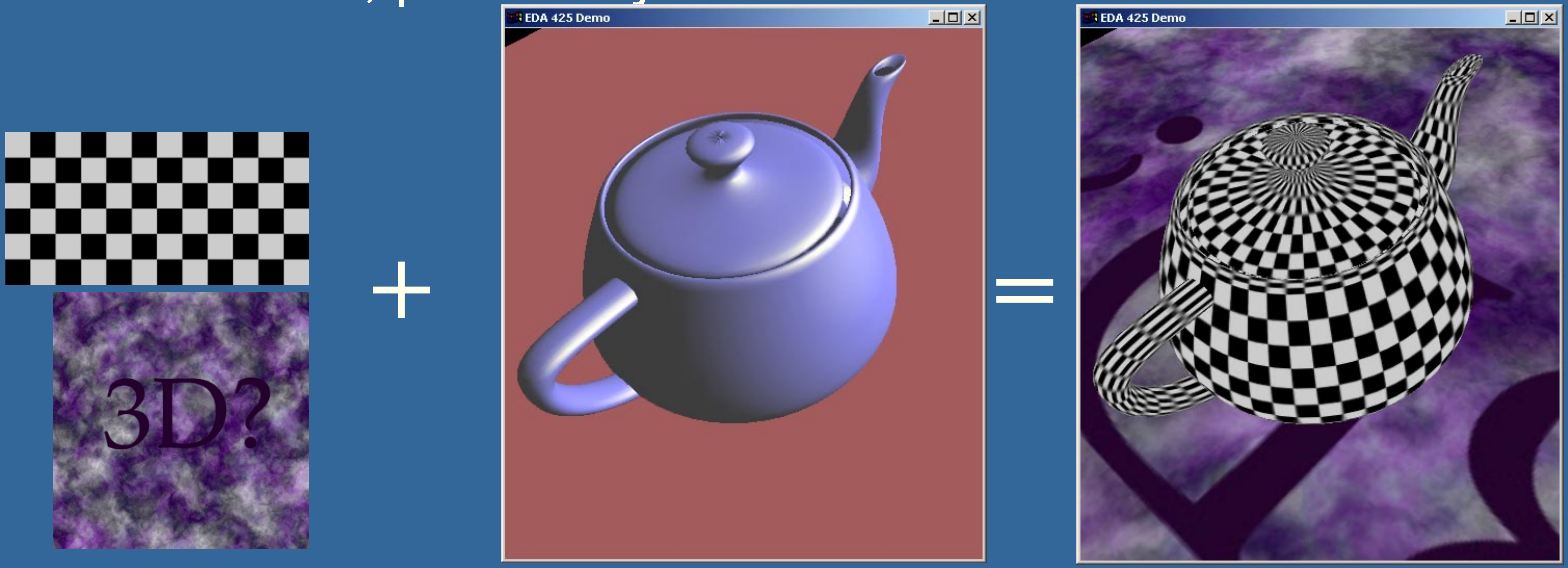
Texturing

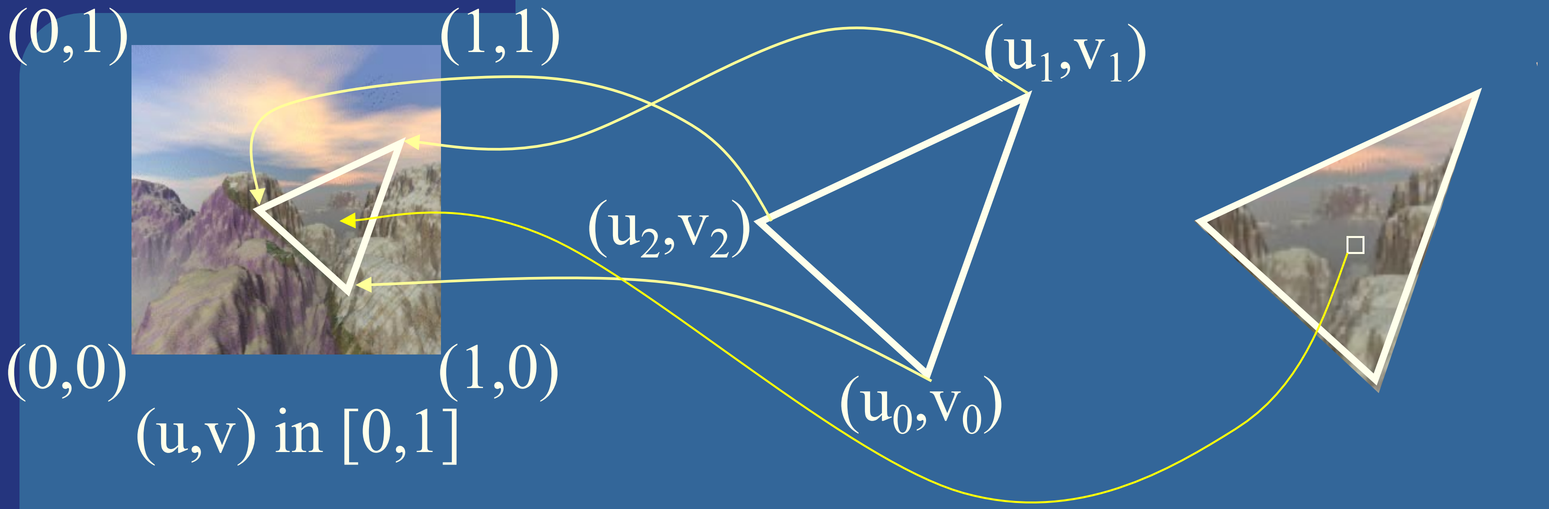
Slides done by Tomas Akenine-Möller
and Ulf Assarsson
Department of Computer Engineering
Chalmers University of Technology

Texturing: Glue n-dimensional images onto geometrical objects

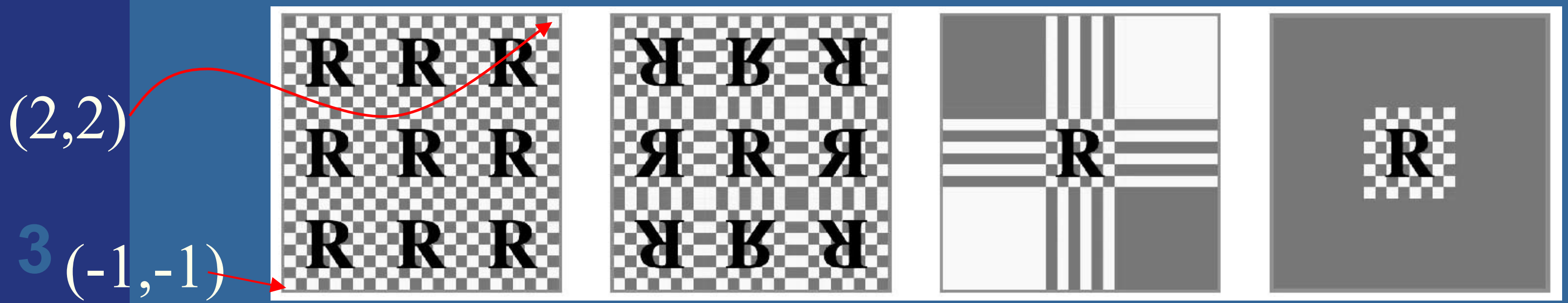
- Purpose: more realism or effects, and this is a cheap way to do it
  - Material textures (colors, roughness, metalness,…)
  - Bump mapping / normal mapping, displacement maps
  - environment mapping (cube maps), 3D textures,
  - Billboards, particle systems…



Texture coordinates

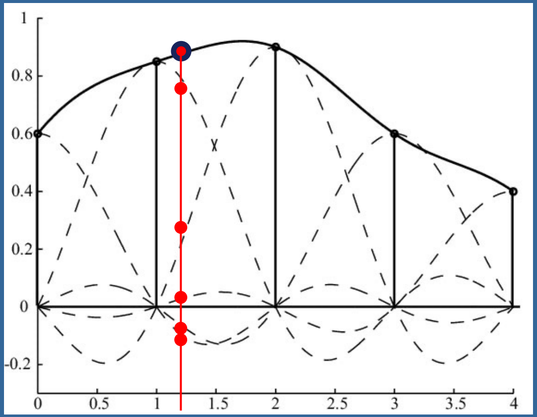


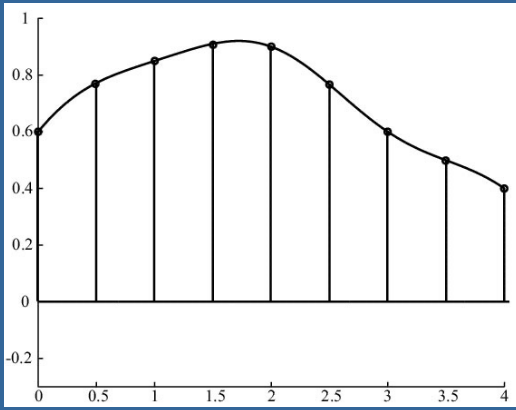
- What if (u,v) >1.0 or <0.0 ?
- To repeat textures, use just the fractional part
  - Example: 5.3 -> 0.3
- Repeat, mirror, clamp_to_edge, clamp_to_border:



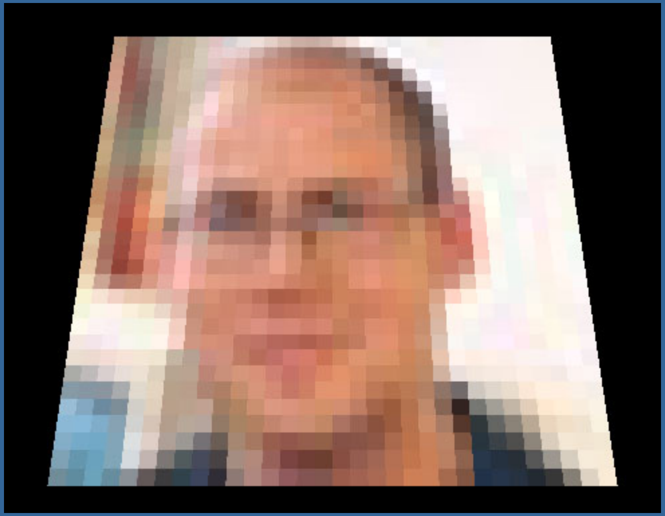
Texture magnification

- What does the theory say…
  - Let's try the sinc(x) filter since it gives best quality.
    - (for minification, use sinc(x/a) where a is the minification factor. See p:136)



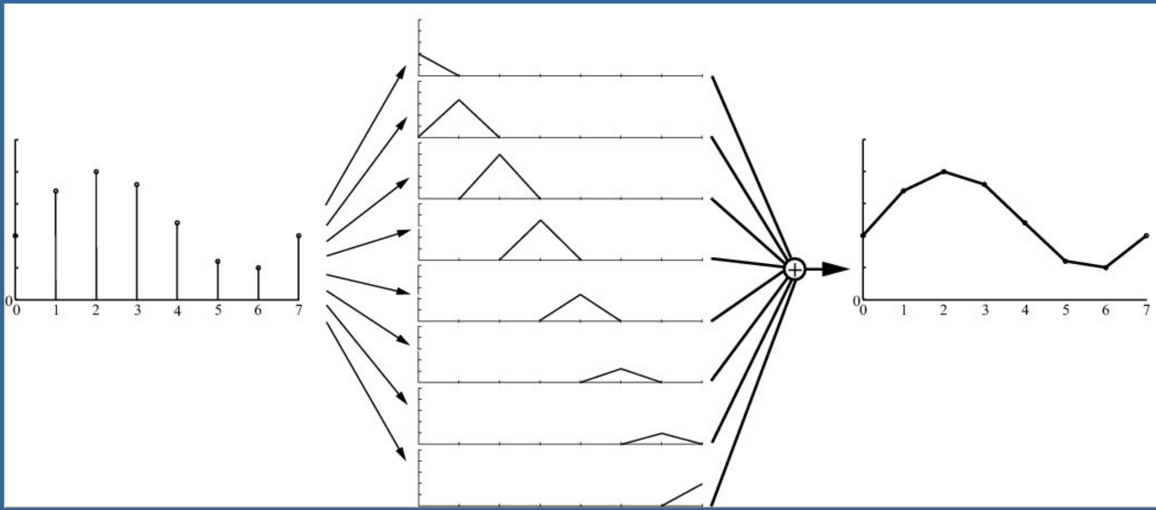


- But $\text{sinc}(x)$ is not feasible in real time
- Box filter (nearest-neighbor) is very fast
  - But poorer quality:

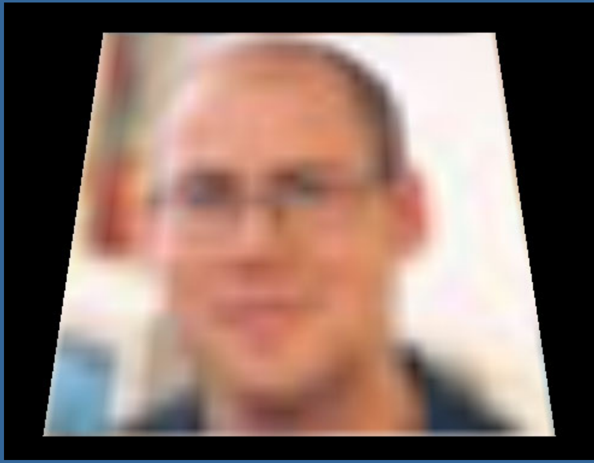


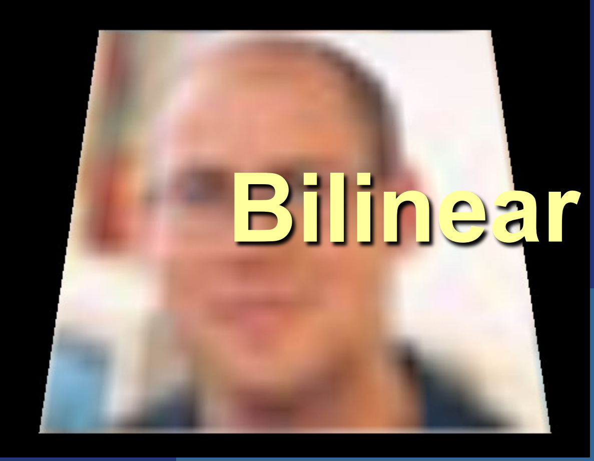
Texture magnification

- Tent filter is feasible!
- Linear interpolation

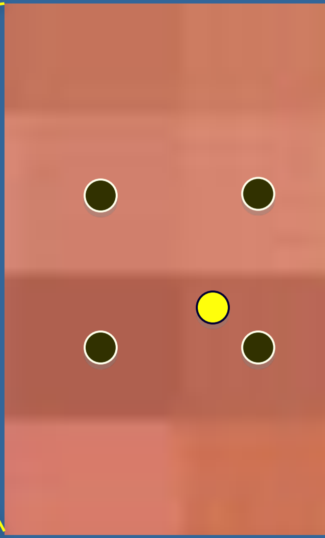


- Looks better
- Simple in 1D:
- (1-t)*color0+t*color1
- How about 2D?



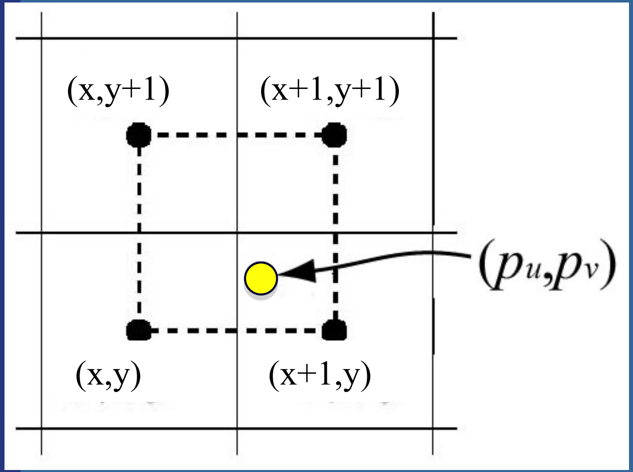


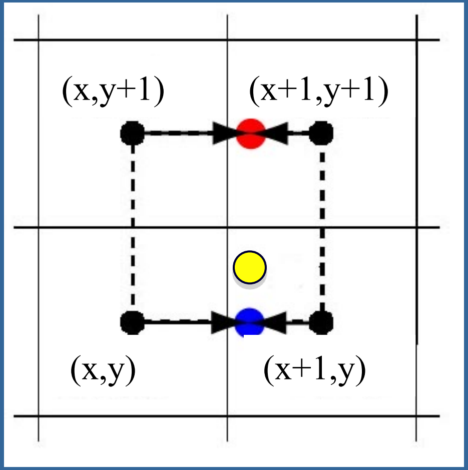
Bilinear interpolation

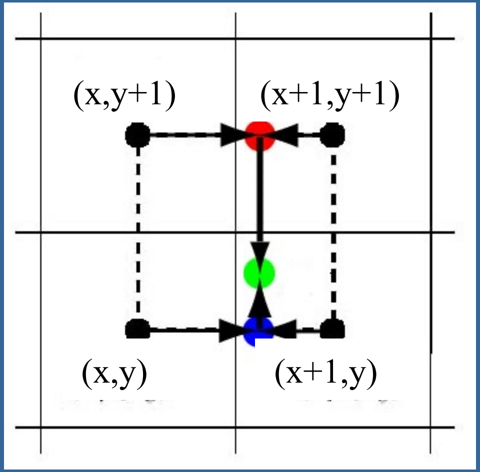


- Interpolate in 1D in x & y respectively, using the fractional part of the texel coordinate:
  1. Interpolate along texture's *x*-axis to obtain two interpolated colors.
  2. Then, interpolate between them along the y-axis.

