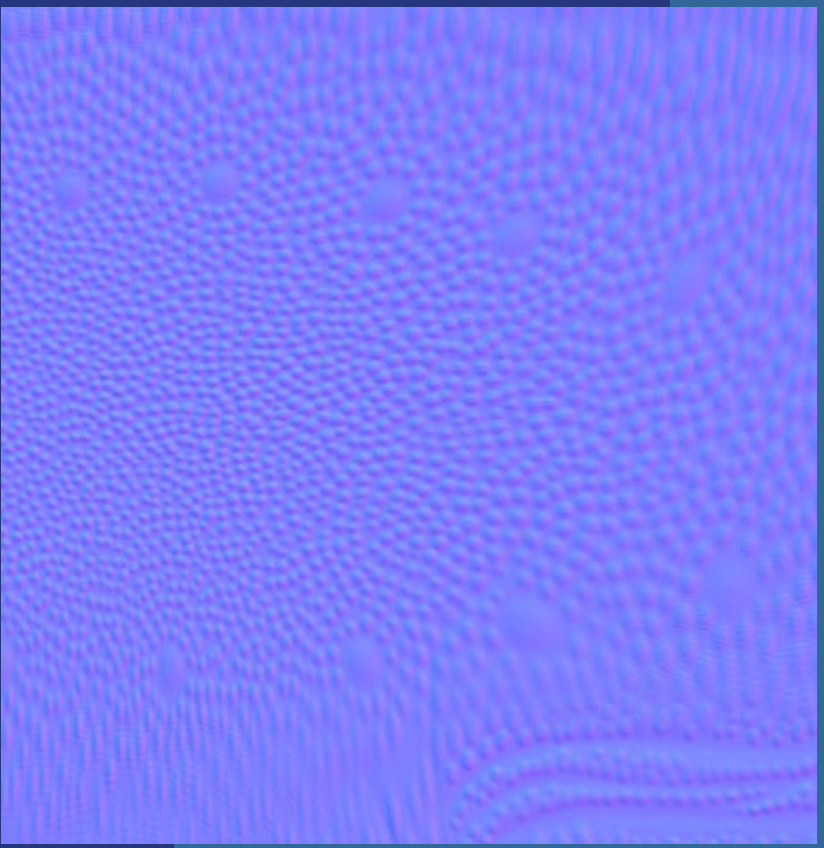
Stores heights: can derive normals

Bump mapping lighting

- How store normals in texture (bump map)
- **n**=(n_x , n_y , n_z) are in [-1,1]
- Add 1, mult 0.5: in [0,1]
- Mult by 255 (8 bit per color component)
- Can be stored in texture:



Bump mapping: example

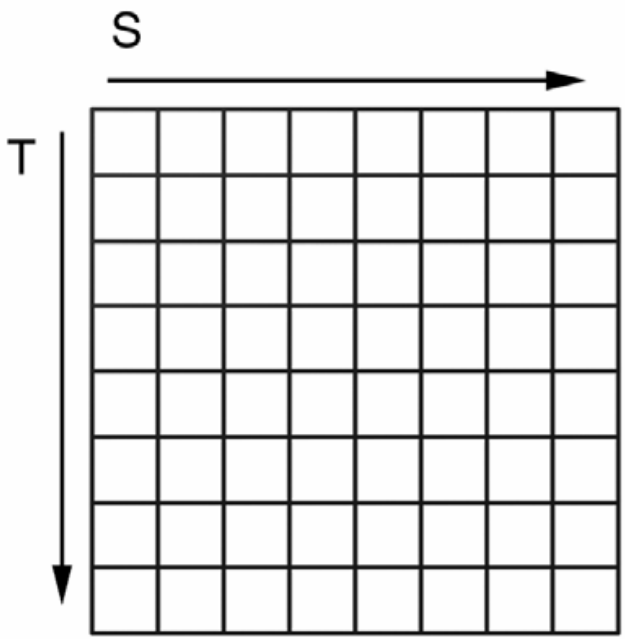


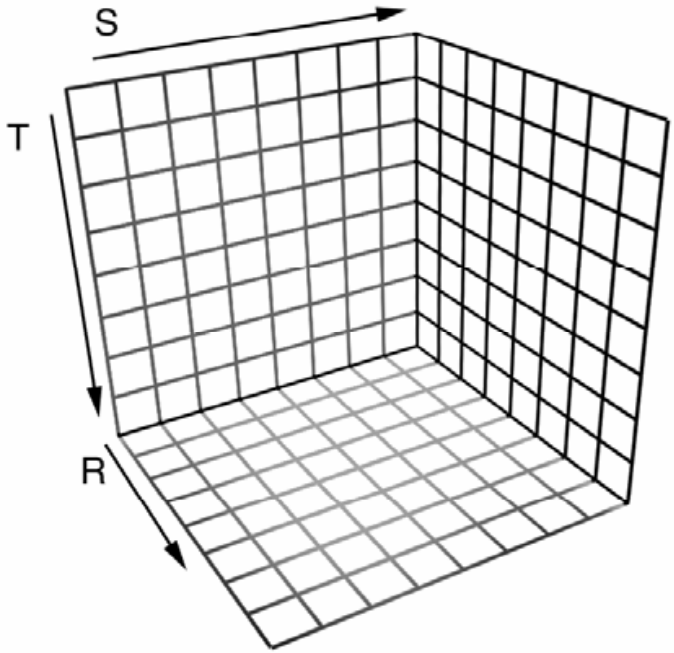
- Other ways to do bump mapping on
  - These are less expensive, and feasible on cheaper hardware

