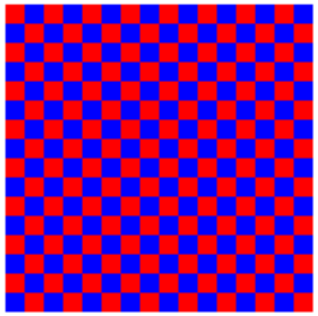
Texture Mapping II

Adding Texture Mapping to Illumination

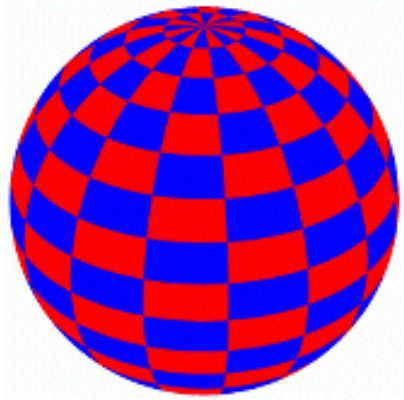
Texture mapping can be used to alter some or all of the constants in the illumination equation. We can simply use the texture as the final color for the pixel, or we can just use it as diffuse color, or we can use the texture to alter the normal, or... the possibilities are endless!

$$I_{total} = k_a I_{ambient} + \sum_{i=1}^{lights} I_i \left(k_d \left(\hat{N} \cdot \hat{L} \right) + k_s \left(\hat{V} \cdot \hat{R} \right)^{n_{shiney}} \right)$$

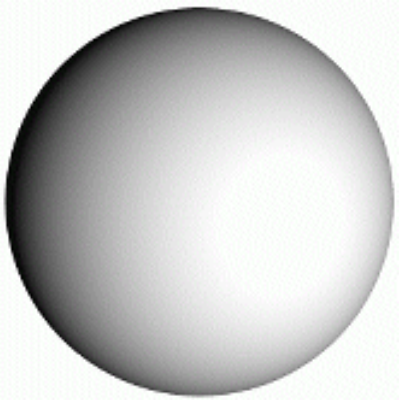
Phong's Illumination Model



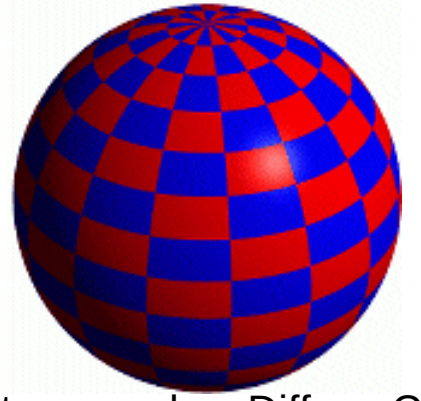
Texture



Texture used as Label



Constant Diffuse Color



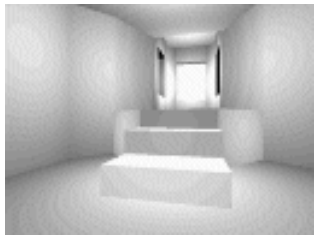
Texture used as Diffuse Color

Texture Mapping in Quake

Quake uses light maps in addition to texture maps. Texture maps are used to add detail to surfaces, and light maps are used to store pre-computed illumination. The two are multiplied together at run-time, and cached for efficiency.

	Texture Maps	Light Maps
Data	RGB	Intensity
Instanced	Yes	No
Resolution	High	Low

Light map image by Nick Chirkov.