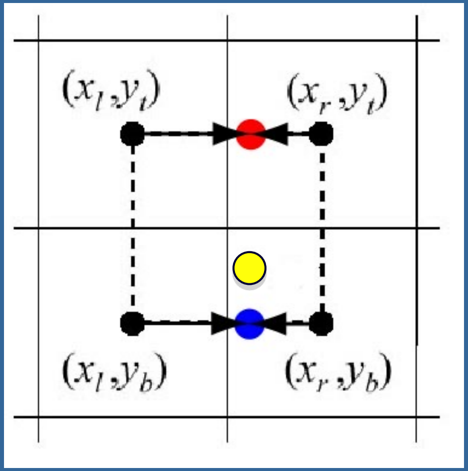
- Texture images size: n*m texels
- Texel coordinates (p_u,p_v) in [(0,0), (w,h)]
- *u,v* coordinates in [0,1]
- Nearest neighbor would access: (floor(n*u+0.5), floor(m*v+0.5))







Bilinear interpolation



Check out this formula at home

- **t**(*x*,*y*) texture color at texel (x,y)
- $p_{u,v}$ = texel coordinate
- *(u',v')* = fractional part of texel coordinate
- **b**(p_u, p_v) bilinear-filtered texture lookup

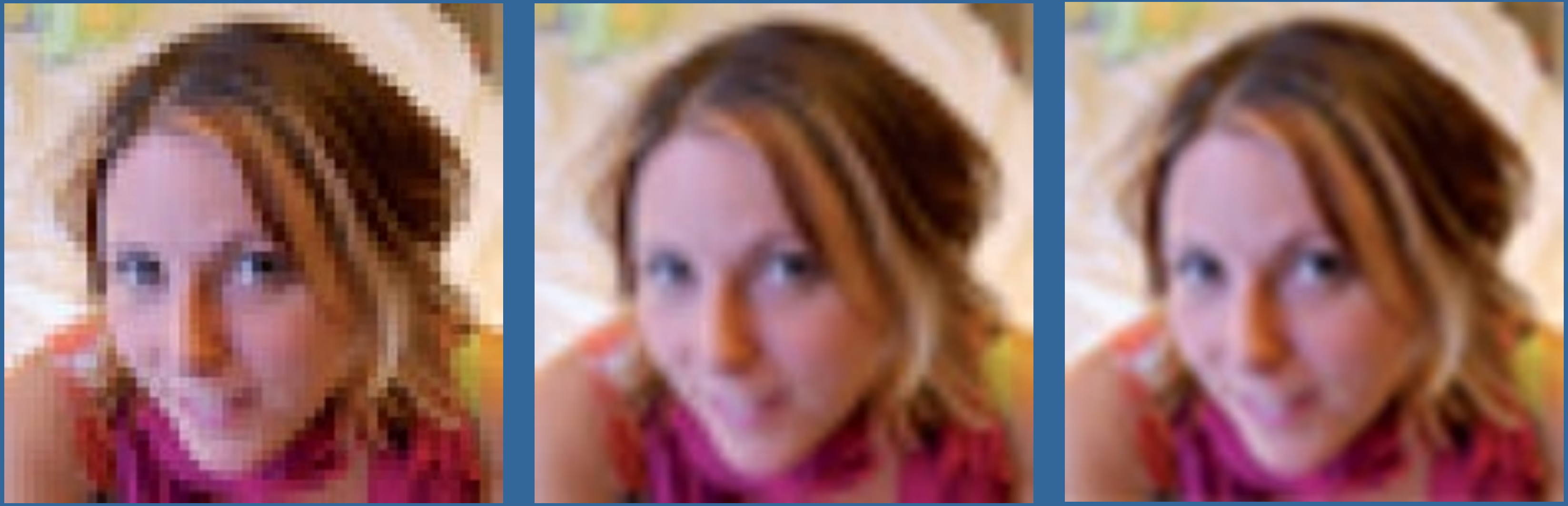
fractional part of texel coordinate $p_{u,v}$

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

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

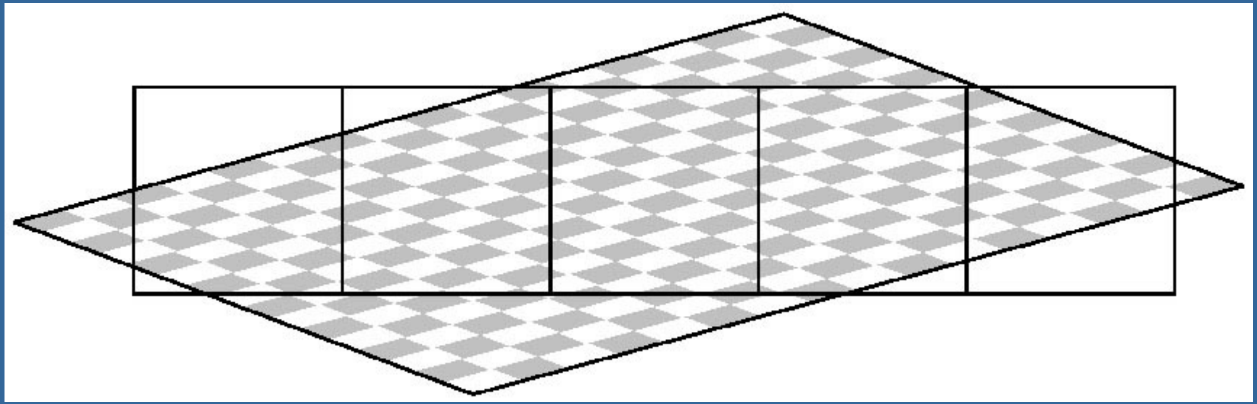
- See RTR, page 179.

Examples - filters for magnification


nearest neighbor | Bilinear filtering | 5x5 cubic filtering

Texture magnification of a 48 x 48 image onto 320 x 320 pixels. Left: nearest neighbor filtering, where the nearest texel is chosen per pixel. Middle: bilinear filtering using a weighted average of the four nearest texels. Right: cubic filtering using a weighted average of the 5x5 nearest texels.

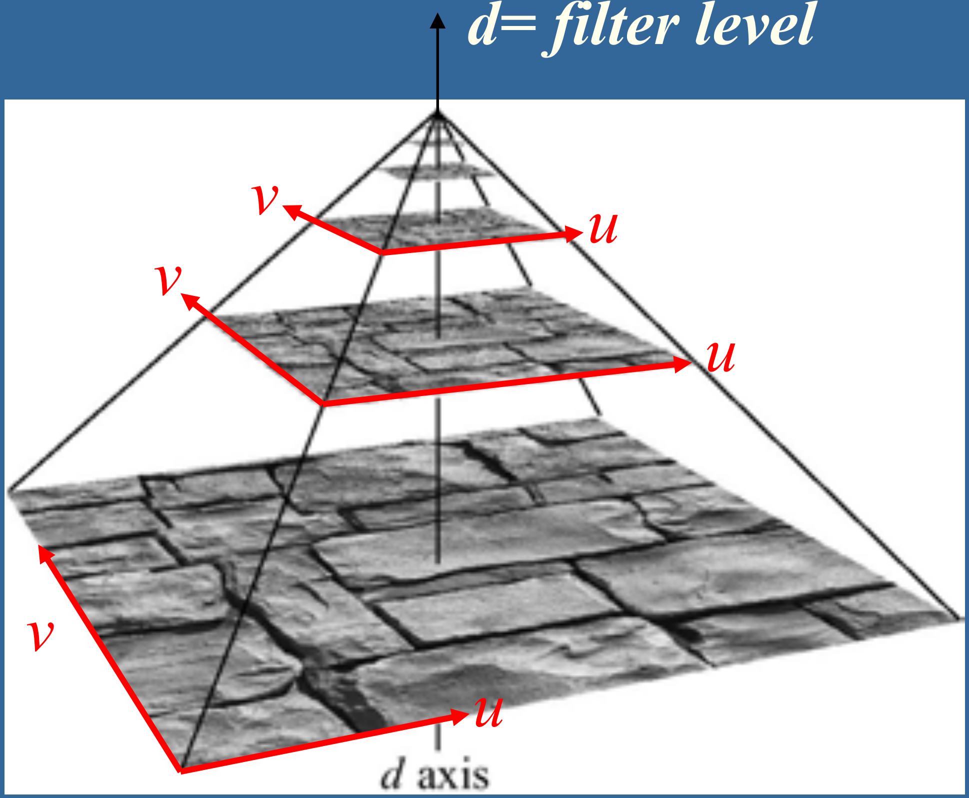
Texture minification
What does a pixel ”see”?



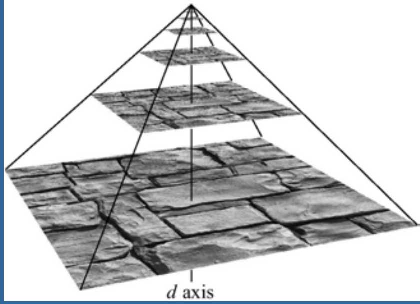
- Theory (sinc) is too expensive
- Cheaper: average of texel inside a pixel
- Still too expensive, actually

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

Mipmapping

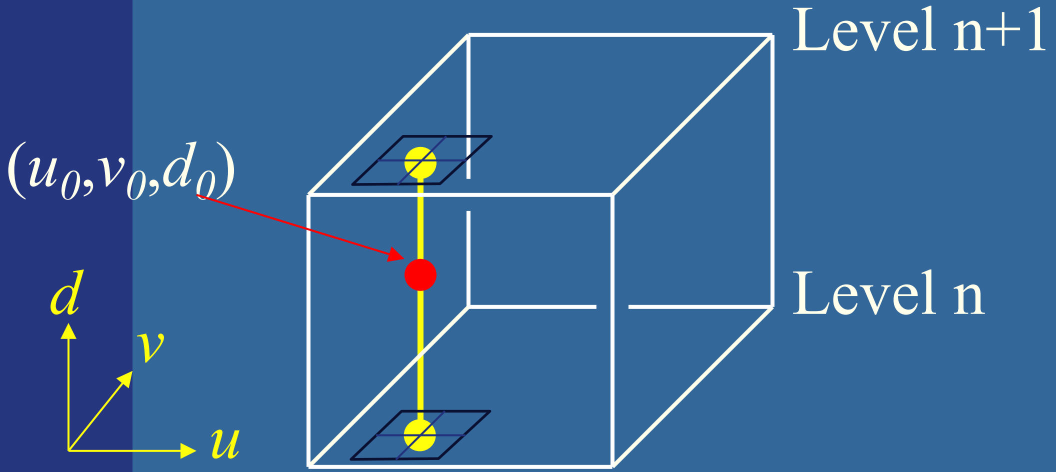


- Image pyramid
- Half width and height when going upwards
- A ”parent texel” is the average of the 4 ”child texels” from the lower level.
- Depending on amount of minification, determine which image to fetch from.
- More accurately:
  - Compute filter level, *d,* first. Will be somewhere between 2 levels.
  - Do bilinear interpolation in both levels, and then a linear interpolation between the 2 levels, based on fraction of *d*…



Mipmapping

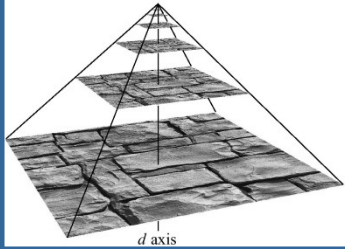
- Do bilinear interpolation in both levels,
- and then a linear interpolation between the 2 levels, based on fraction of *d*…
- Gives trilinear interpolation

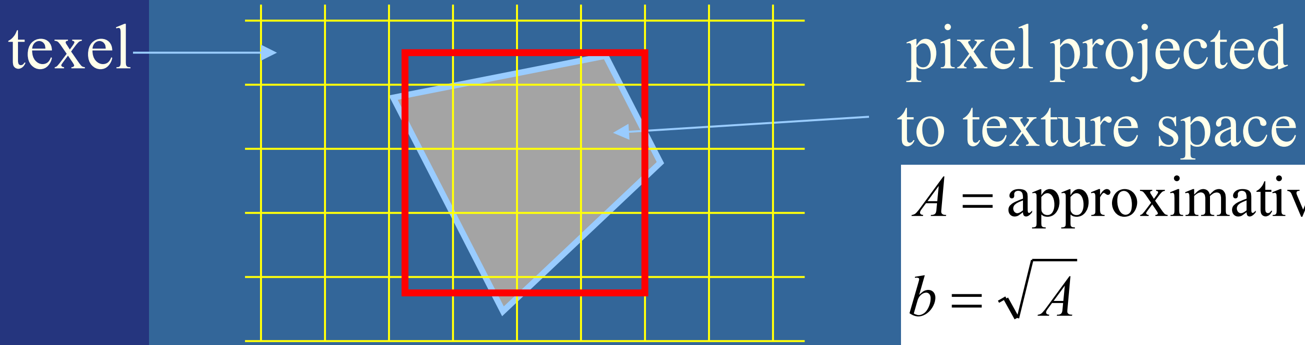


- Constant time filtering: 8 texel accesses
- How to compute *d* (i.e., the filter level)?

Bonus:

Computing *d* for mipmapping – the old way





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

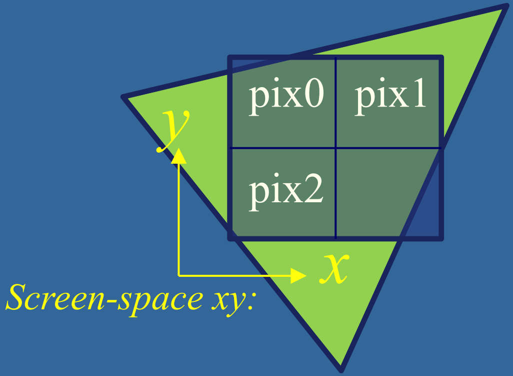
$$b = \sqrt{A}$$

$$d = \log_2 b$$

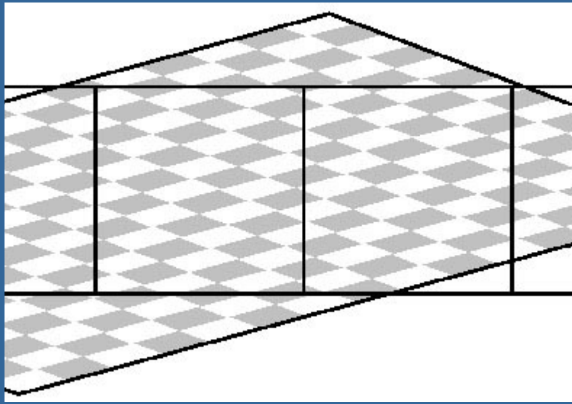
- Approximate quad with square
- (Can give some overblur as seen here, but better to much blur than too little – to avoid aliasing artifacts.)

Computing *d* for mipmapping today

- Is based on how much the texel coordinate changes between adjacent pixels.
- Fragment shaders are always executed in parallel for at least 2x2 pixel blocks.
  - Hence, the GPU knows the uv-coordinate difference between such 2x2 pixels
  - Let's say texture image size is n*m texels
  - Needed minification in *u*-direction: max(*du/dx, du/dy)*n*
  - Needed minification in *v*-direction: max(*du/dx, du/dy)*m*



$$du/dx = u_{pix1} - u_{pix0}$$
$$du/dy = u_{pix2} - u_{pix0}$$
$$dv/dx = v_{pix1} - v_{pix0}$$
$$dv/dy = v_{pix2} - v_{pix0}$$

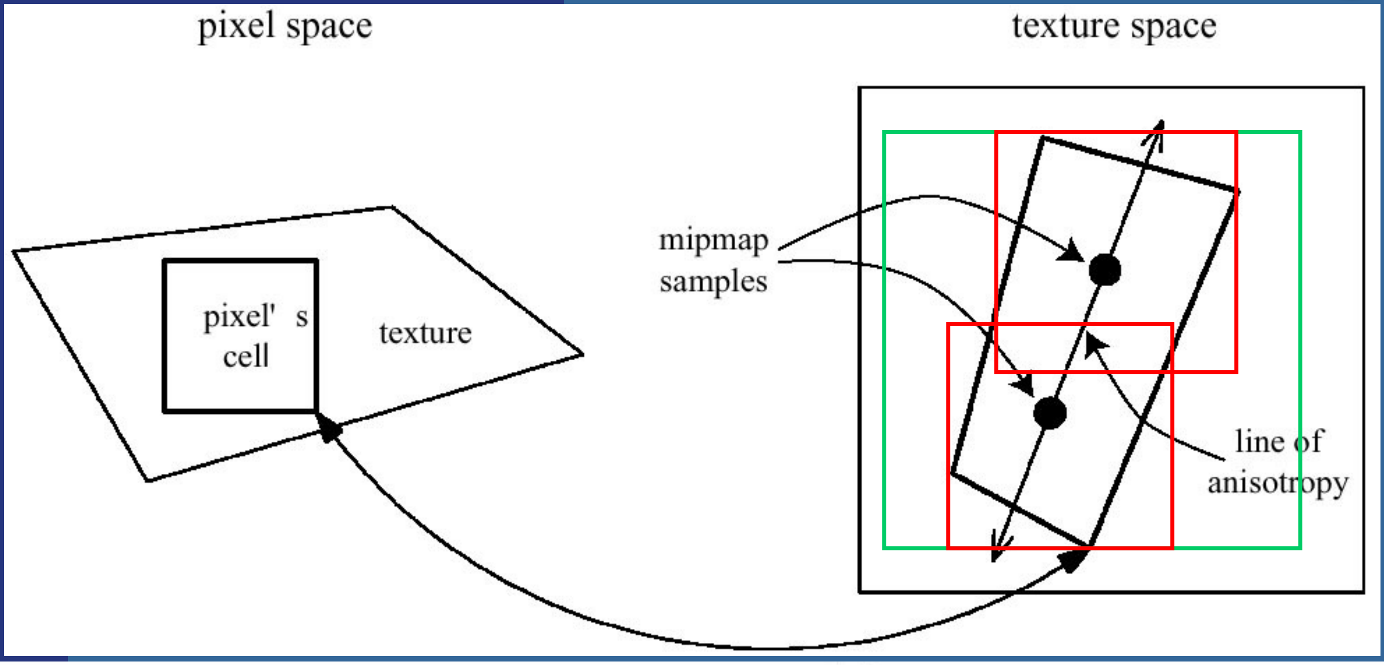


E.g.: $\boldsymbol{d}_{u,v} = \log_2 \max\left(\left(\frac{du}{dx}, \frac{du}{dy}\right)n, \left(\frac{dv}{dx}, \frac{dv}{dy}\right)m \right)$

Take max of required filtering in u and v direction → $d = \max(d_u, d_v)$

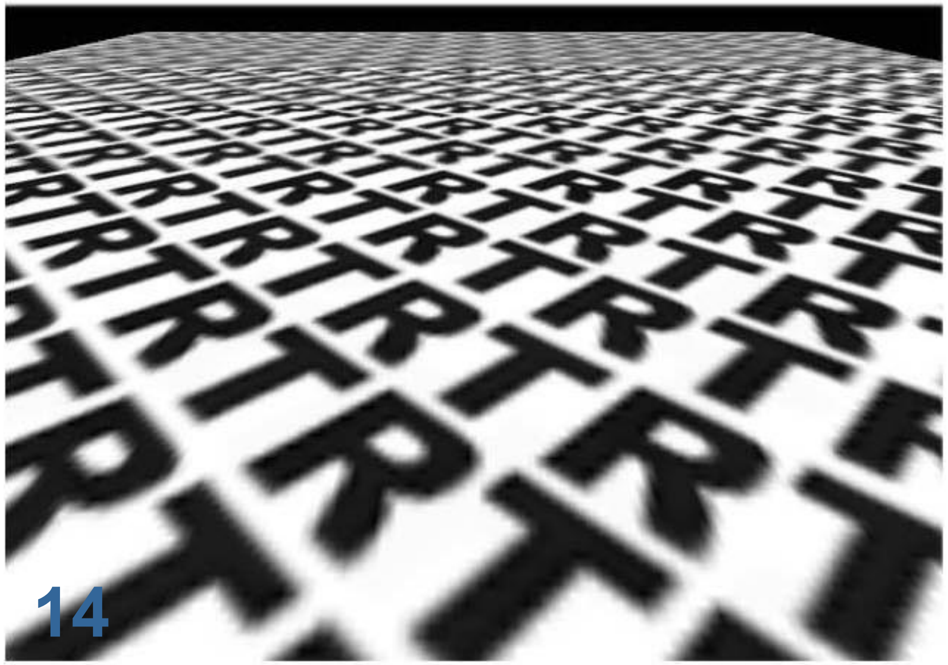
- Take max of d_u, d_v since overblur is better than aliasing.
- Nevertheless, if $d_u \neq d_v$, then d gives overblur for one of the dimensions.
  - Even better: *anisotropic* texture filtering
    - Approximate pixel coverage with several smaller mipmap samples… see next slide