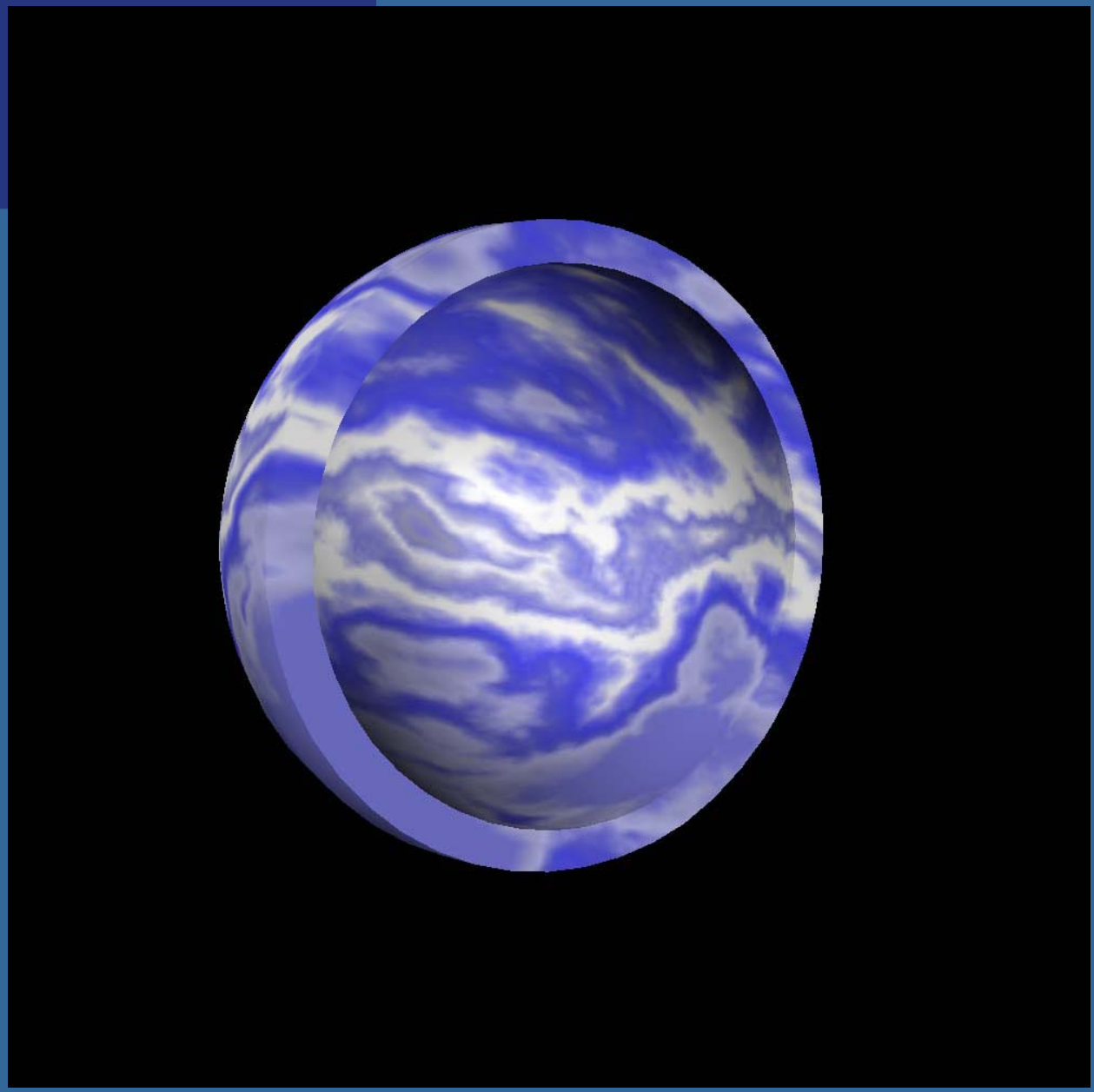
More...

- 3D textures:
  - Feasible on modern hardware as well
  - Texture filtering is no longer trilinear
  - Rather quadlinear (linear interpolation 4 times)
  - Enables new possibilities
    - Can store light in a room, for example
- Displacement Mapping
  - Offsets the position per pixel or per vertex
    - Offsetting per vertex is easy in vertex shader
    - Offsetting per pixel is architecturally hard
      - Cannot easily be done in fragment shader
      - Can be done using Geometry Shader (e.g. Direct3D 10) by ray casting in the displacement map