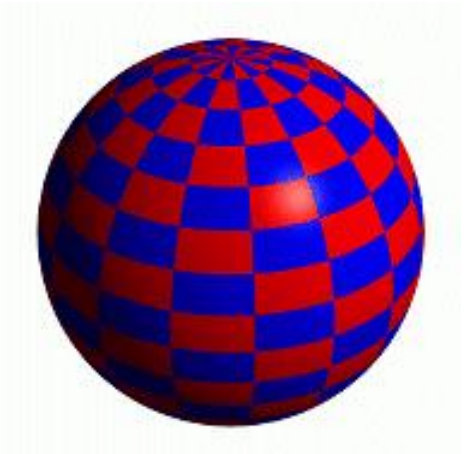
Textures Only

Textures & Light Maps

Bump Mapping

Textures can be used to alter the surface normal of an object. This does NOT change the actual shape of the surface -- we are only shading it as if it were a different shape! This technique is called *bump mapping*. The texture map is treated as a single-valued height function. The value of the function is not actually used, just its **partial derivatives**. The partial derivatives tell how to alter the true surface normal at each point on the surface to make the object appear as if it were deformed by the height function.

Bump Mapping assumes that the Illumination model is applied at every pixel (as in Phong Shading or ray tracing).



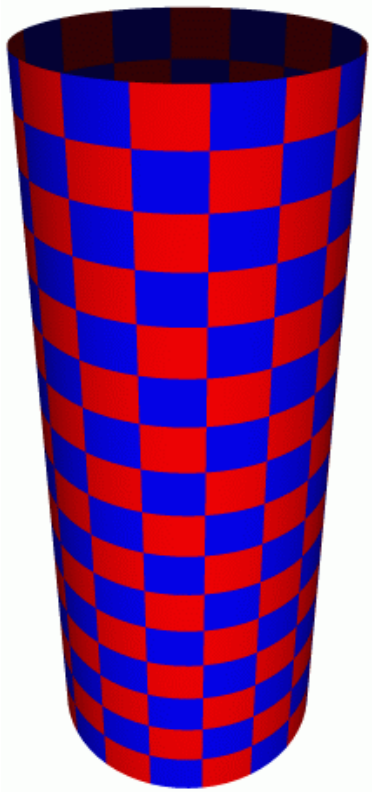
Sphere w/Diffuse Texture

Swirly Bump Map

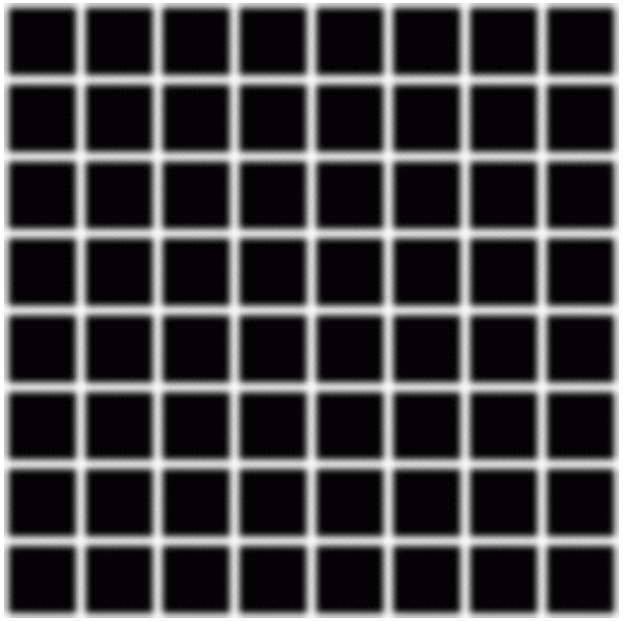
Sphere w/Diffuse Texture & Bump Map

More Bump Map Examples

Since the actual shape of the object does not change, the silhouette edge of the object will not change.