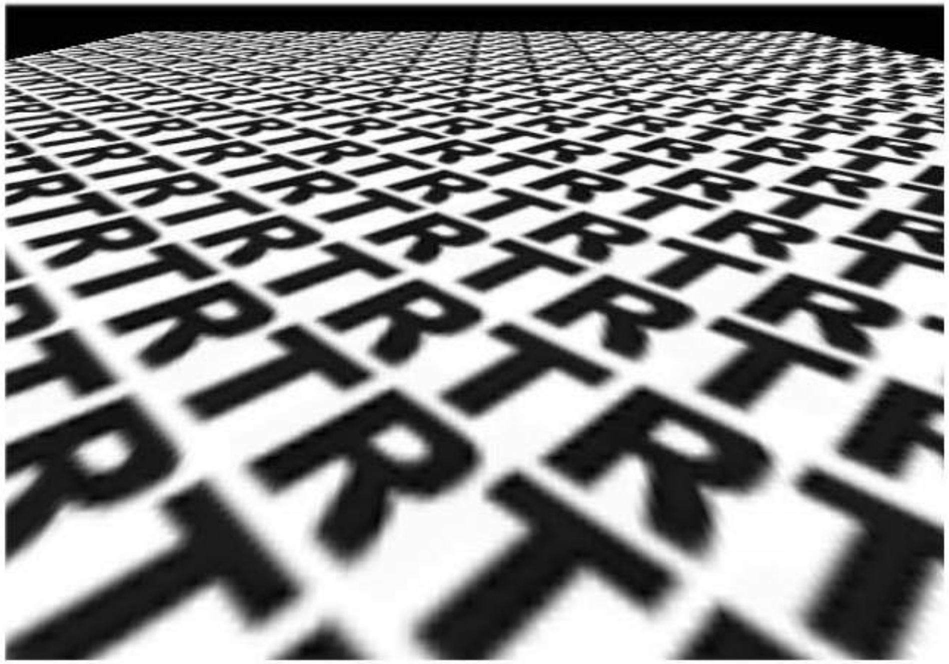
Anisotropic texture filtering

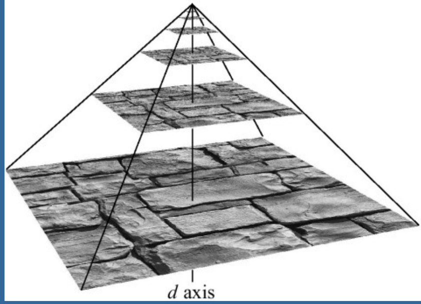


Approximate pixel coverage with several smaller mipmap lookups along the line of anisotropy.

16 samples

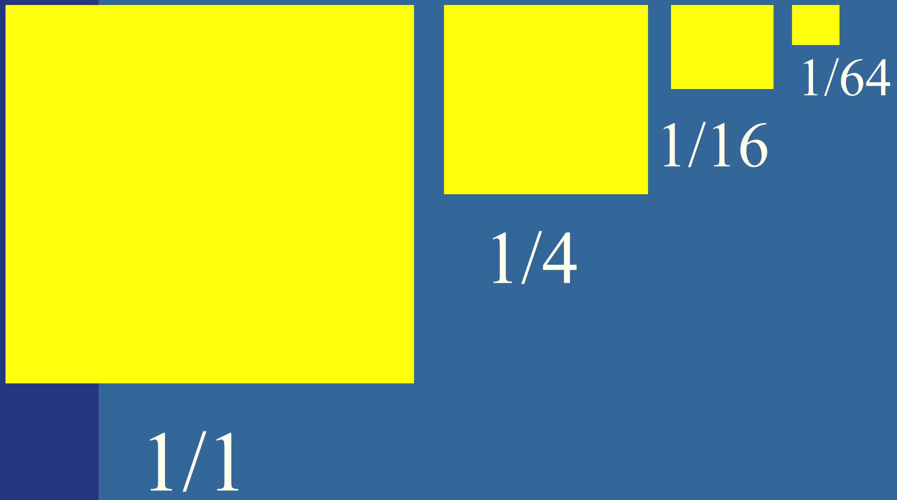


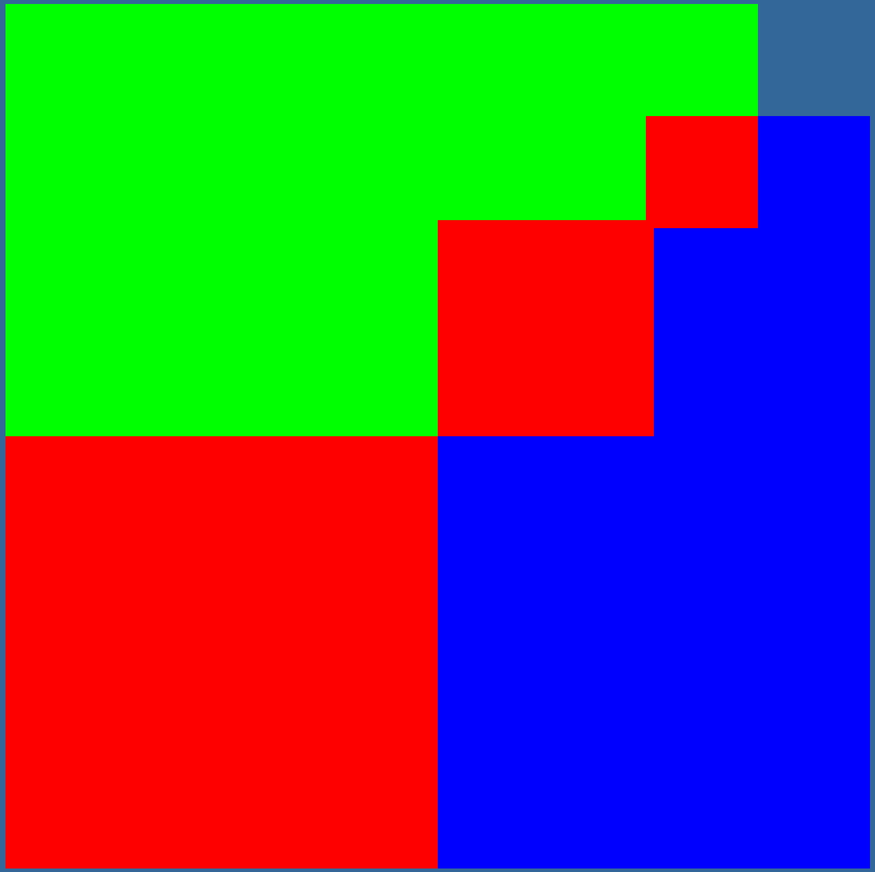




Mipmapping: Memory requirements

- Not twice the number of bytes…!

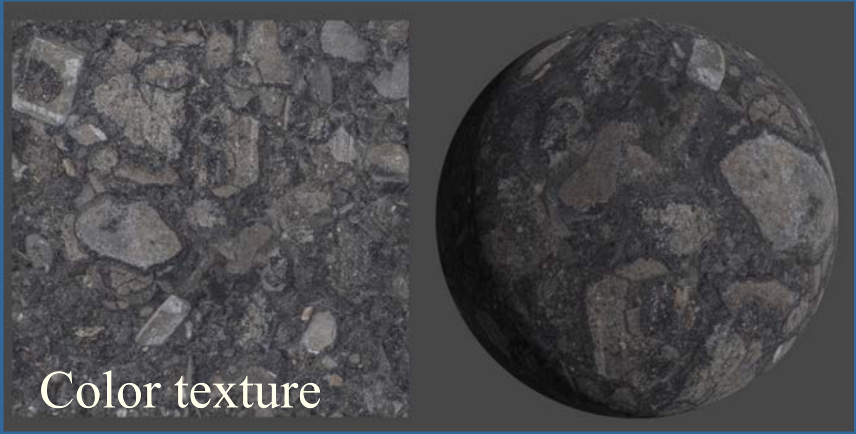


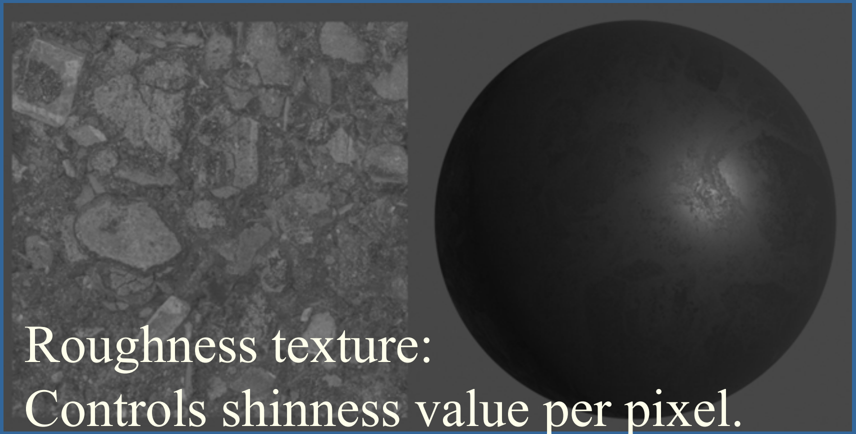


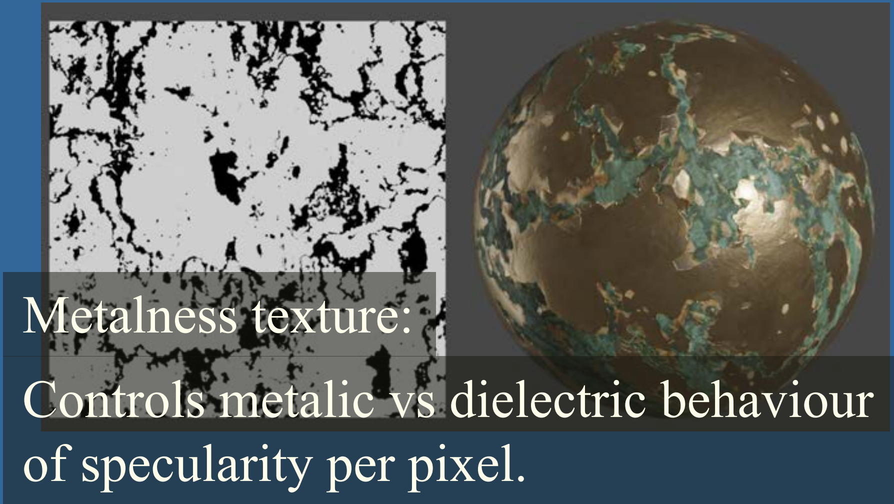
- Rather 33% more – not that much

Materials

- Textures:
  - vary material parameters over the surfaces, used by the lighting computations
- Common texture maps:
  - Color, Roughness, Metalness, Normal texture
  - See lab 4 for these material parameters.


Color texture


Roughness texture:
Controls shinness value per pixel.


Metalness texture:
Controls metalic vs dielectric behaviour of specularity per pixel.

Using textures in OpenGL

Do once when loading texture:

```
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture);
int w, h, comp; // width, height, #components (rgb=3, rgba=4), #comp
unsigned char* image = stbi_load("floor.jpg", &w, &h, &comp, STBI_rgb_alpha);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, w, h, 0, GL_RGBA, GL_UNSIGNED_BYTE, image);
stbi_image_free(image);
glGenerateMipmap(GL_TEXTURE_2D);

//Indicates that the active texture should be repeated over the surface
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
// Sets the type of mipmap interpolation to be used on magnifying and minifying the texture. These are the
// nicest available options.
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAX_ANISOTROPY_EXT, 16);
```

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

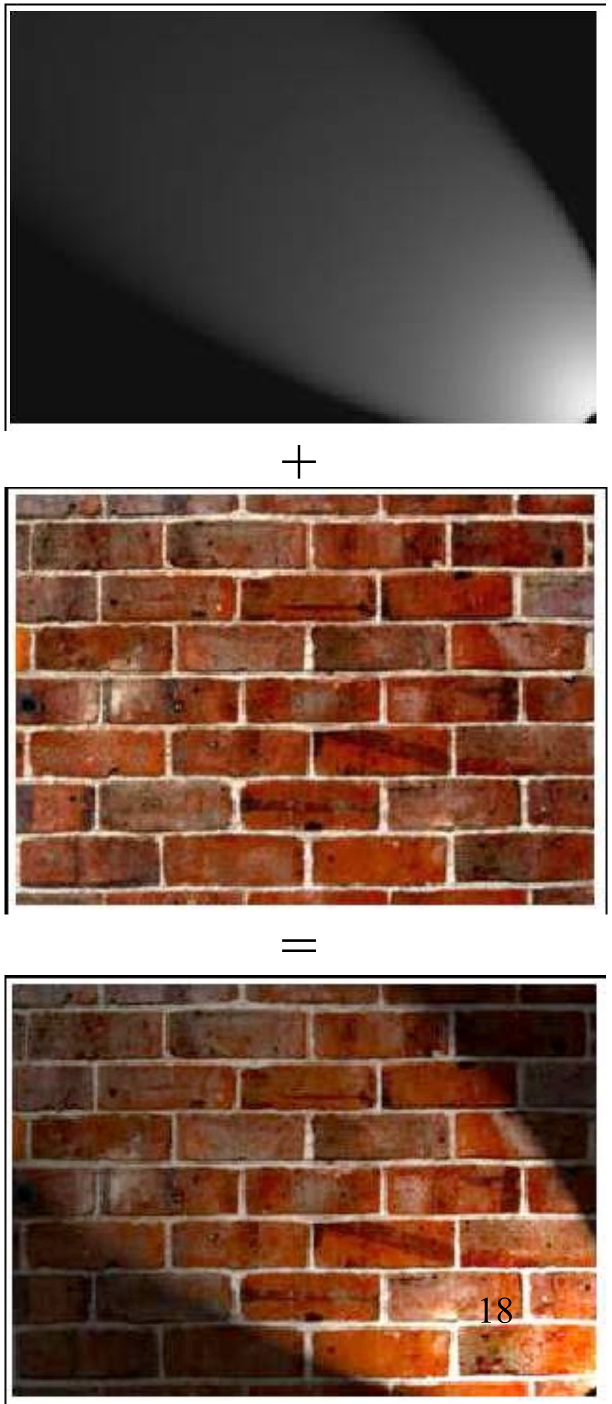
```
//selects which texture unit subsequent texture state calls will affect:
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture);
// Now, draw your triangles with texture coordinates specified
```

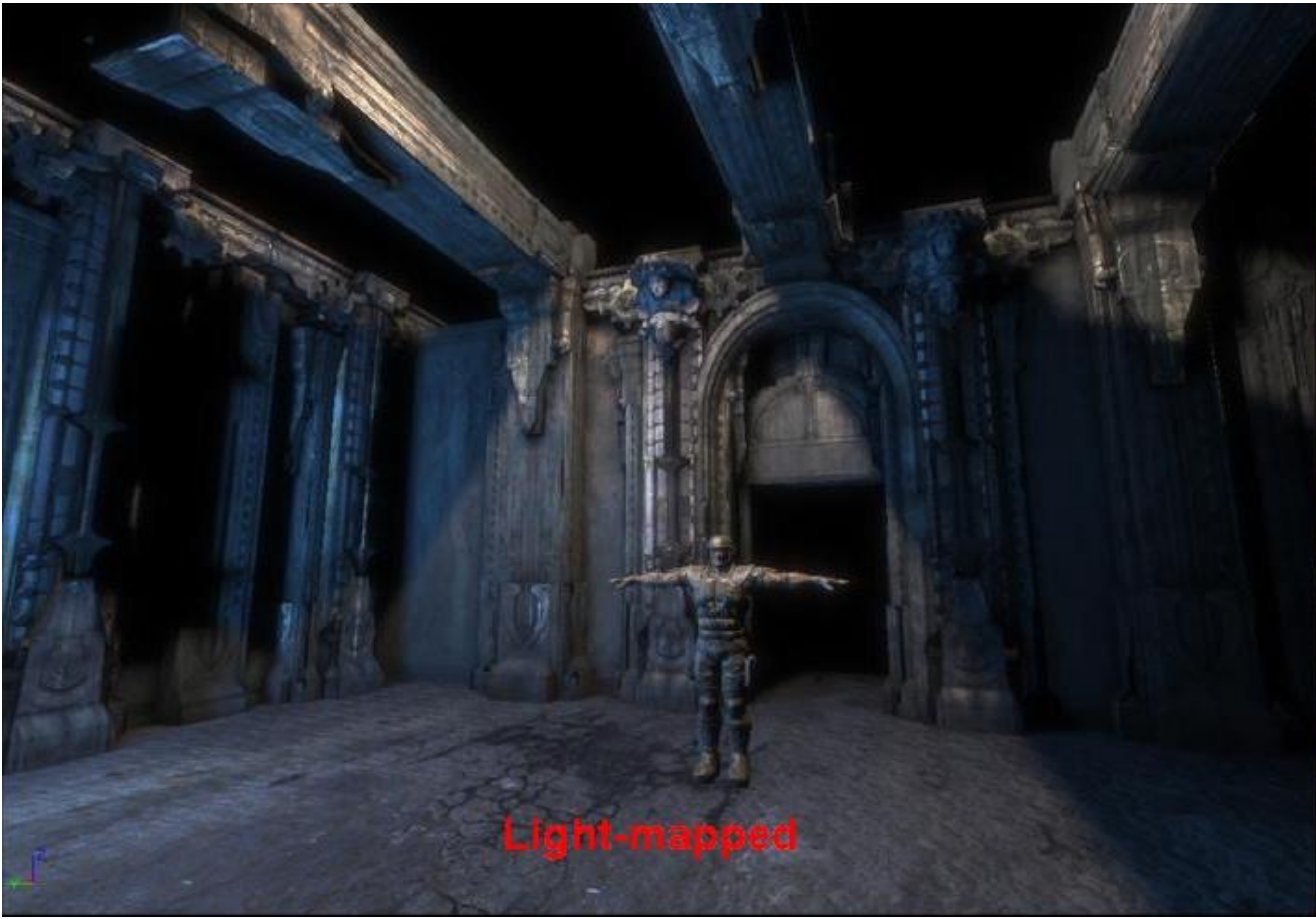
FRAGMENT SHADER

```
in vec2 texCoord;
layout(binding = 0) uniform sampler2D coltex;

void main()
{
   gl_FragColor = texture2D(coltex,  texCoord.xy);
}
```