2D texture vs 3D texture



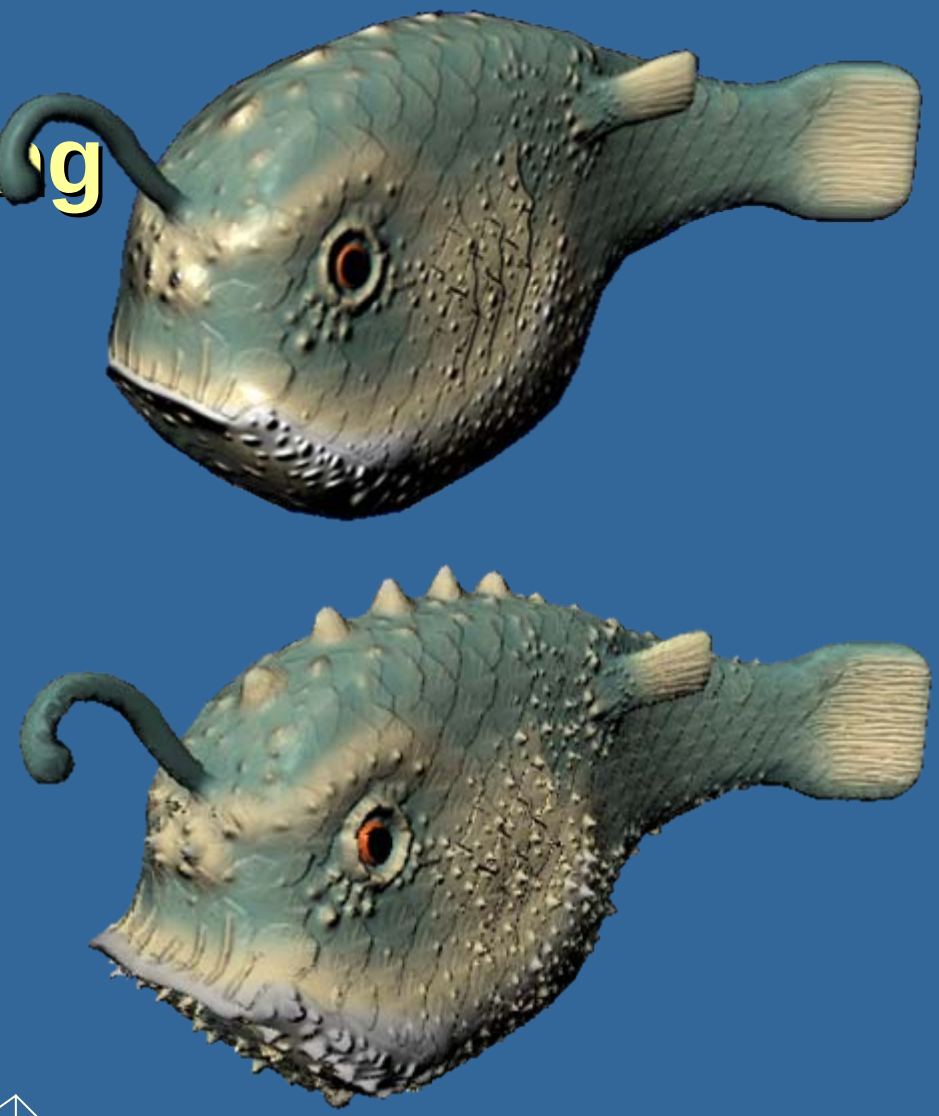


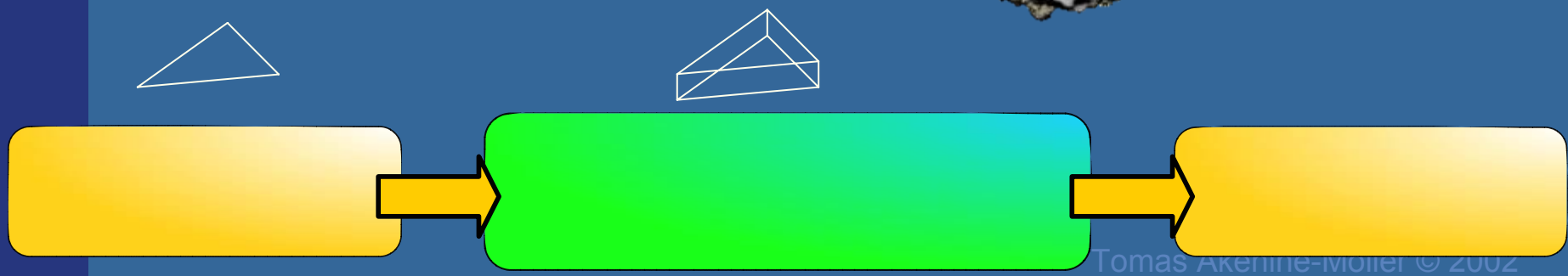


From http://www.ati.com/developer/shaderx/ShaderX_3DTextures.pdf

Displacement Mapping

- Uses a map to displace the surface at each position
- Can be done with a Geometry Shader

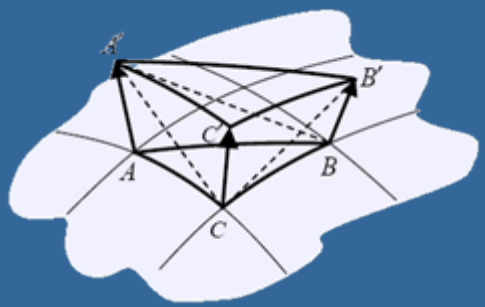




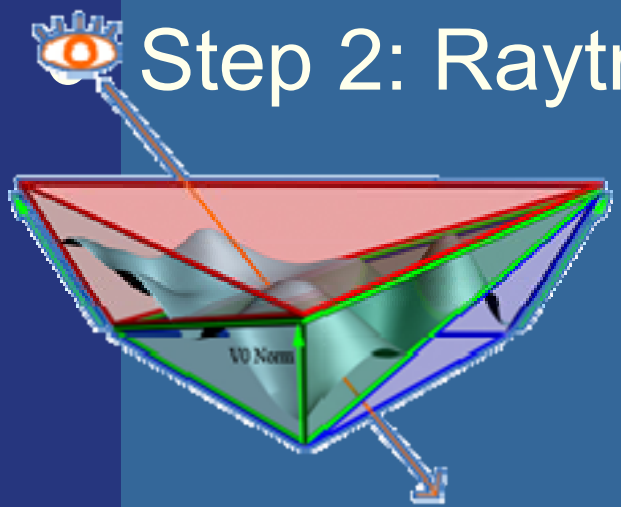
Geometry Shader Example

Generalized Displacement Maps

- Step 0: Process Vertices (VS)
- Step 1: Extrude Prisms (GS)