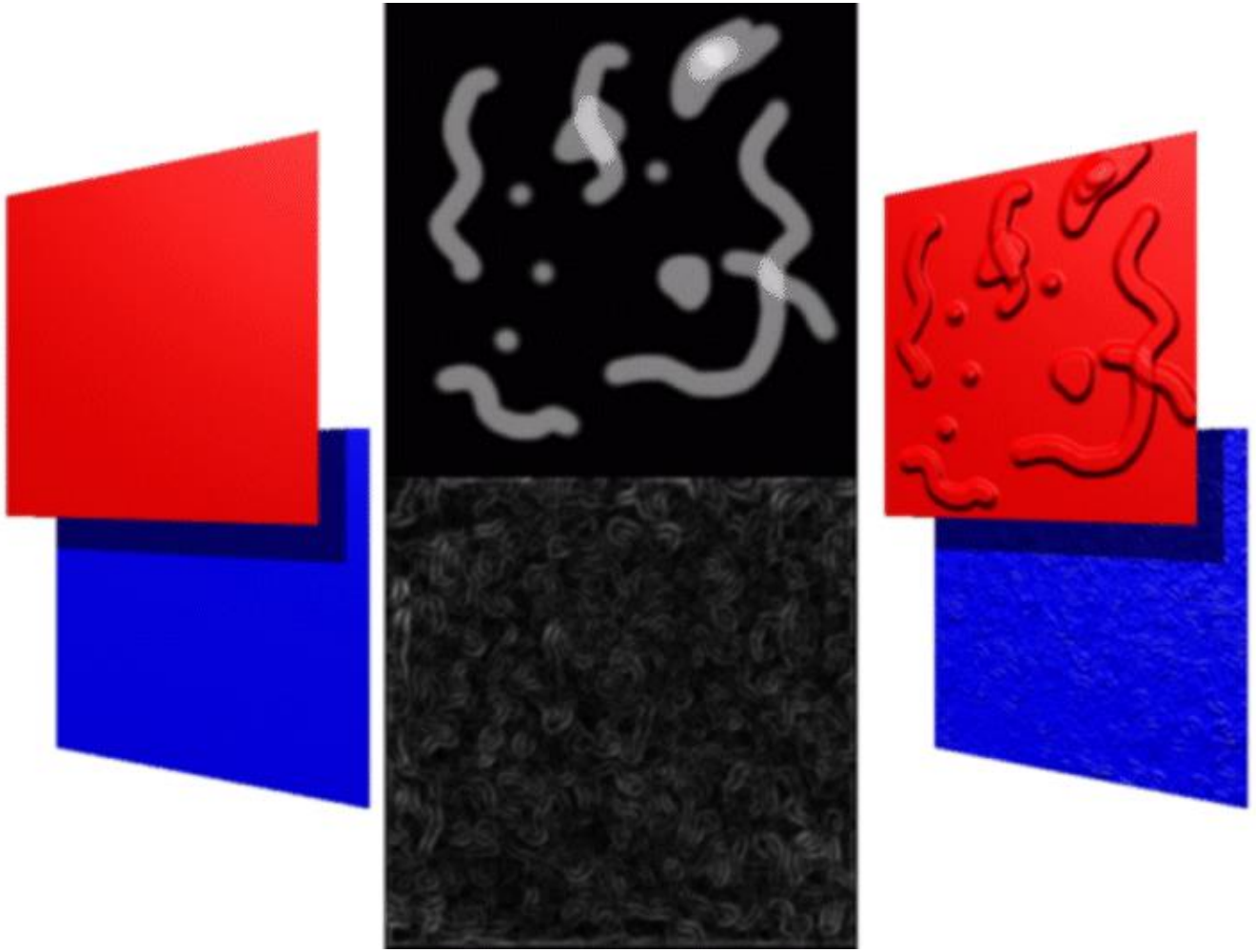
Cylinder w/Diffuse Texture Map



Bump Map

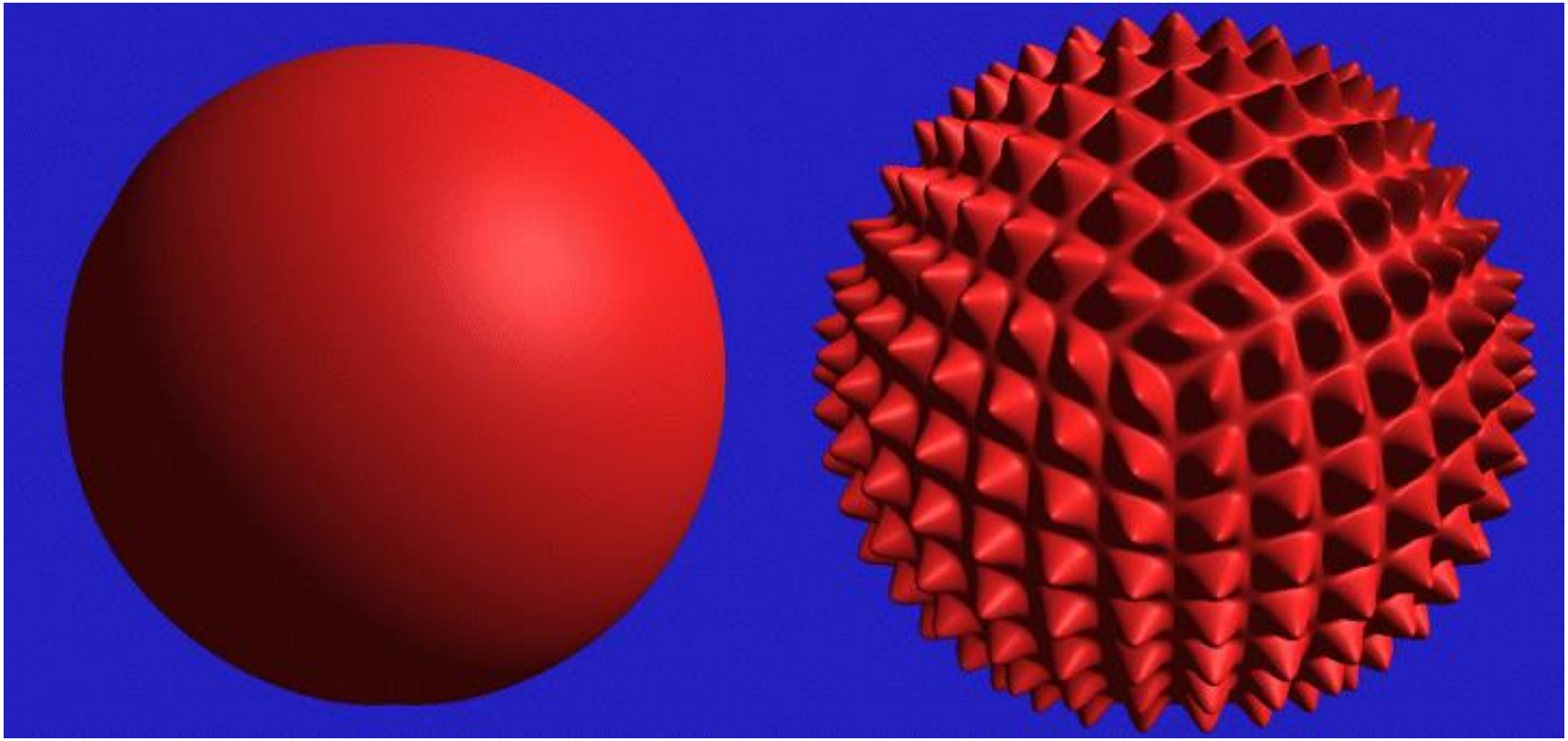
Cylinder w/Texture Map & Bump Map

One More Bump Map Example