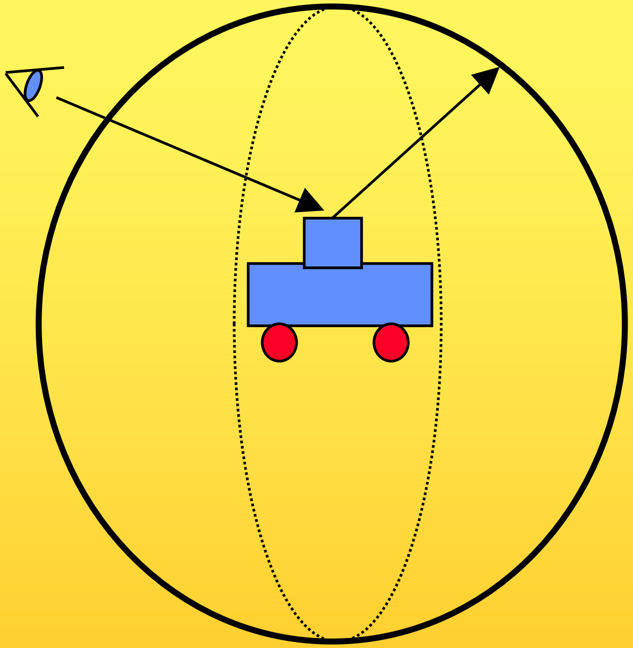
Light Maps

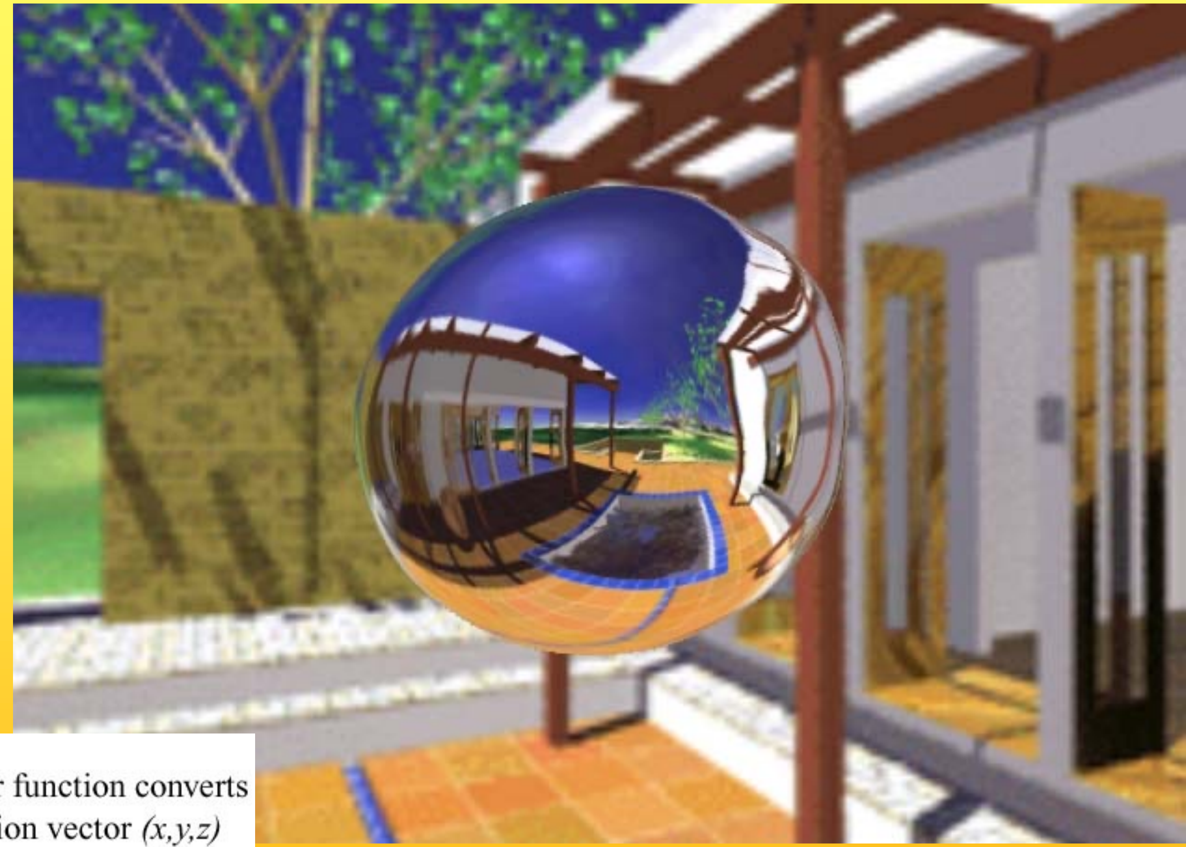
- Often used in games
- Mutliply both textures with each other in the fragment shader, or (old way):
  - render wall using brick texture
  - render wall using light texture and blending to the frame buffer

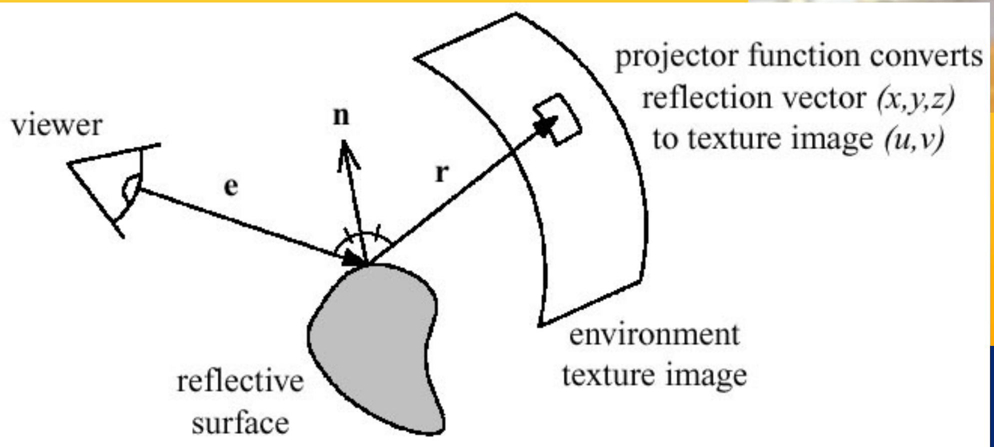



Light-mapped

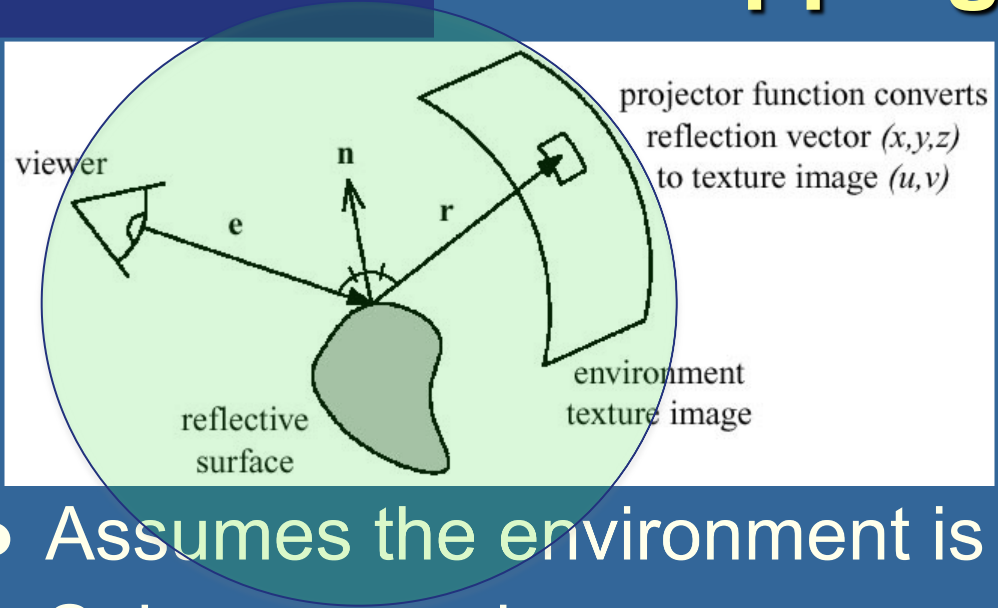
Environment mapping

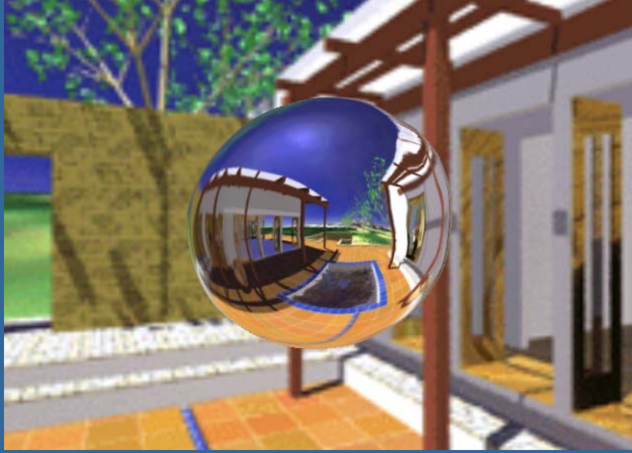






Environment mapping

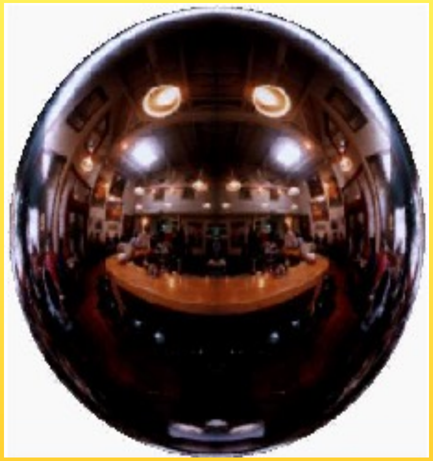

viewer
n
e
r
projector function converts reflection vector (x,y,z) to texture image (u,v)
environment texture image
reflective surface

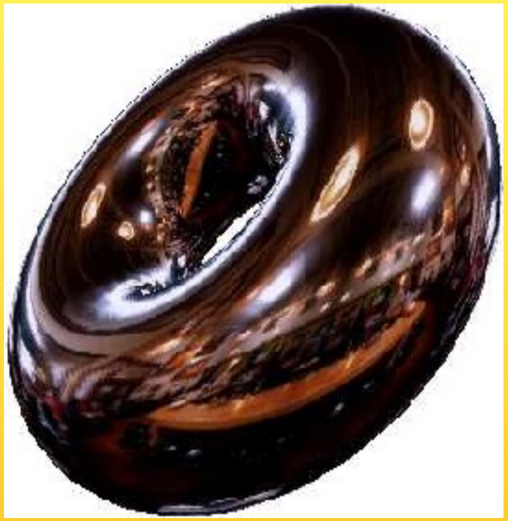


- Assumes the environment is infinitely far away
- Sphere mapping
- Cube mapping is the norm nowadays
  - Advantages: no singularities as in sphere map
  - Much less distortion
  - Gives better result
  - Not dependent on a view position

Sphere map

- example


Sphere map (texture)


Sphere map applied on torus

Sphere Map

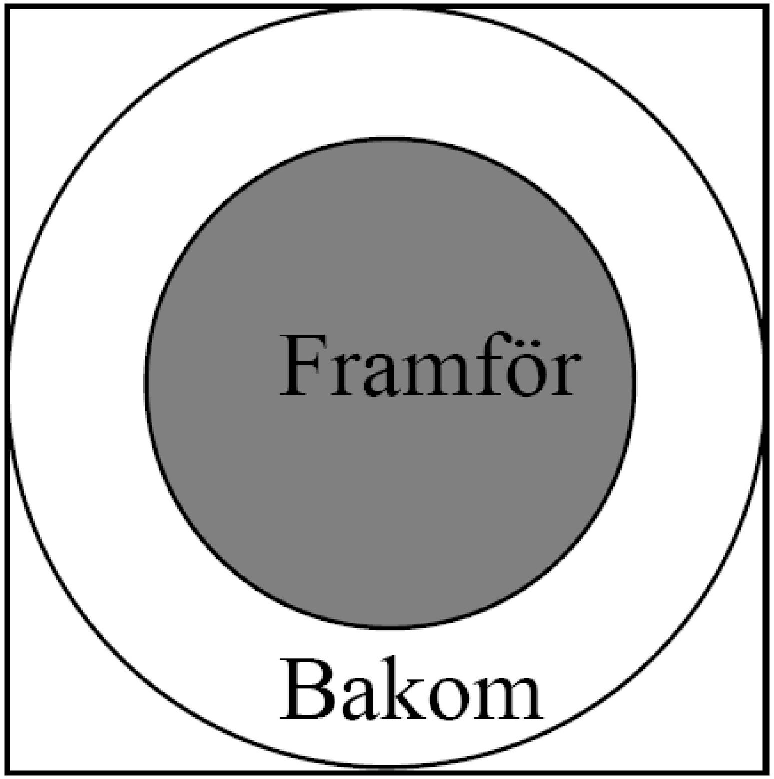
- Assume surface normals are available
- Then OpenGL can compute reflection vector at each pixel
- The texture coordinates s,t are given by:
  - (see OH 169 for details)

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

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

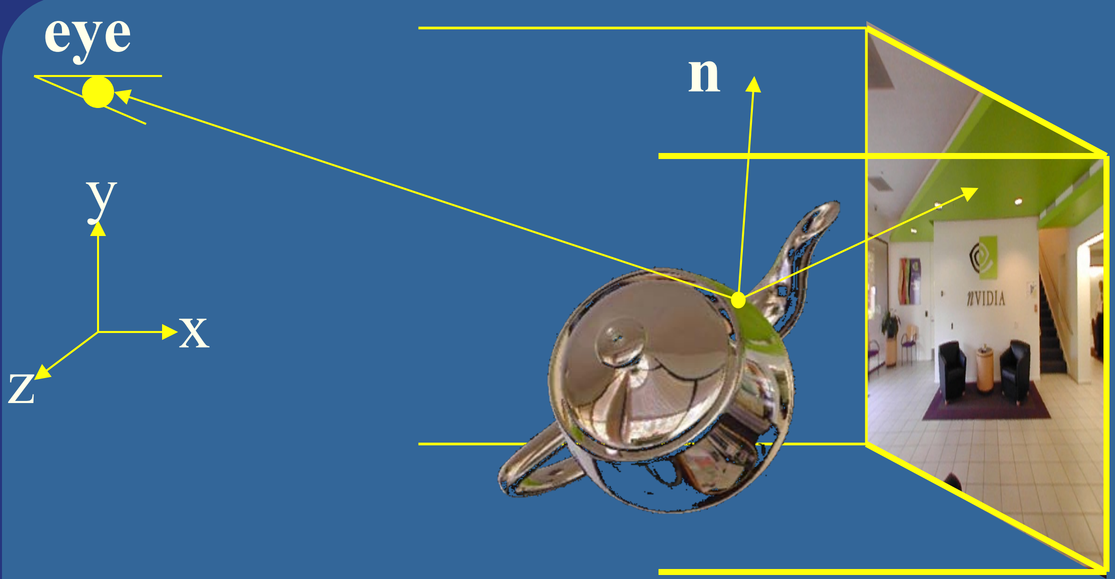
$$t = 0.5\left(\frac{R_y}{L} + 1\right)$$

Sphere Map



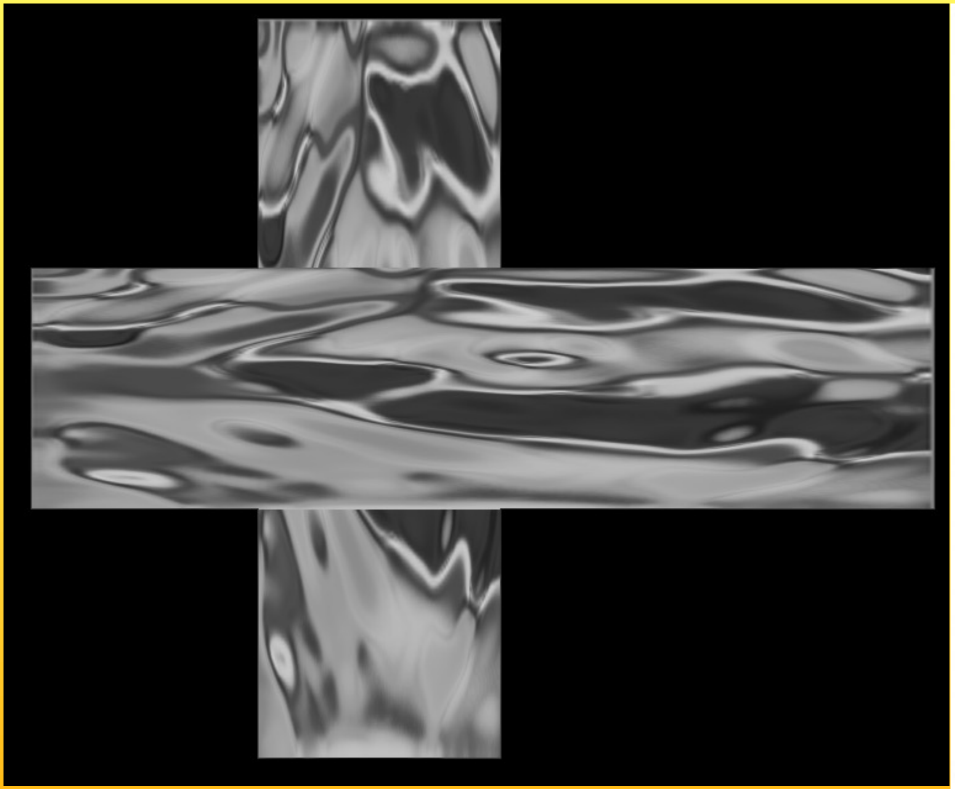
In front of the sphere.
Behind the sphere.