- Step 2: Raytrace! (PS)

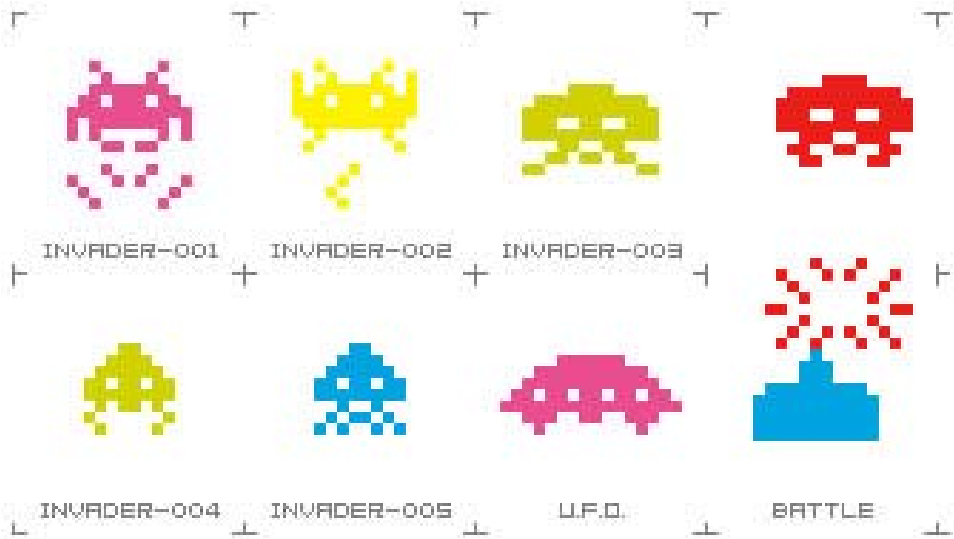


Rendering to Texture

- WGL_ARB_render_to_texture
  - See OH 201
- Many different ways:
  - PBuffers – NVIDIA, Linux and Windows
  - FrameBufferObjects (FBO:s)
    - ARB superbuffers, ARB_uber_buffers
    - Pixel Buffer Objects = e.g. for updating textures
    - Draw Buffers is multiple output buffers, usually 4
- E.g for creating cube maps
  - Direct3D 10 can draw all 6 cube sides in one rendering pass

Sprites

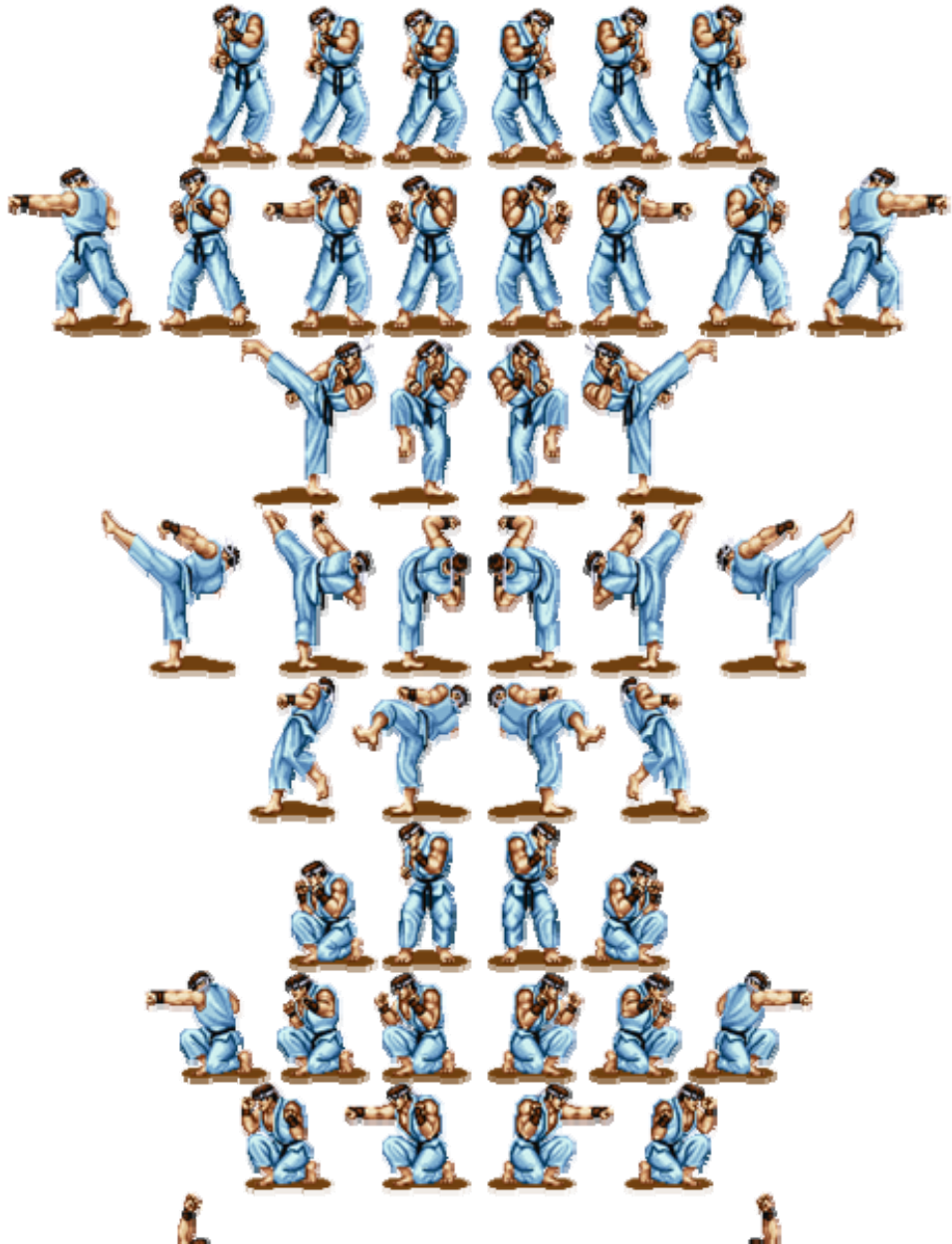
```
GLbyte M[64]=
{   127,0,0,127, 127,0,0,127,
    127,0,0,127, 127,0,0,127,
    0,127,0,0, 0,127,0,127,
    0,127,0,127, 0,127,0,0,
    0,0,127,0, 0,0,127,127,
    0,0,127,127, 0,0,127,0,
    127,127,0,0, 127,127,0,127,
    127,127,0,127, 127,127,0,0};

void display(void) {
    glClearColor(0.0,1.0,1.0,1.0);
    glClear(GL_COLOR_BUFFER_BIT);
    glEnable (GL_BLEND);
    glBlendFunc (GL_SRC_ALPHA,
          GL_ONE_MINUS_SRC_ALPHA);
    glRasterPos2d(xpos1,ypos1);
    glPixelZoom(8.0,8.0);
    glDrawPixels(width,height,
          GL_RGBA, GL_BYTE, M);

    glPixelZoom(1.0,1.0);
    glutSwapBuffers();
}
```