Displacement Mapping

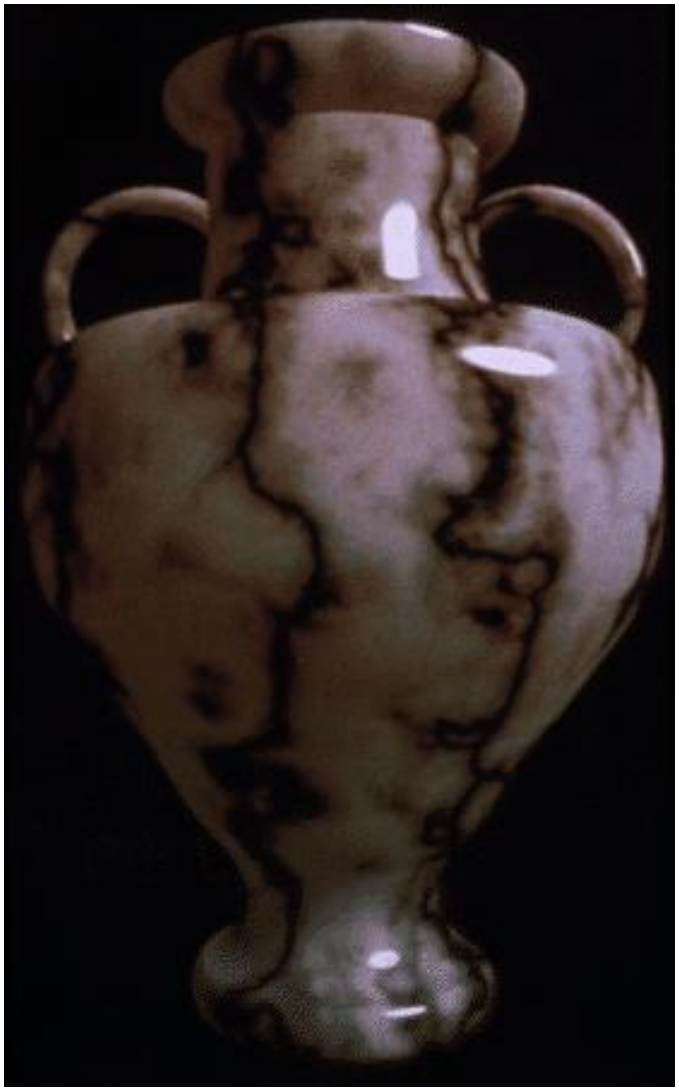
We use the texture map to actually move the surface point. This is called displacement mapping. How is this fundamentally