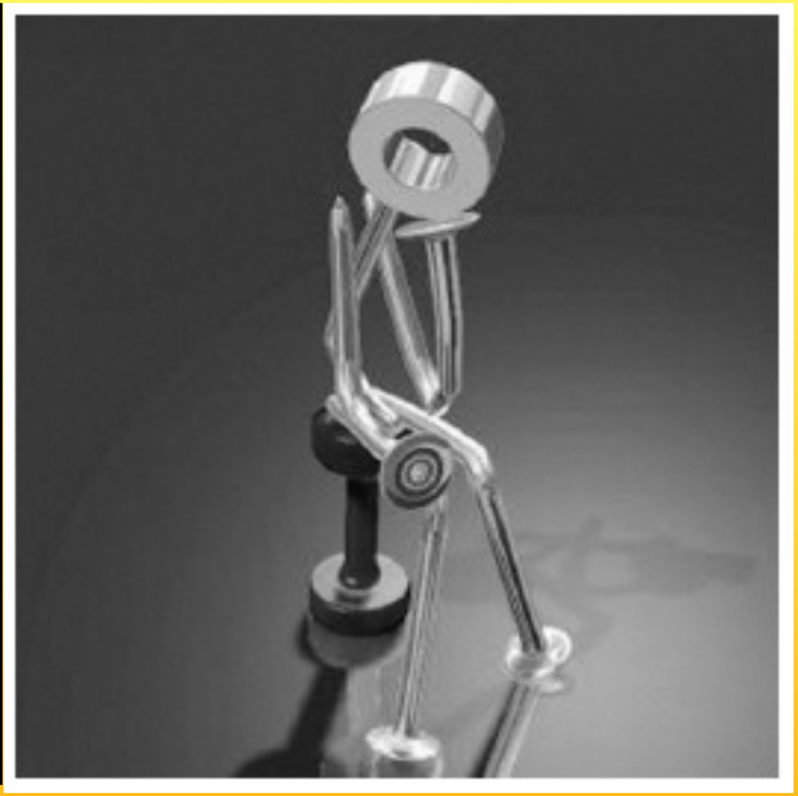
Cube mapping

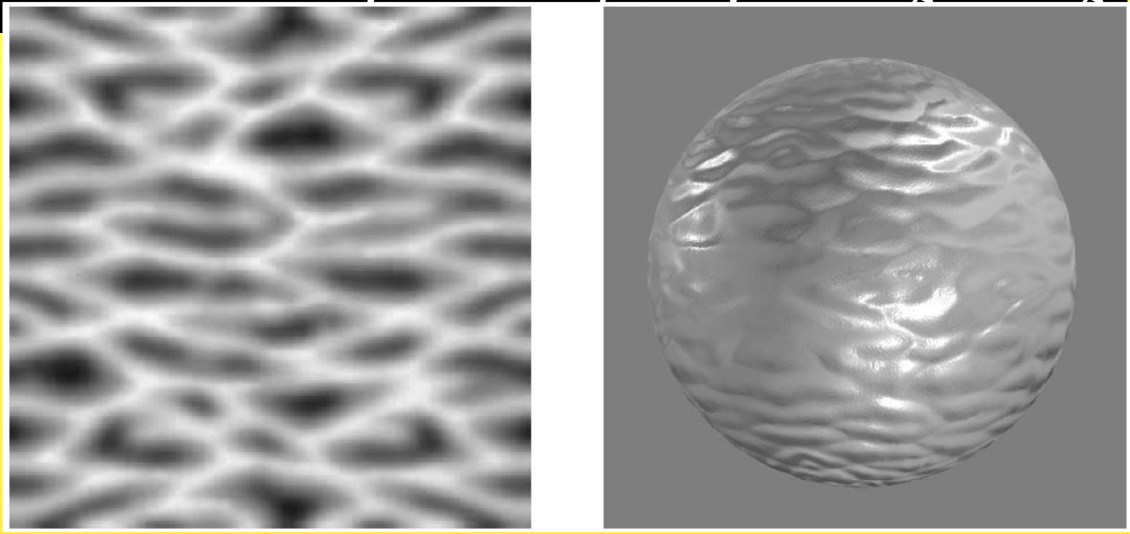


- Simple math: compute reflection vector, **r**
- Largest abs-value of component, determines which cube face.
  - Example: **r**=(5,-1,2) gives POS_X face
- Divide **r** by abs(5) gives (*u*,*v*)=(-1/5,2/5)
- Remap from [-1,1] to [0,1], i.e., ((u,v)+(1,1))/2
- Your hardware does all the work. You just have to compute the reflection vector. (See lab 4)

Example

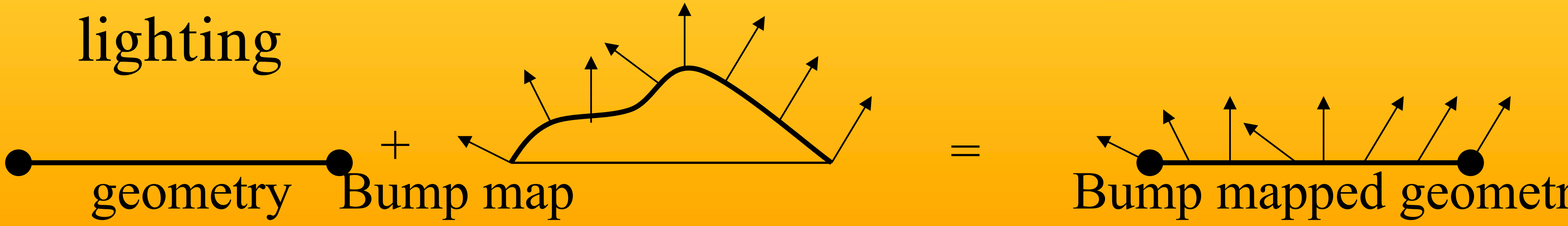






Bump mapping

- 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

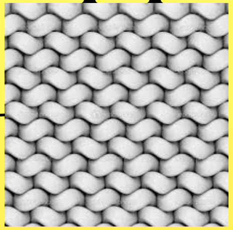


Stores heights: can derive normals

Bump mapping

Storing bump maps:

1. as a **gray scale** image
   or

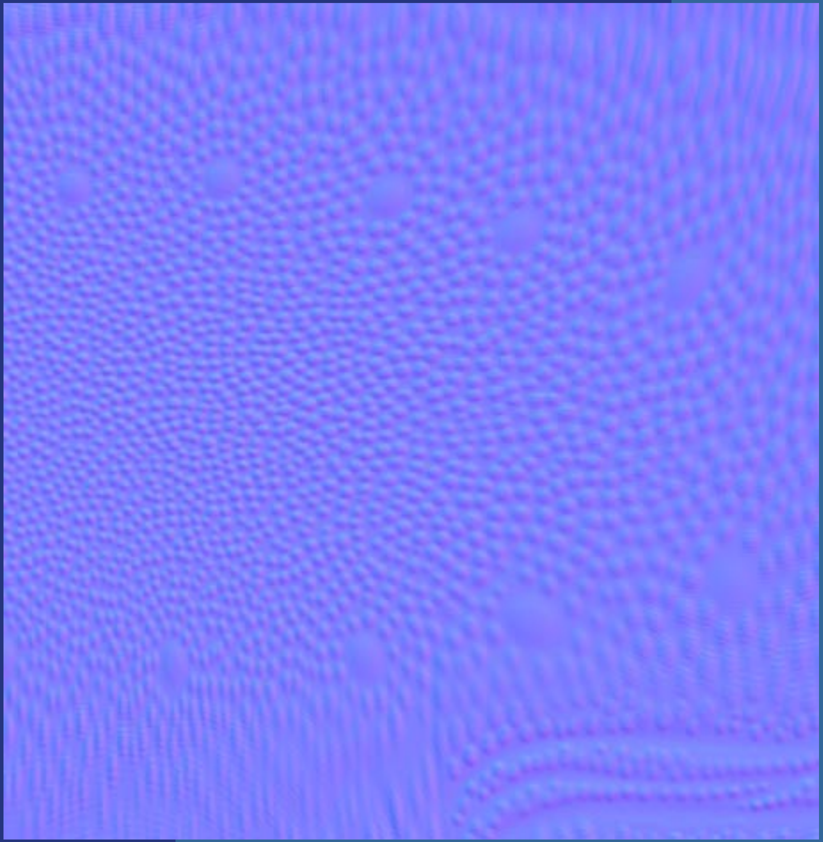


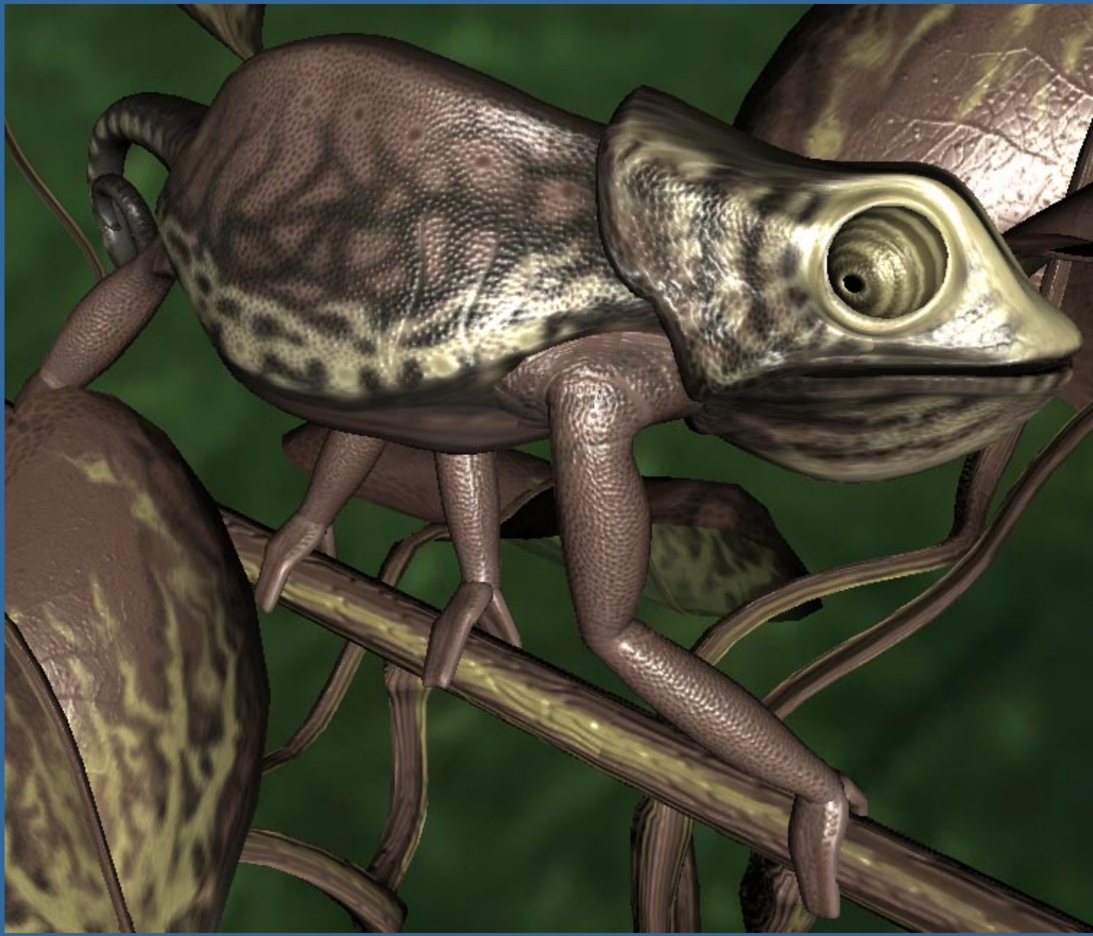
2. as **normals** (n_x, n_y, n_z)

(Then typically called *Normal mapping*)

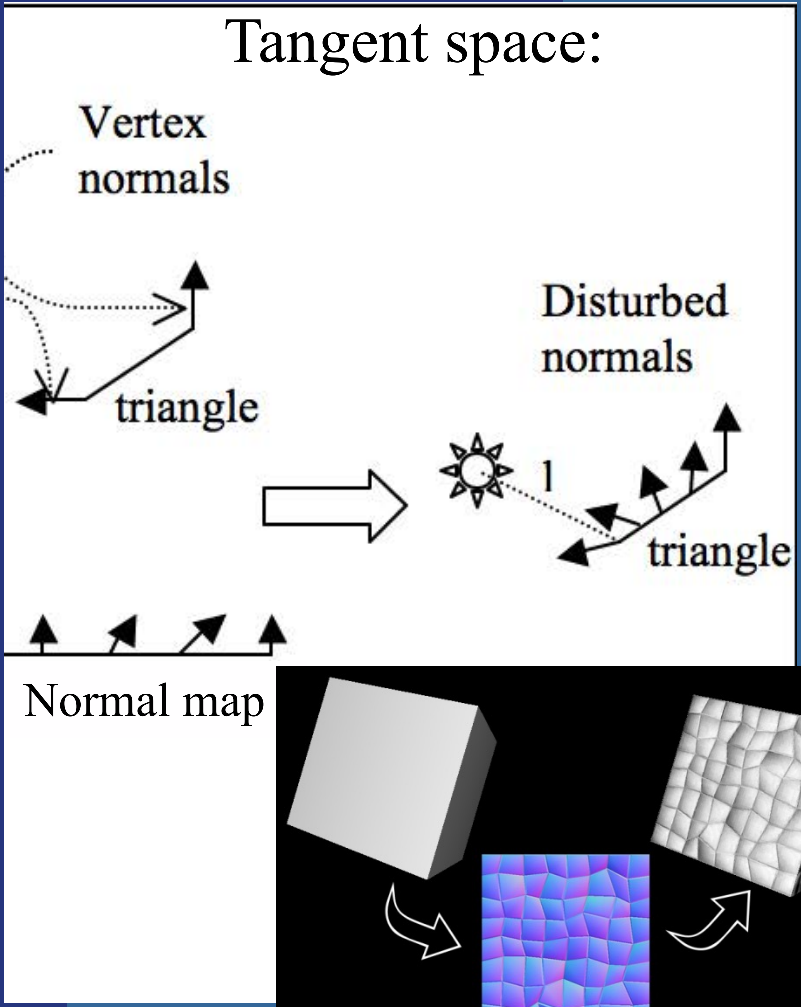
- How to store normals in a texture:
  - $\mathbf{n}=(n_x, n_y, n_z)$ are in [-1,1]
  - Add 1, mult 0.5: in [0,1]
  - Mult by 255 (8 bit per color component)
  - Values can now be stored in 8-bit rgb texture

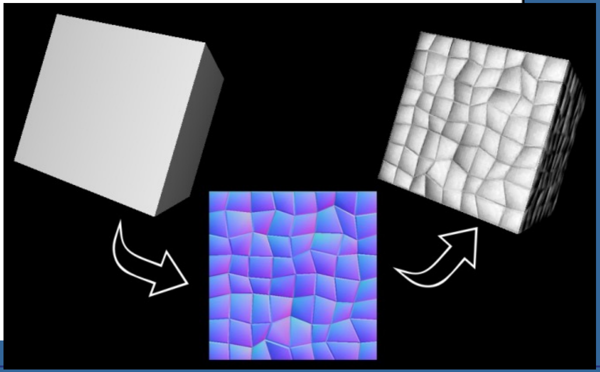
Bump mapping: example





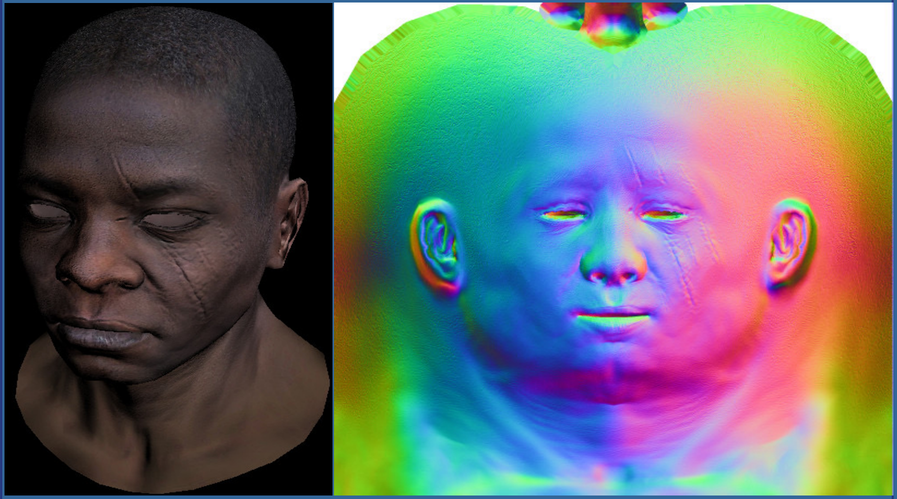
Normal mapping in tangent vs object space





Object space:

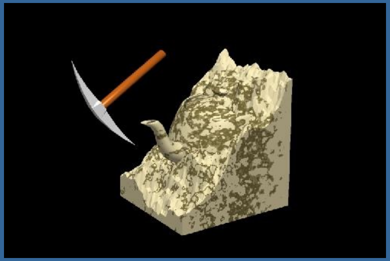
•Normals are stored directly in model space. I.e., as including both face orientation plus distortion.

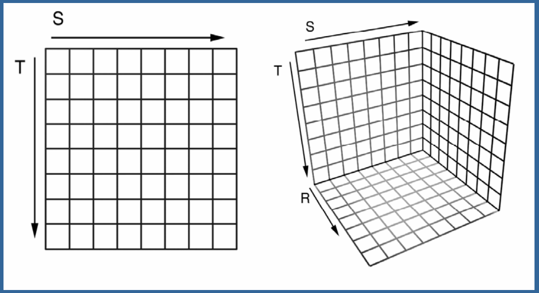


Tangent space:

●Normals are stored as distortion of face orientation. The same bump map can be tiled/repeated and reused for many faces with different orientation

More...