Sprites

Animation Maps

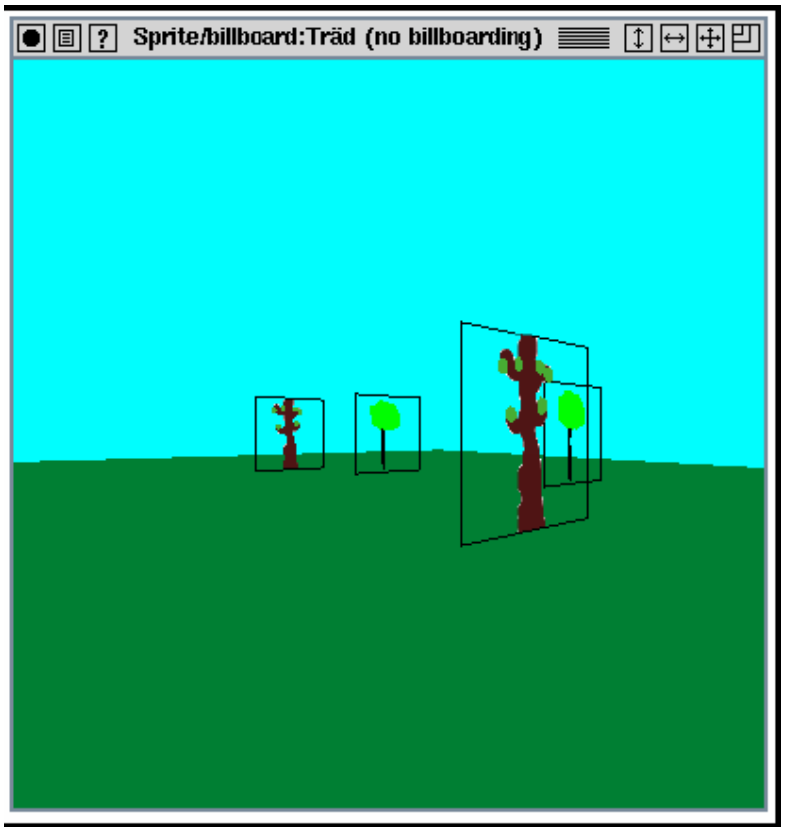
The sprites for Ryu:

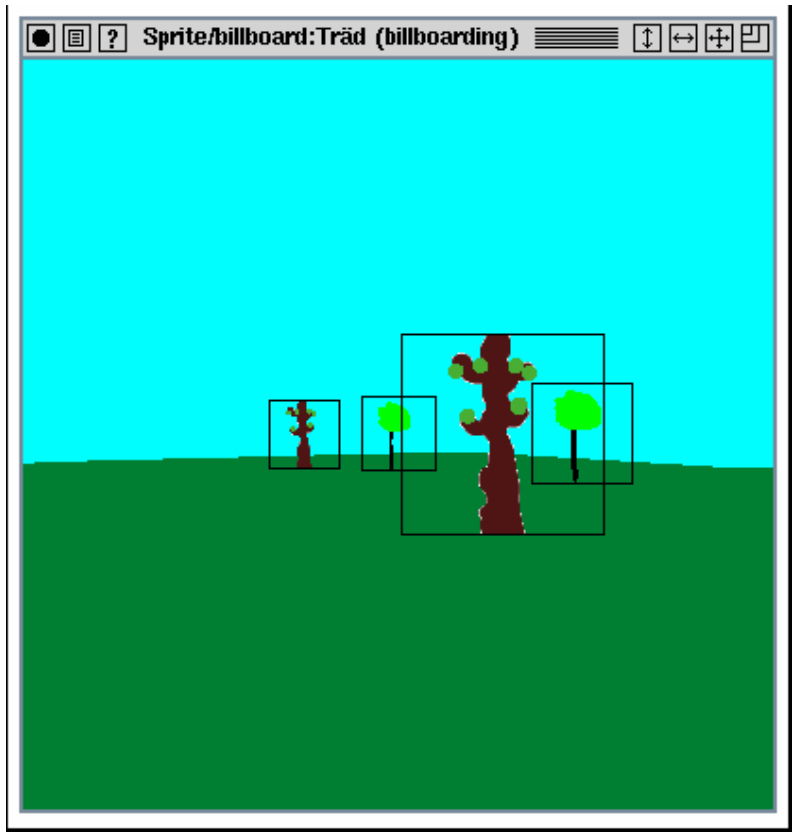


Billboards

- 2D images used in 3D environments
  - Common for trees, explosions, clouds, lens flares