different than bump mapping?



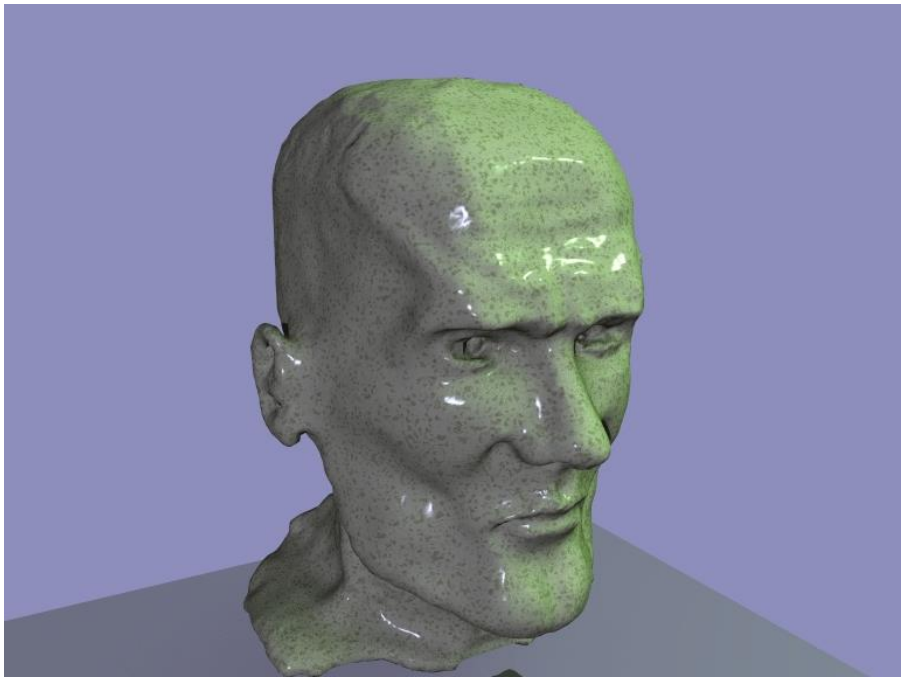
The geometry must be displaced before visibility is determined. Is this easily done in the graphics pipeline?