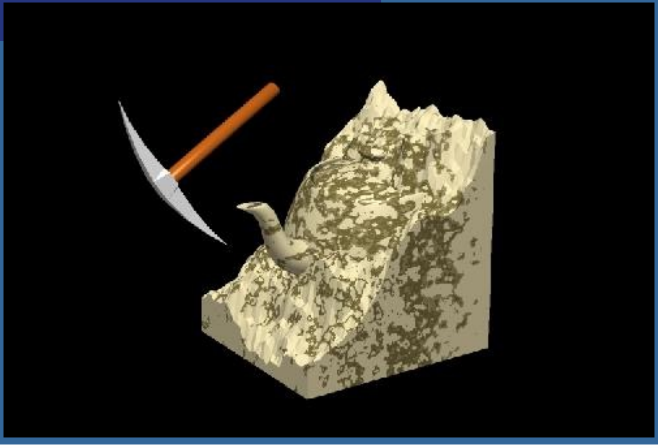
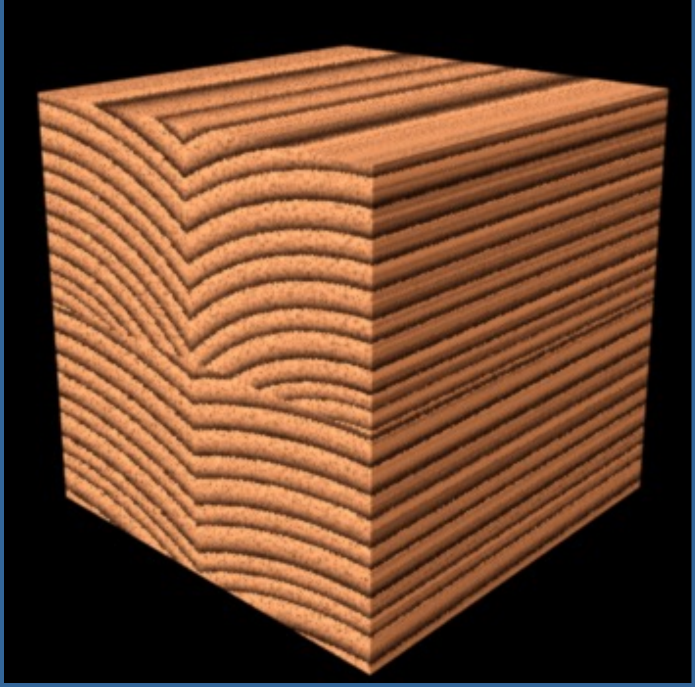
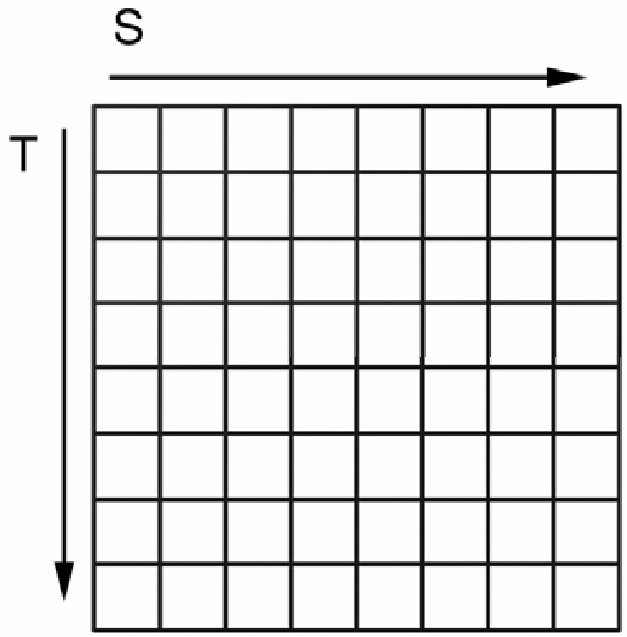


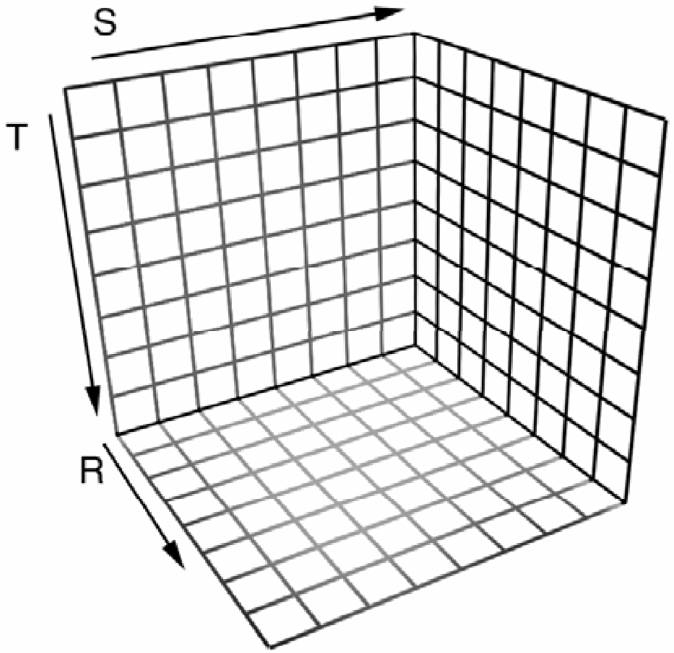
2D 3D

- 3D textures:
  - Texture filtering is no longer trilinear
  - Rather quadlinear
    - (trilinear interpolation in both 3D-mipmap levels + between mipmap levels)
  - Enables new possibilities
    - Can store light in a room, for example

- Displacement Mapping
  - Like bump/normal maps but truly offsets the surface geometry (not just the lighting).
  - Gfx hardware cannot offset the fragment's position
    - Offsetting per vertex is easy in vertex shader but requires a highly tessellated surface.
    - Tesselation shaders are created to increase the tessellation of a triangle into many triangles over its surface. Highly efficient.
    - (Can also be done using Geometry Shader (e.g. Direct3D 10) by ray casting in the displacement map, but tessellation shaders are generally more efficient for this.)

2D texture vs 3D texture

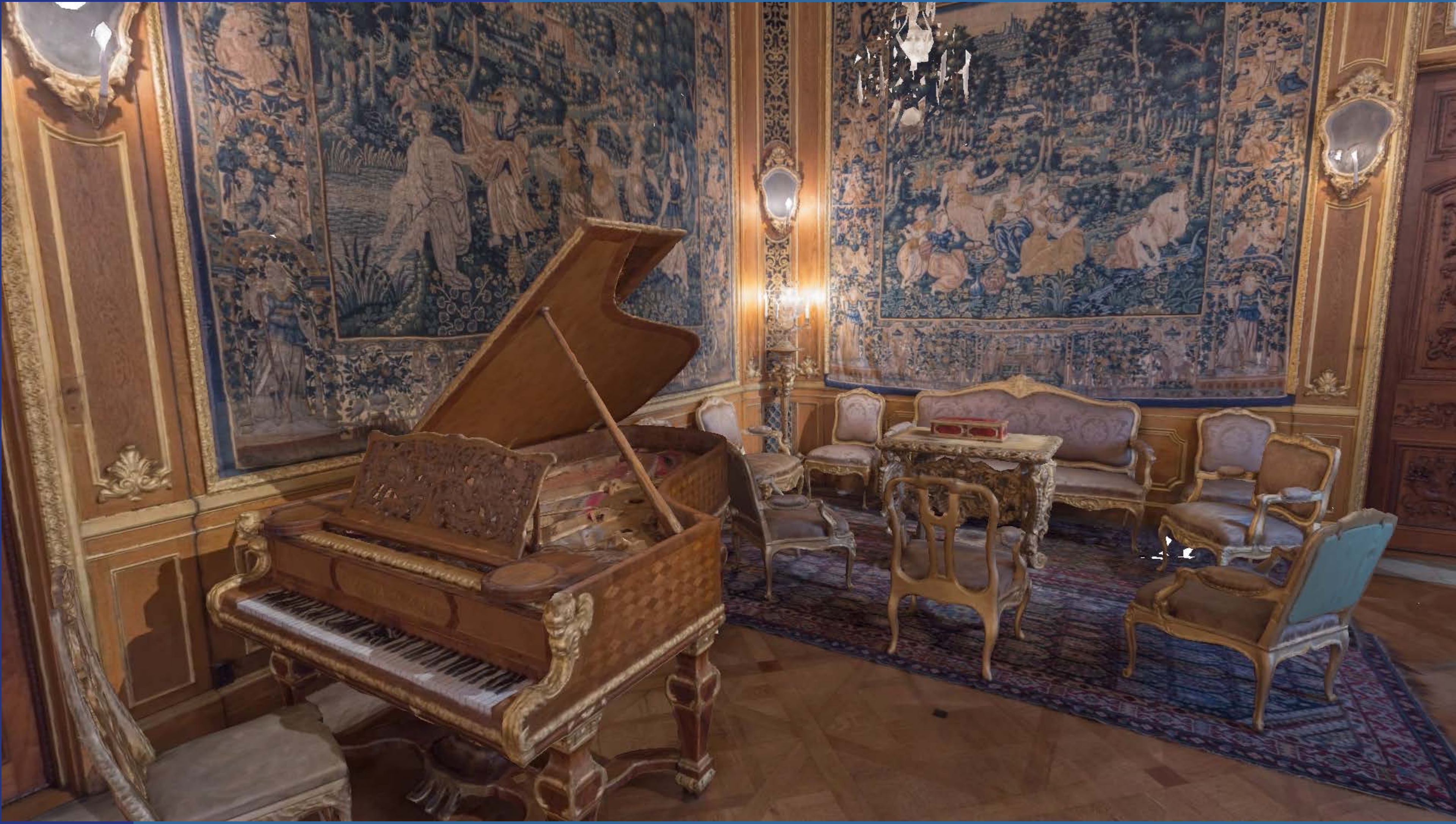


Precomputed Light fields

Max Payne 2 by Remedy Entertainment

Samuli Laine and Janne Kontkanen

High-res 3D texture – Sparse Voxel DAG:s

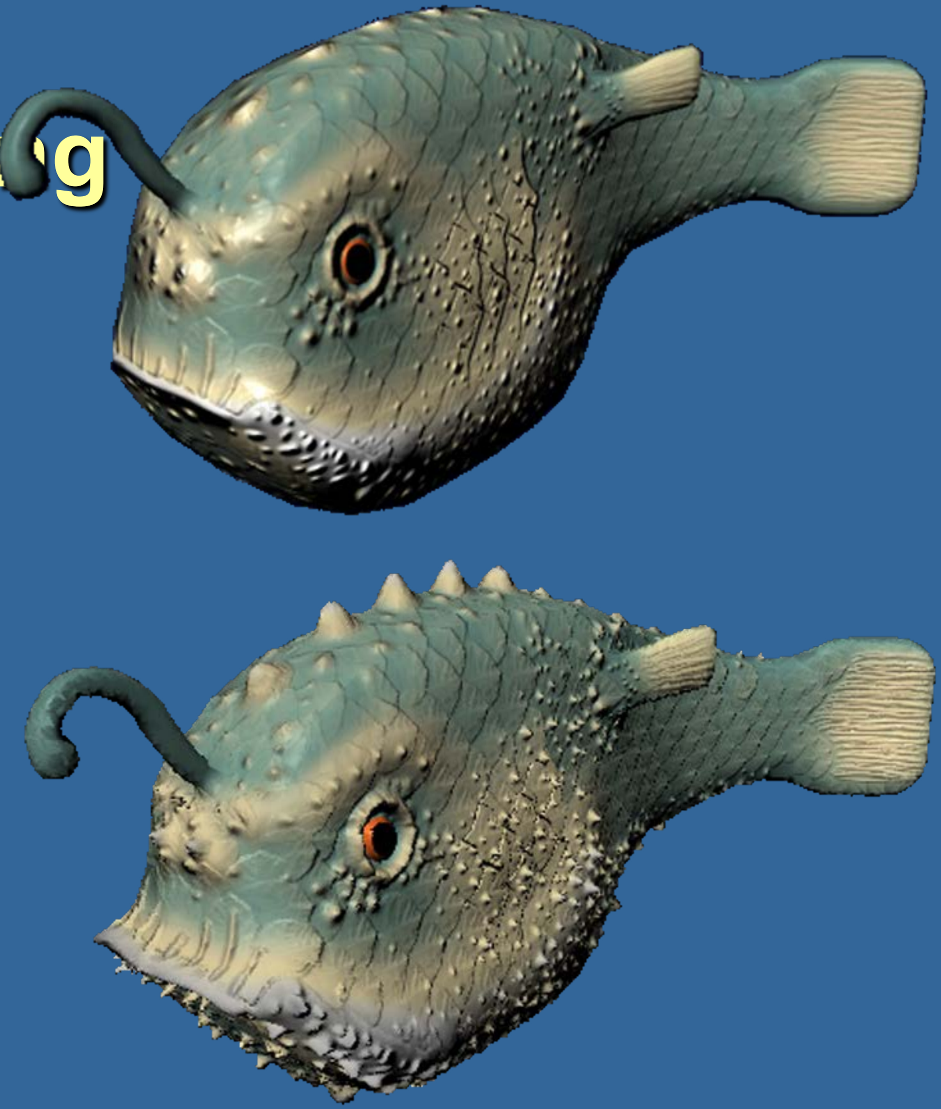


4096^3, <30 MB

Dan Dolonius, Erik Sintorn, Ulf Assarsson. *UV-free Texturing using Sparse Voxel DAGs, CGF 2020*

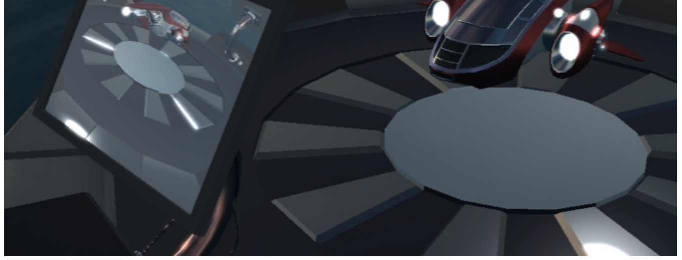
Displacement Mappi g

- Uses a map to displace the surface at each position
- Can be done with a tesselation shader



Rendering to Texture

(See also Lab 5)



```
//*******************************************************
// Create a Frame Buffer Object (FBO) that we  first render to and then use as a texture
//*******************************************************
glGenFramebuffers(1, &frameBuffer);                          // generate framebuffer id
glBindFramebuffer(GL_FRAMEBUFFER, frameBuffer);     // following commands will affect ”frameBuffer”

// Create a texture for the frame buffer, with specified filtering, rgba-format and size
glGenTextures( 1, &texFrameBuffer );
glBindTexture( GL_TEXTURE_2D, texFrameBuffer );          // following commands will affect ”texFrameBuffer”
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
glTexImage2D( GL_TEXTURE_2D, 0, 4, 512, 512, 0, GL_RGBA, GL_UNSIGNED_BYTE, NULL );

// Create a depth buffer for our FBO
glGenRenderbuffers(1, &depthBuffer);                    // get the ID to a new Renderbuffer
glBindRenderbuffer(GL_RENDERBUFFER, depthBuffer);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, 512, 512);

// Set rendering of the default color0-buffer to go into the texture
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
                       texFrameBuffer, 0);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER,
depthBuffer); // Associate our created depth buffer with the FBO
```

Or simply render to back-buffer and copy into texture using command: glCopyTexSubImage (). But is slower.

Drawing to several buffers at once in fragment shader

Fragment shader can draw to several buffers at once:

OpenGL CPU-side:

// specify an array of the color buffers you want to use

```
const GLenum buffers[] = {GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1,
                         GL_COLOR_ATTACHMENT2, GL_COLOR_ATTACHMENT3};
```

// give the array to OpenGL

```
glDrawBuffers(4, buffers);
```

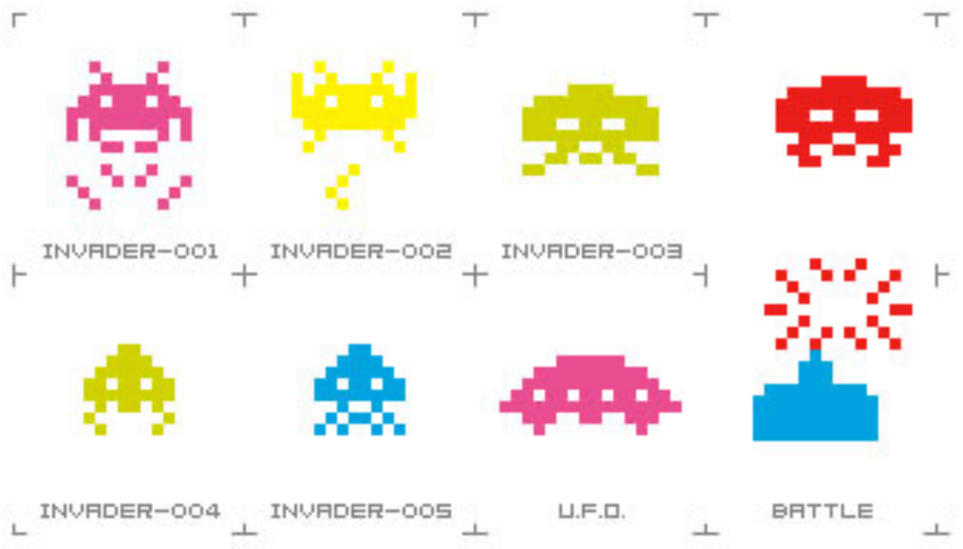
In the fragment shader:

```
layout(location = 0) out vec4 diffuseColor;
layout(location = 1) out vec4 specularColor;
layout(location = 2) out vec3 normal;
layout(location = 3) out vec3 position;
```

Sprites

Sprites (=älvor) was a technique on older home computers, e.g. VIC64. As opposed to billboards, sprites do not use the frame buffer. They are rasterized directly to the screen using a special chip. (A special bit-register also marked colliding sprites.)