Billboards

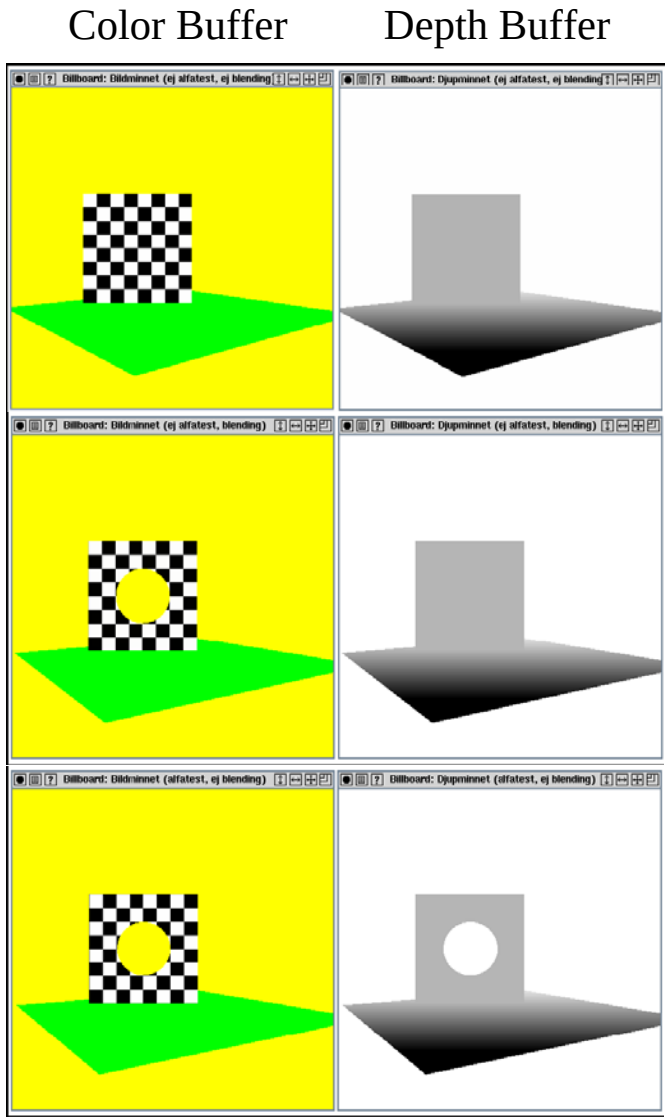




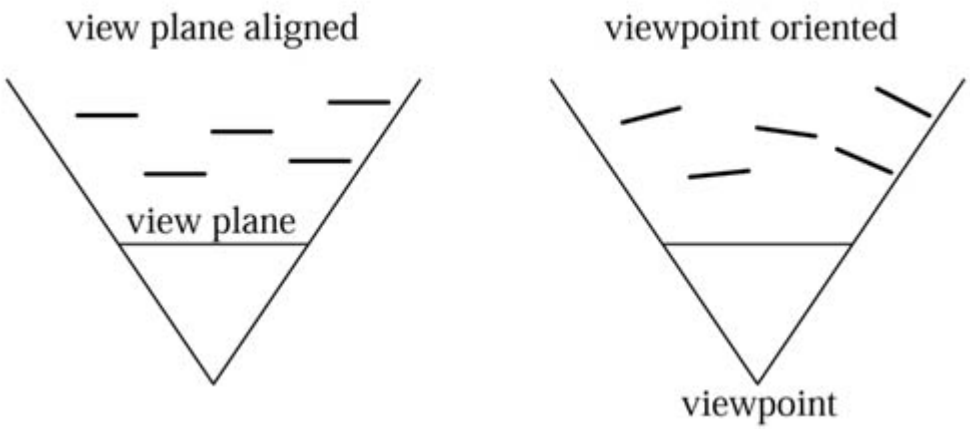
- Rotate them towards viewer
  - Either by rotation matrix (see OH 288), or
  - by orthographic projection