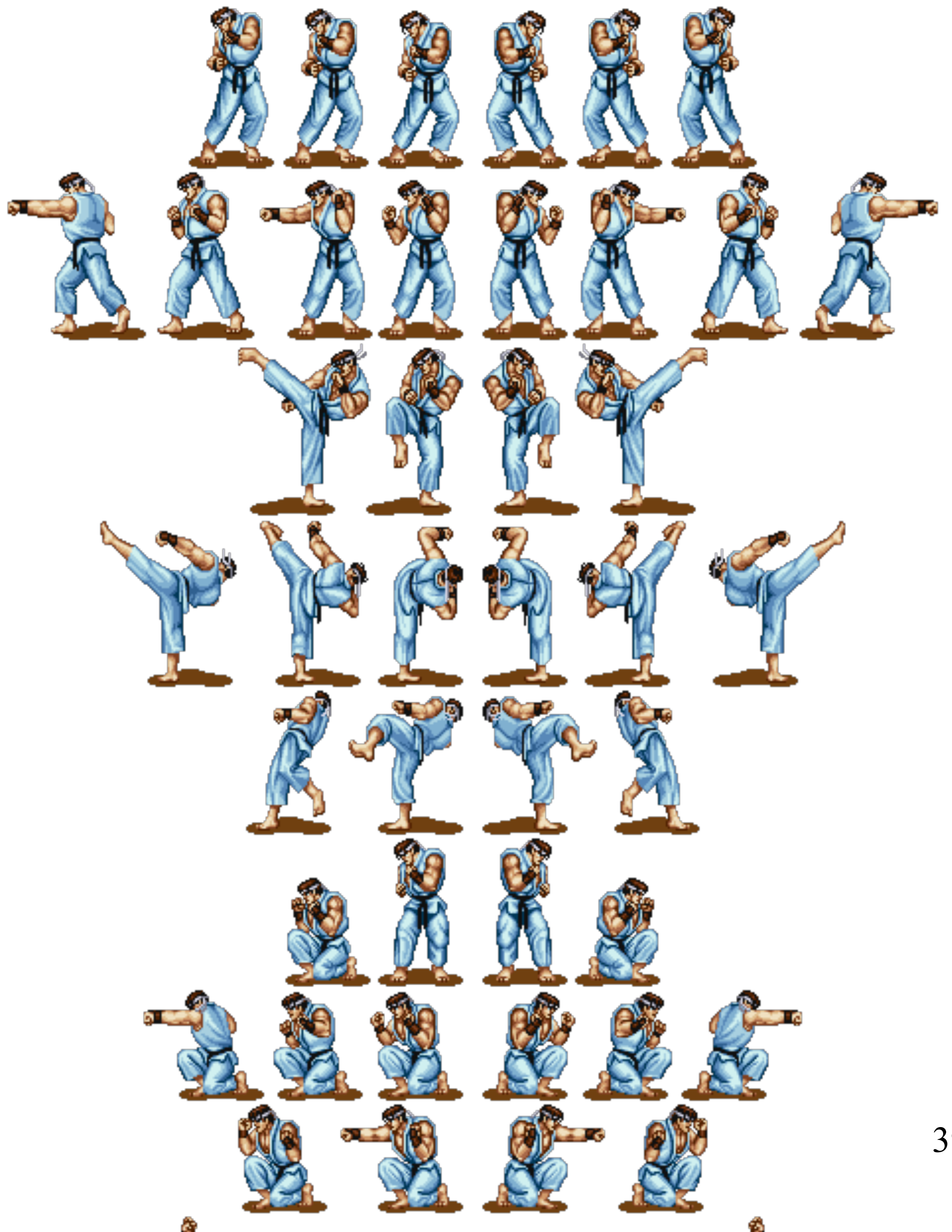
```
GLbyte M[64]=
{   127,0,0,127, 127,0,0,127,
    127,0,0,127, 127,0,0,127,
    0,127,0,0,   0,127,0,127,
    0,127,0,127, 0,127,0,0,
    0,0,127,0,   0,0,127,127,
    0,0,127,127, 0,0,127,0,
    127,127,0,0, 127,127,0,127,
    127,127,0,127, 127,127,0,0};

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);
    SDL_GL_SwapWindow //"Swap buffers"
}
```



Sprites

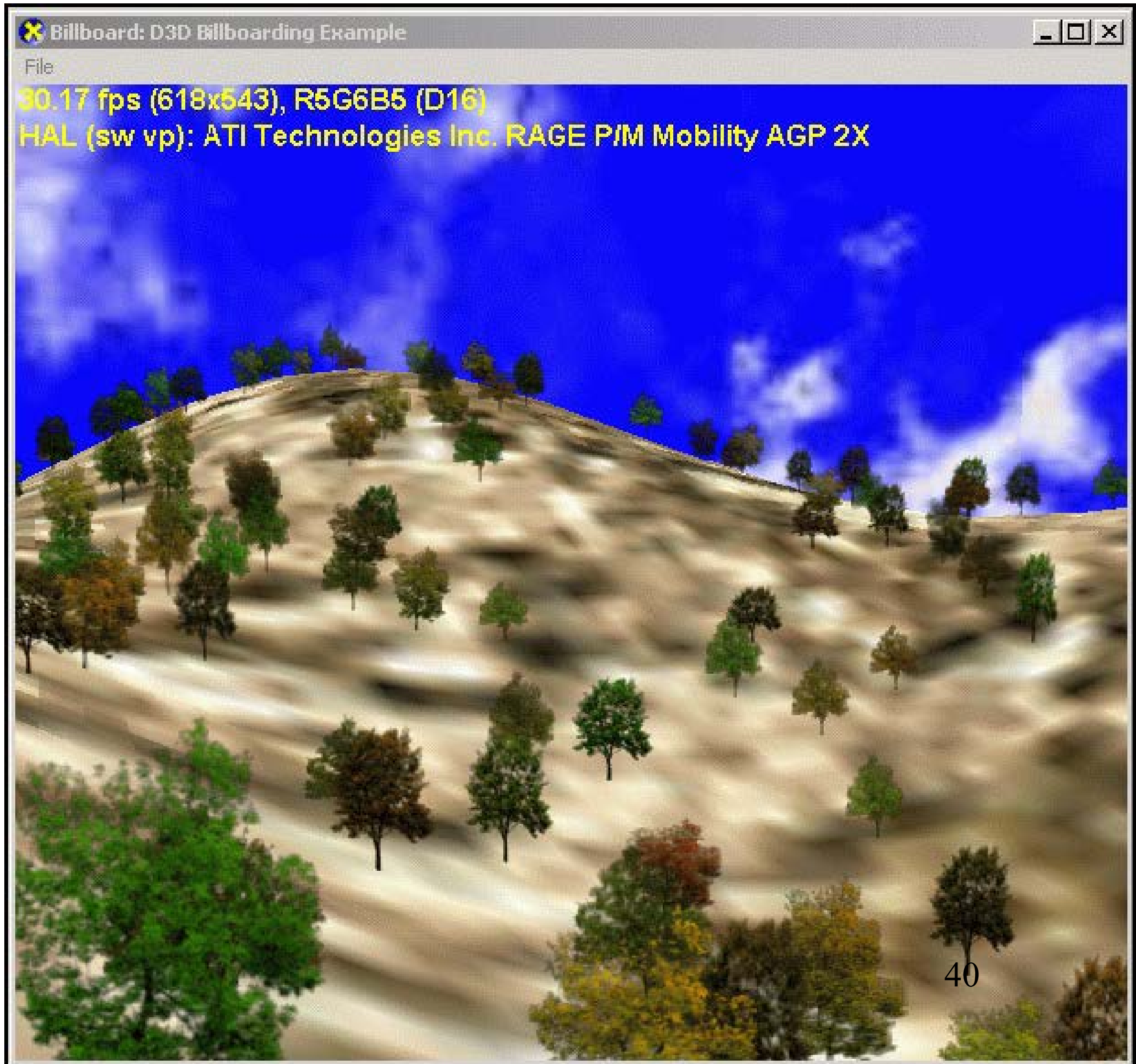
Animation Maps

The sprites for Ryu in Street Fighter:



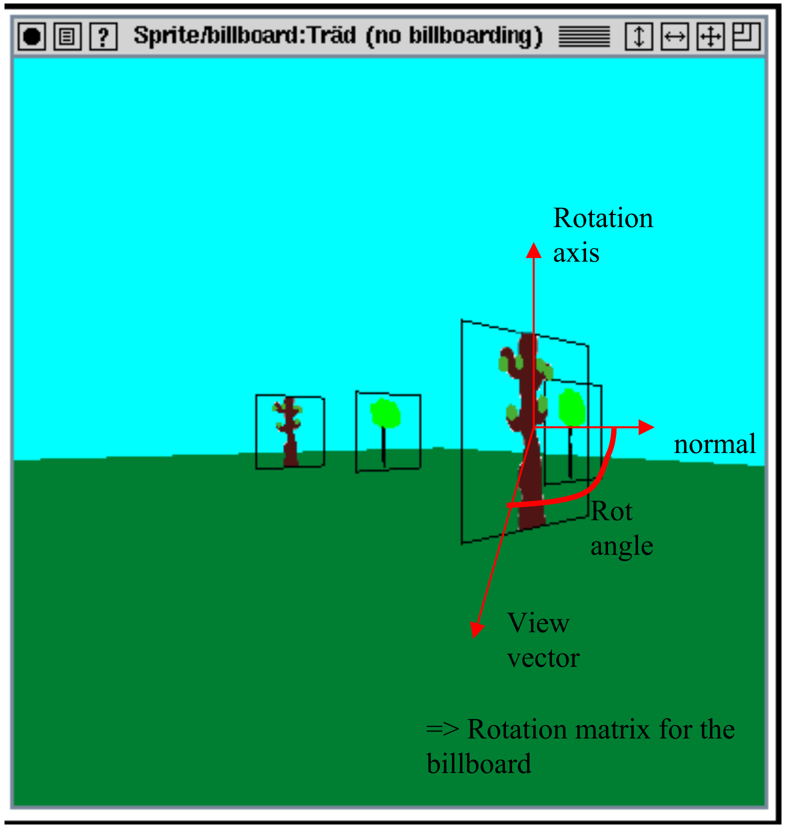
Billboards

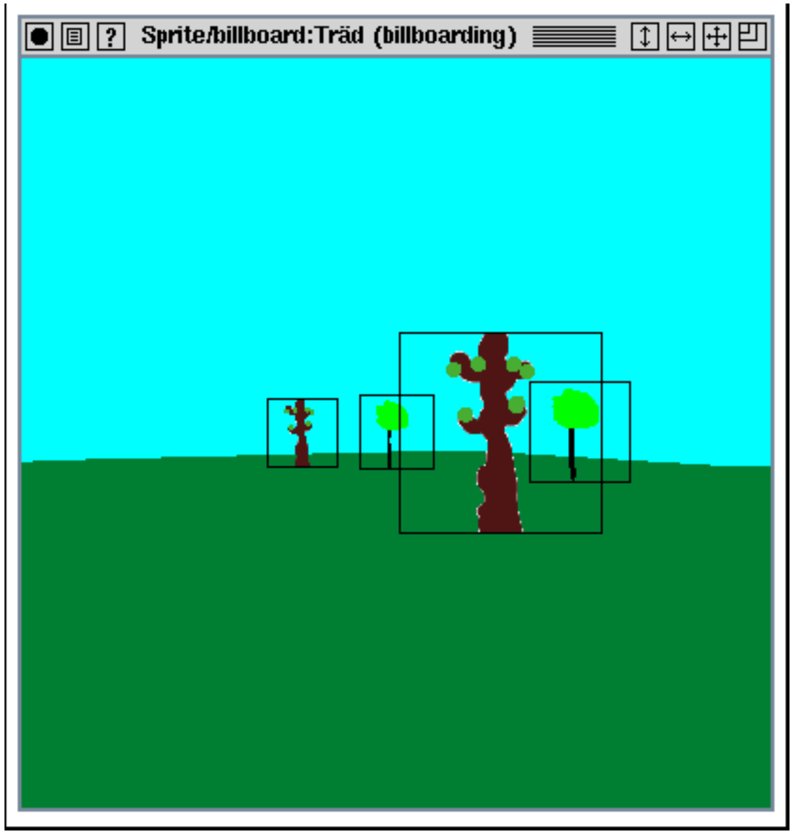
- 2D images used in 3D environments
  - Common for trees, explosions, clouds, lens-flares



30.17 fps (618x543), R5G6B5 (D16)
HAL (sw vp): ATI Technologies Inc. RAGE P/M Mobility AGP 2X

Billboards


Sprite/billboard:Träd (no billboarding)
Rotation axis
normal
Rot angle
View vector
=> Rotation matrix for the billboard

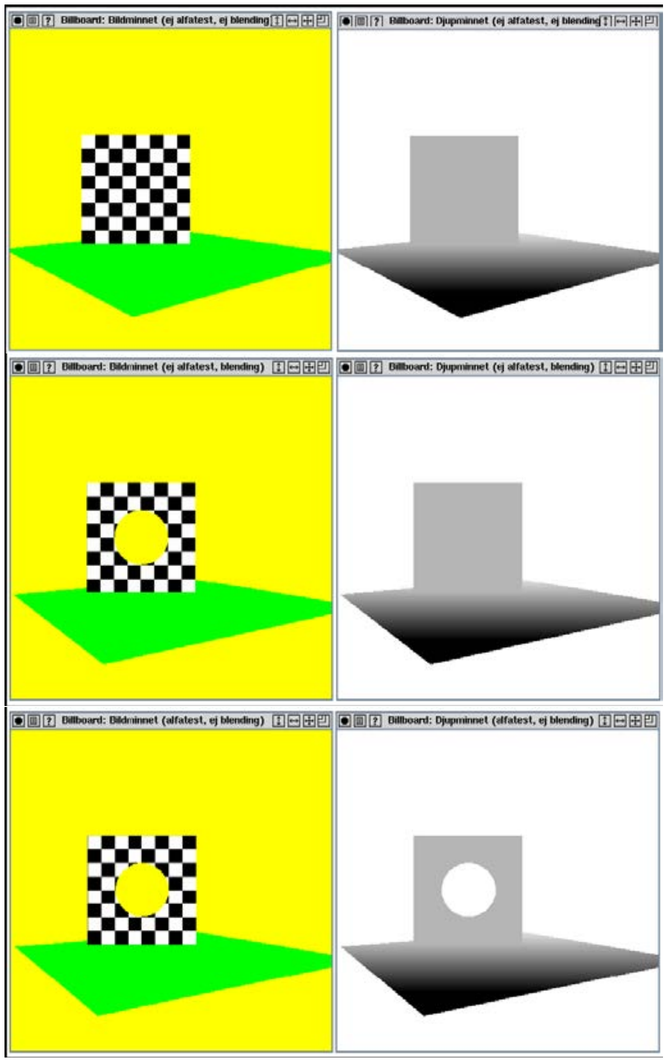


- Rotate them towards viewer
  - Either by rotation matrix, or
  - by orthographic projection

Billboards

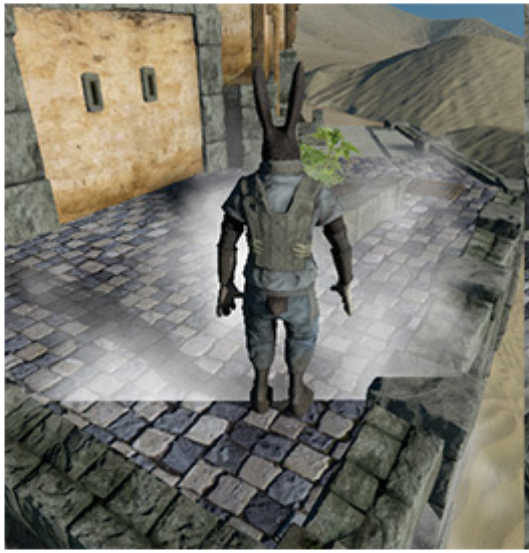
- Fix correct transparency by blending AND using alpha-test
  - In fragment shader:

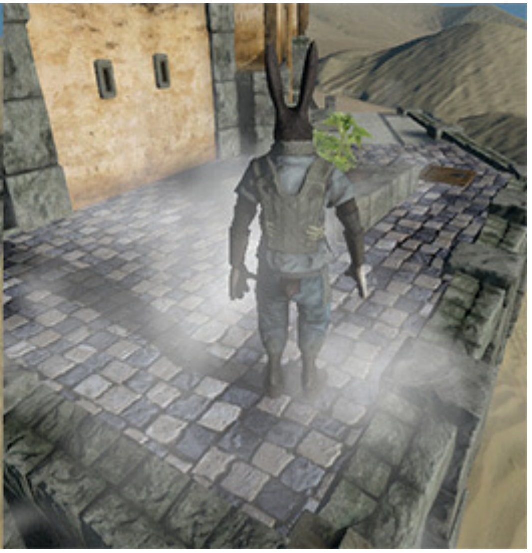
    `if (color.a < 0.1) discard;`
- Or: sort back-to-front and blend
  - (Depth writing could then be disabled to gain speed)
    - glDepthMask(0);

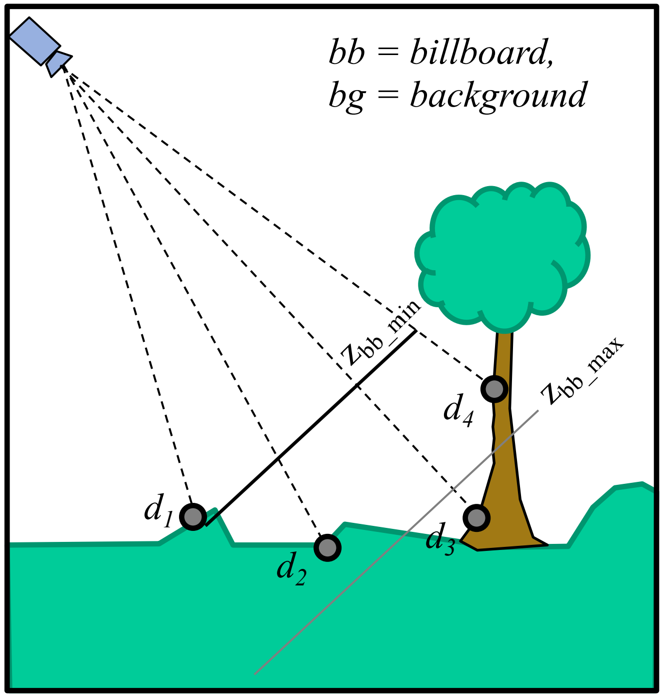

Color Buffer
Depth Buffer
Billboard: Bildminnet (ej alfatest, ej blending)
Billboard: Djupminnet (ej alfatest, ej blending)
Billboard: Bildminnet (ej alfatest, blending)
Billboard: Djupminnet (ej alfatest, blending)
Billboard: Bildminnet (alfatest, ej blending)
Billboard: Djupminnet (alfatest, ej blending)
With only blending
With alpha test

Soft Particles

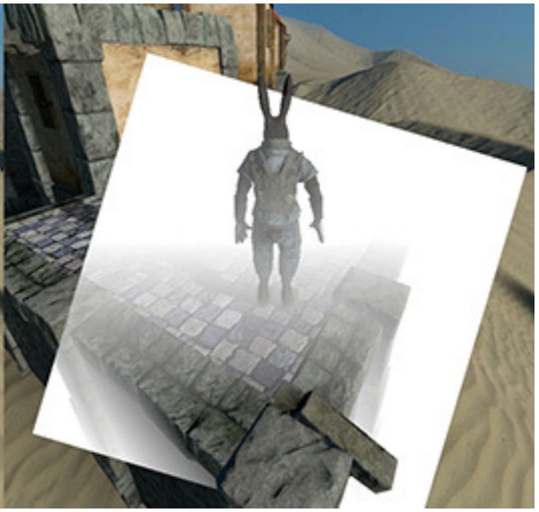
BONUS


Normal billboard


Soft Particle



Billboard's mid depth


Blending billboard and background color, based on depth difference. The whiter, the more of billboard color.