Billboards

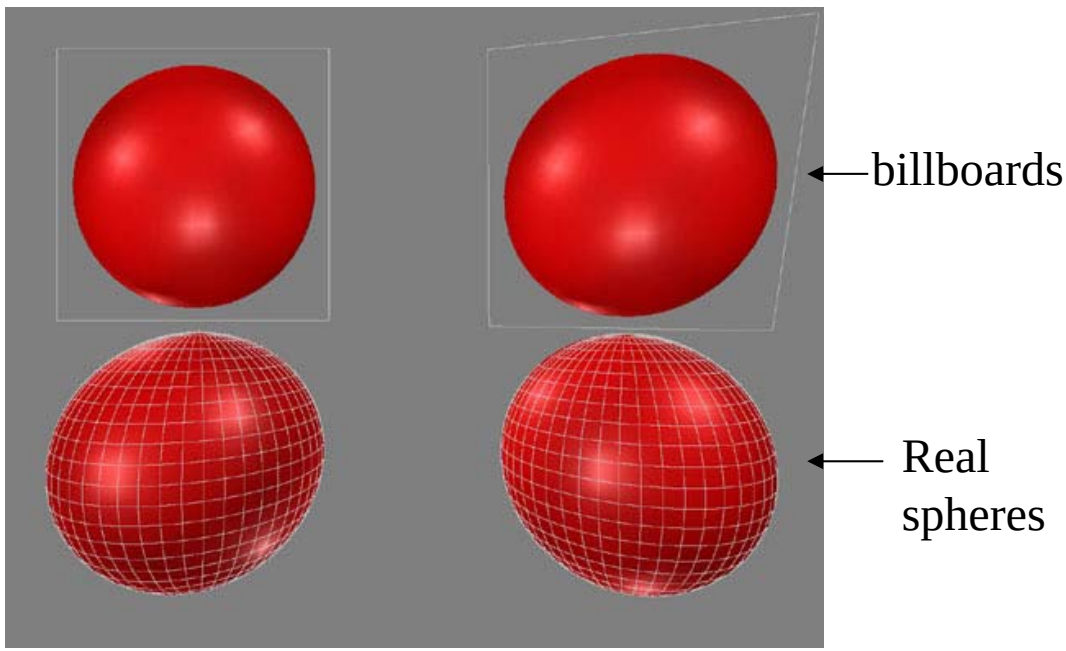
- Fix correct transparency by blending AND using alpha-test
  - glEnable(GL_ALPHA_TEST);
  - glAlphaFunc(GL_GREATER, 0.1);
- Or: sort back-to-front and blend with depth writing disabled
  - glDepthMask(0);
- OH 289-291



Which is preferred?

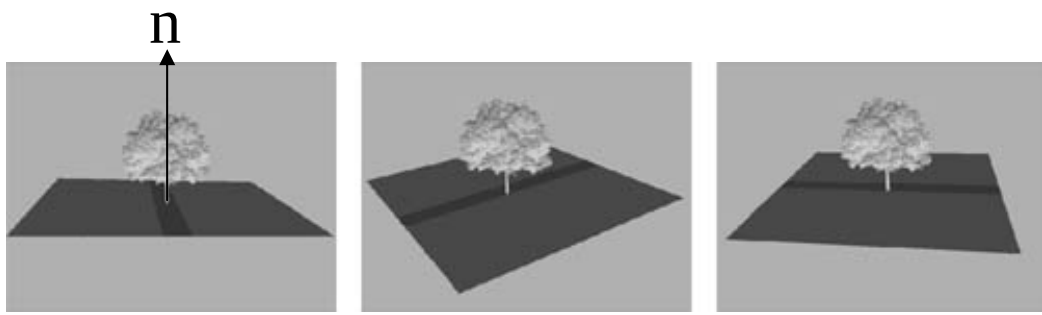


This is the result



Actually, viewpoint oriented is usually preferred since it most closely resembles the result using standard triangulated geometry