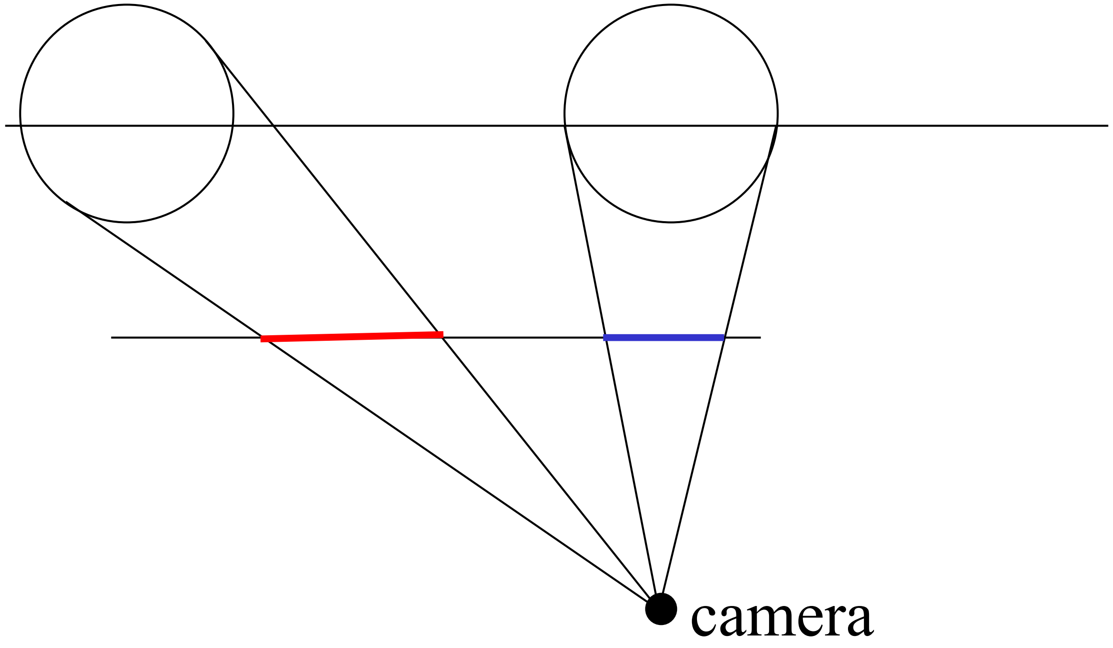
d1 is negative (in front of billboard's z range) so the standard depth test kills the billboard's fragment; at d2 and d4, the particle blends with the background; in d3 the fragment is opaque.

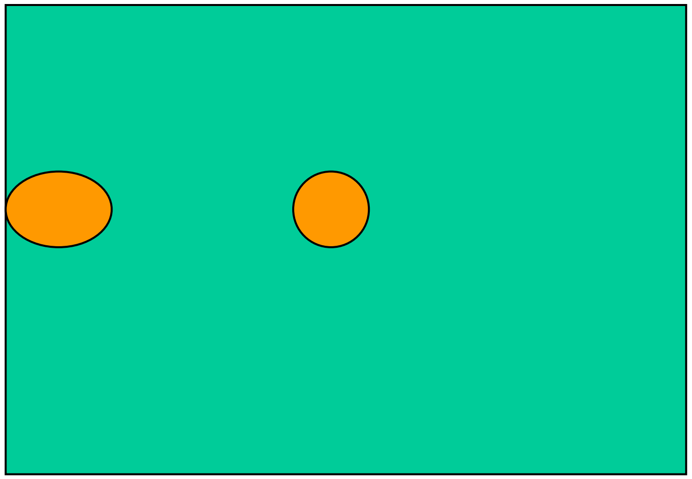
$d = (z_{bg} - z_{bb\_min}) / (z_{bb\_max} - z_{bb\_min})$;
f =smooth(d,0,1); // clamp smoothly [0,1]
c = f $\mathbf{c}_{bb}$ + (1-f) $\mathbf{c}_{bg}$; // blending bg and bb

http://blog.wolfire.com/2010/04/Soft-Particles

Perspective distortion

- Spheres often appear as ellipsoids when located in the periphery. Why?

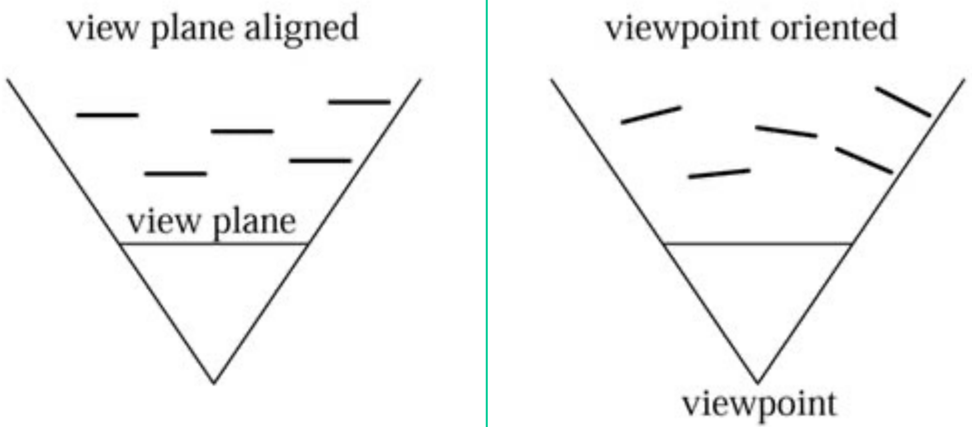




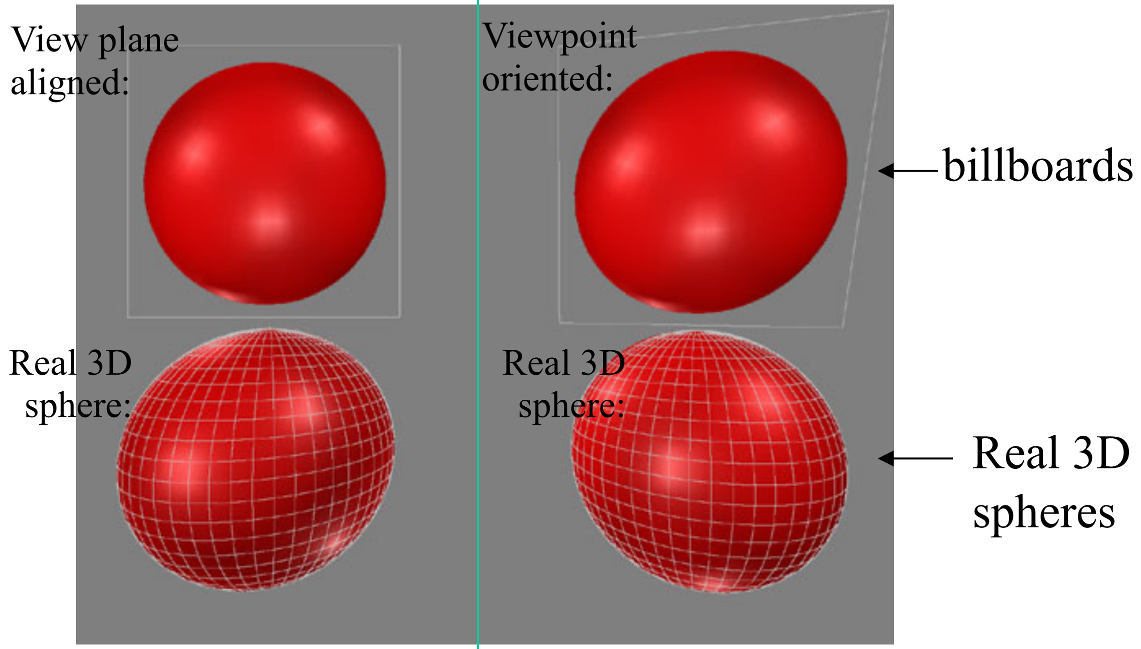
Exaggerated example

If our eye was placed at the camera position, we would not see the distortion. We are often positioned way behind the camera

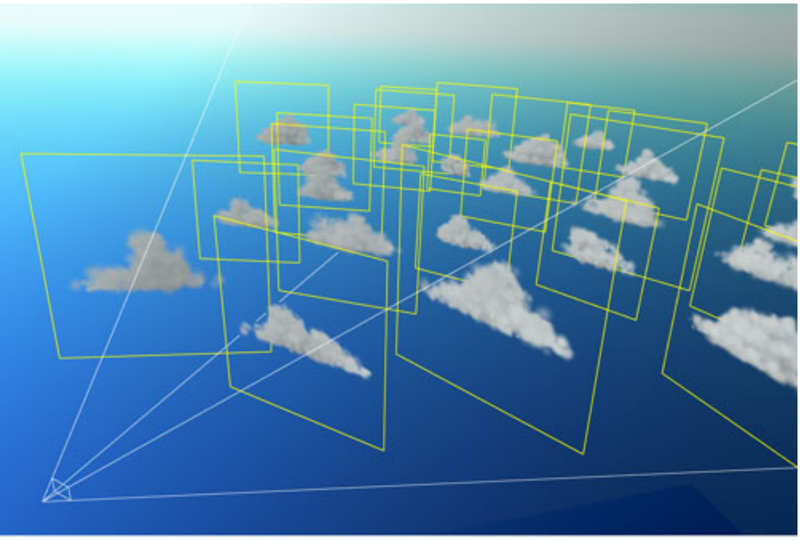
Which is preferred?



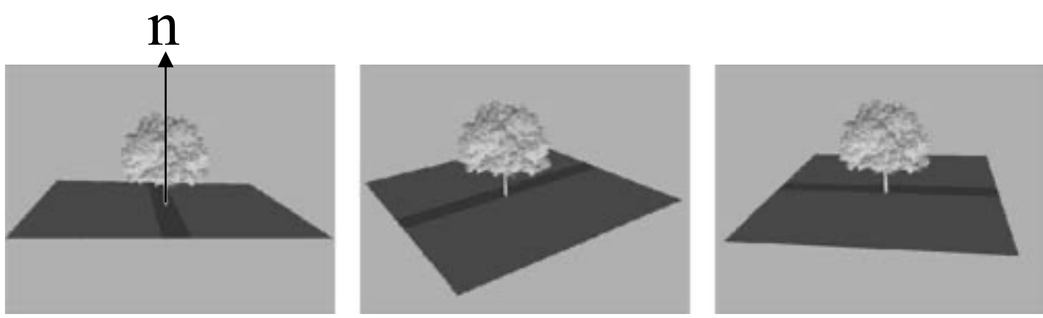
This is the result



Actually, viewpoint oriented can be preferred since it most closely resembles the result using standard 3D geometry



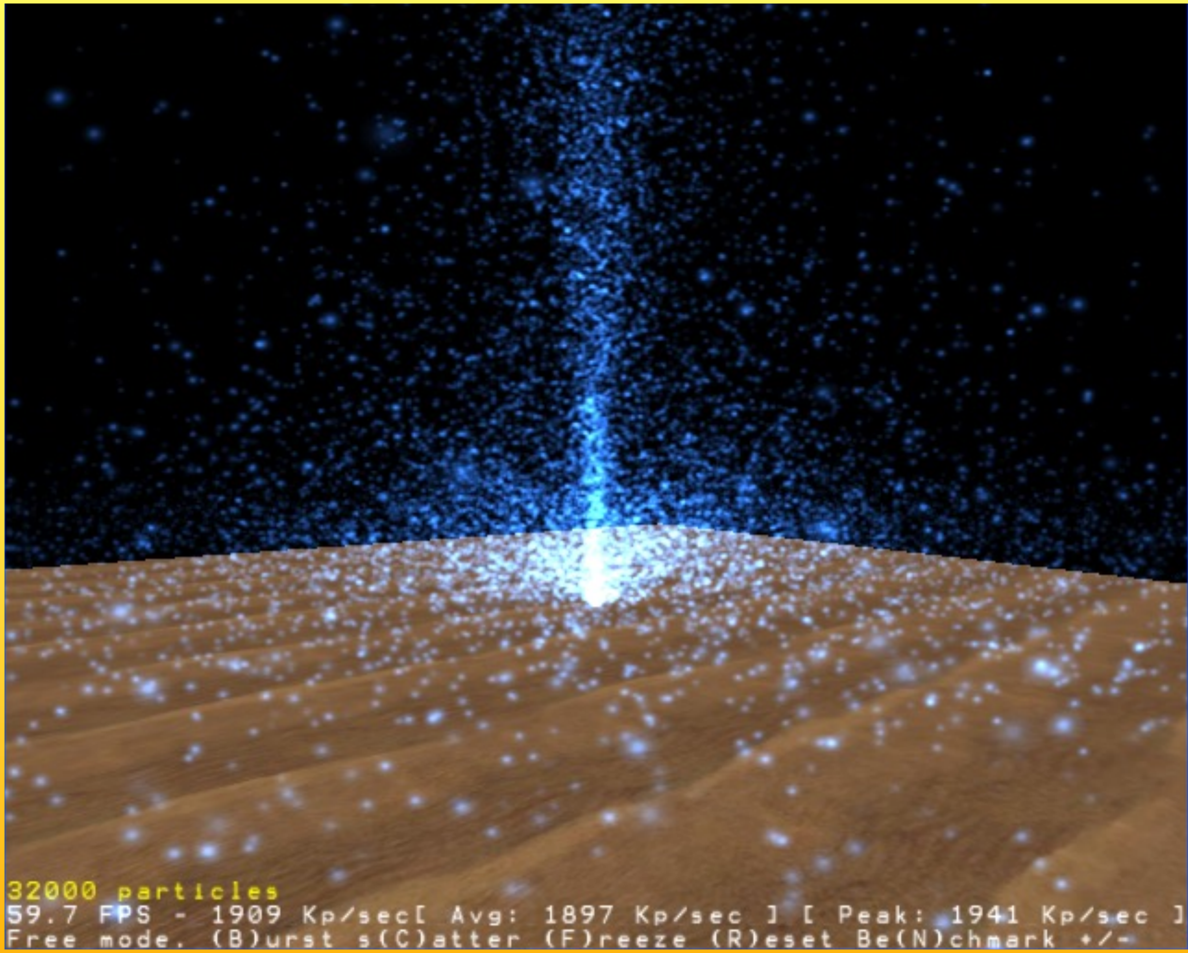
Also called *Impostors*



axial billboarding

The rotation axis is fixed and disregarding the view position

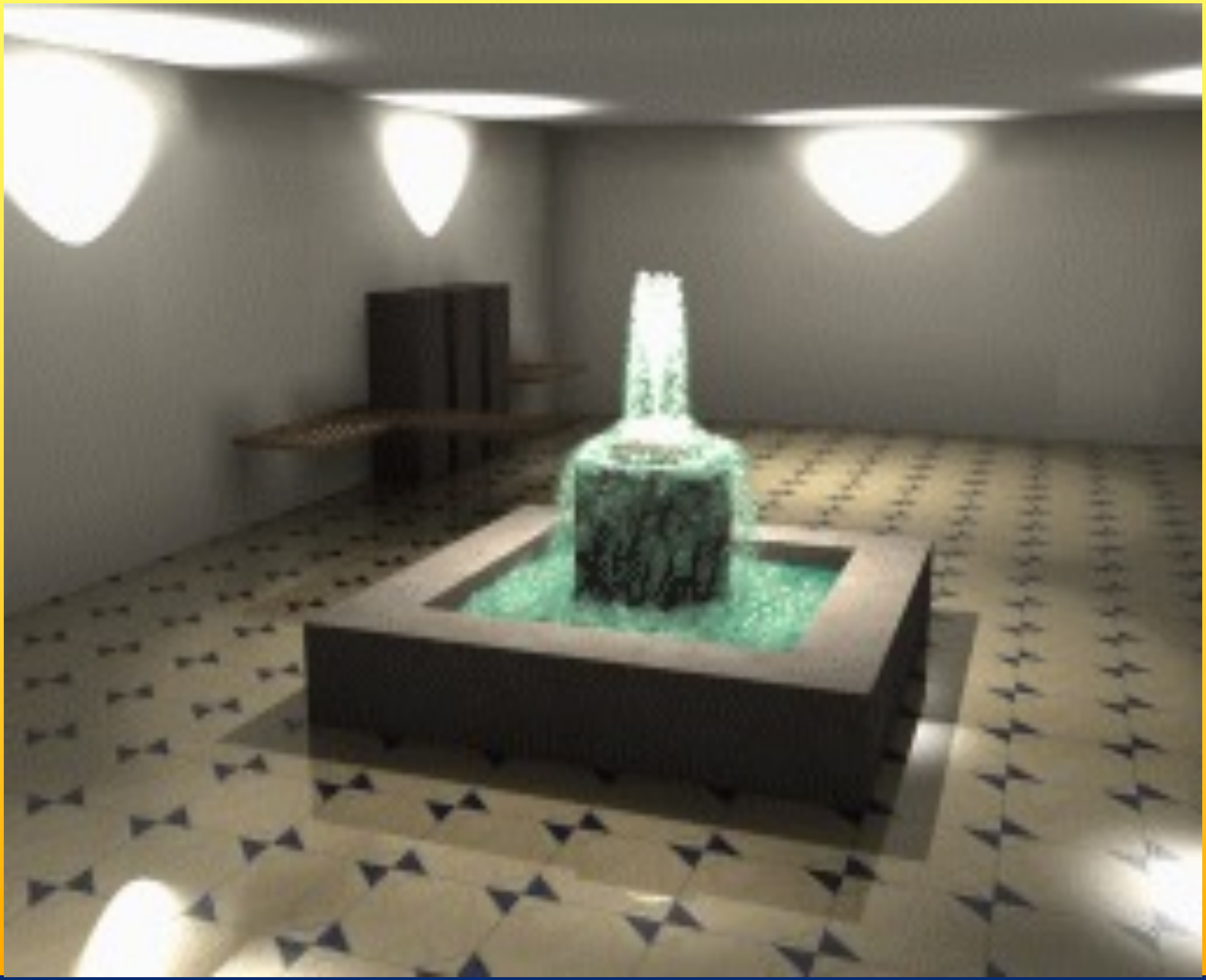
Particle system



```
32000 particles
59.7 FPS - 1909 Kp/sec[ Avg: 1897 Kp/sec ] [ Peak: 1941 Kp/sec ]
Free mode. (B)urst s(C)atter (F)reeze (R)eset Be(N)chmark +/-
```

Particles

Partikelsystem



What’s most important?

Texturing:

- Filtering:
  - Magnification – nearest neightbor, linear
  - Minification – nearest neighbor, linear, bilinear & trilinear-filtered mipmap lookup.
  - Mipmaps + their memory cost
  - How compute bilinear/trilinear filtering
  - Number of texel accesses for trilinear filtering
  - Anisotropic filtering – take several trilinear-filtered mipmap lookups along the line of anisotropy (e.g., up to 16 lookups)
- Environment mapping – cube maps. How compute lookup.
- Bump mapping
- 3D-textures – what is it?
- Sprites
- Billboards/Impostors, viewplane vs viewpoint oriented, axial billboards, how to handle depth buffer for fully transparent texels.
- Particle systems