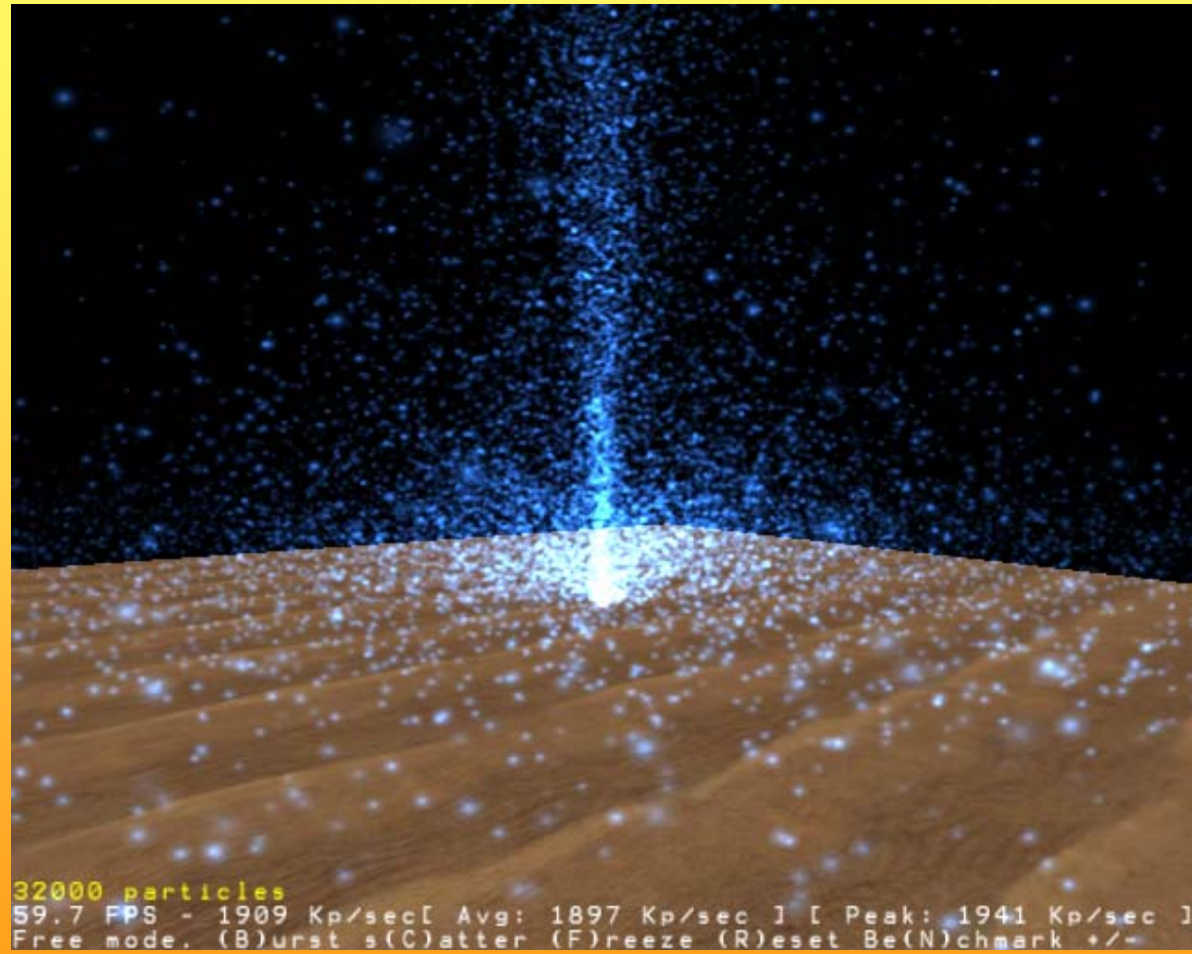
Also called *Impostors*



axial billboarding

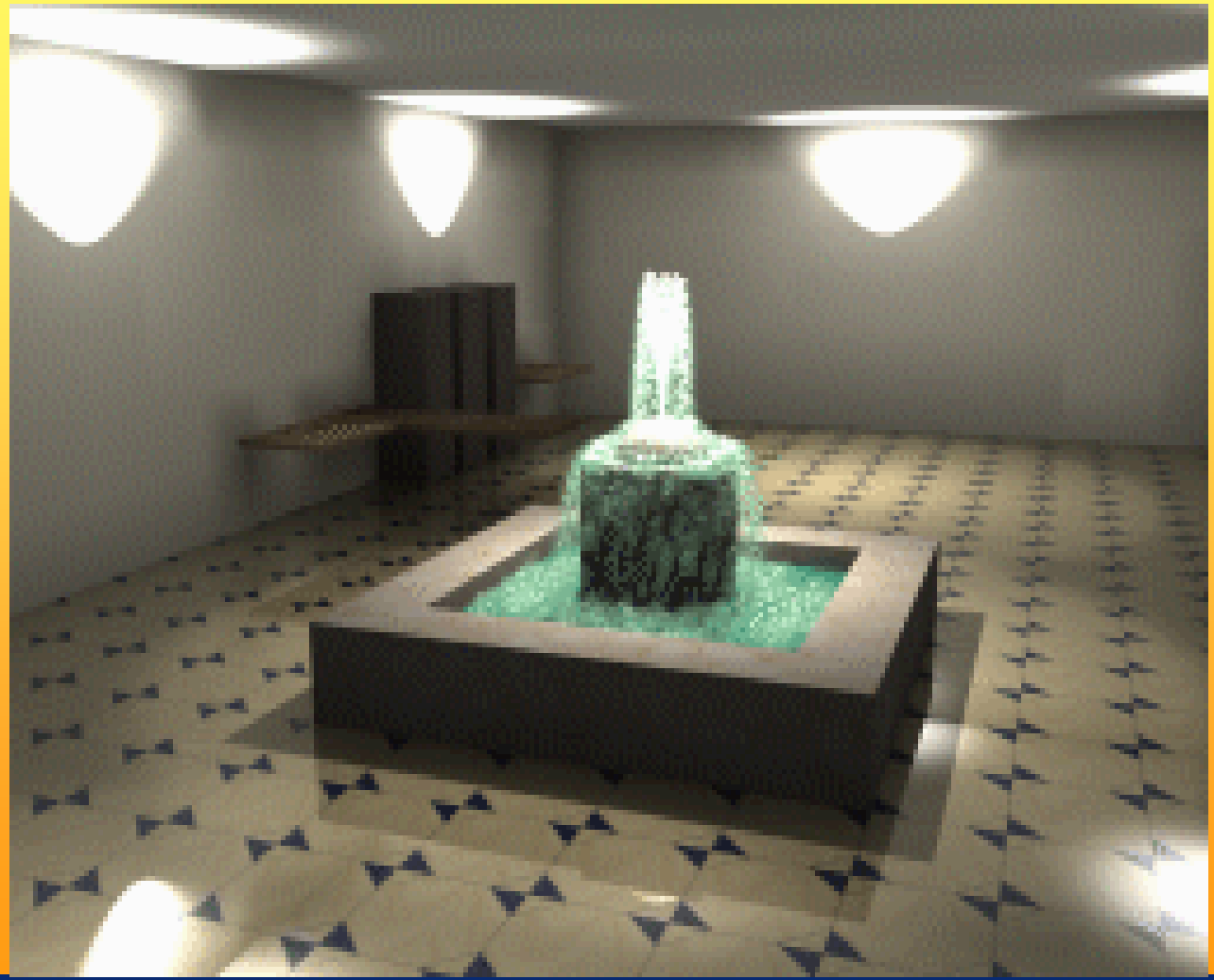
The rotation axis is fixed and disregarding the view position

Partikelsystem



Particles

Partikelsystem



Particle Systems

- Particle Systems: see also OH 228-229
- Boids (flockar), see OH 230
  - 3 rules:
    1. Separation:Avoid obstacles and getting to close to each other
    2. Alignment (strive for same speed and direction as nearby boids
    3. Cohesion: steer towards center of mass of nearby boids