Three Dimensional or Solid Textures

The textures that we have discussed to this point are two-dimensional functions mapped onto two-dimensional surfaces. Another approach is to consider a texture as a function defined in a three-dimensional volume. Textures of this type are called *solid textures.* Solid textures are very effective at representing some types of materials such as marble and wood. Generally, solid textures are defined procedural functions rather than tabularized or sampled functions as used in 2D. A popular approach is based on *An Image Synthesizer, by Ken Perlin, SIGGRAPH '85. The vase to the right is from this paper.*

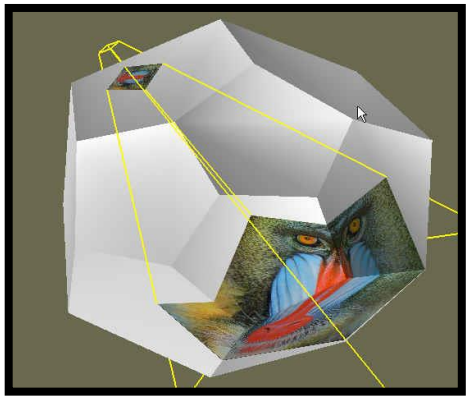


Examples



Projective Textures

- Use to simulate effects:
  - Slide projector
  - Spotlight illumination
  - Shadows
  - Reproject photograph of an object onto object geometry



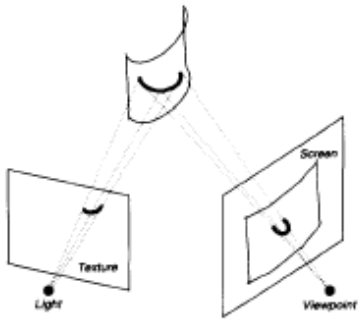


Figure 1. Viewing a projected texture.

Source: Wolfgang Heidrich [99]

Projective Textures

- Facts
  - Texture coordinates can be transformed by a matrix (e.g. a perspective projection)
  - OpenGL generalizes texture coordinates to 4-component homogenous coordinates
  - q coordinate is analogous to w
  - Texture image can be subjected to a projection independent from the viewing projection

Example Code

Here is a code fragment implementing projective textures in OpenGL

// Basically, the first group of setting says that we will not be supplying texture coordinates.
// Instead, they will be automatically established based on the vertex coordinates in “EYE-SPACE”
// (after application of the MODEL_VIEW matrix).

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, (int) GL_EYE_LINEAR);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, (int) GL_EYE_LINEAR);
glTexGeni(GL_R, GL_TEXTURE_GEN_MODE, (int) GL_EYE_LINEAR);
glTexGeni(GL_Q, GL_TEXTURE_GEN_MODE, (int) GL_EYE_LINEAR);
```

// These calls initialize the TEXTURE_MAPPING function to identity. We will be using
// the Texture matrix stack to establish this mapping indirectly.

```
float [] eyePlaneS = { 1.0f, 0.0f, 0.0f, 0.0f };
float [] eyePlaneT = { 0.0f, 1.0f, 0.0f, 0.0f };
float [] eyePlaneR = { 0.0f, 0.0f, 1.0f, 0.0f };
float [] eyePlaneQ = { 0.0f, 0.0f, 0.0f, 1.0f };
glTexGenfv(GL_S, GL_EYE_PLANE, eyePlaneS);
glTexGenfv(GL_T, GL_EYE_PLANE, eyePlaneT);
glTexGenfv(GL_R, GL_EYE_PLANE, eyePlaneR);
glTexGenfv(GL_Q, GL_EYE_PLANE, eyePlaneQ);
```

Example Code

Here is where the extra “Texture” transformation on the vertices is inserted.

glMatrixMode(GL_TEXTURE);

glLoadIdentity();

glTranslated(0.5, 0.5, 0.5); // Scale and bias the [-1,1] NDC values

glScaled(0.5, 0.5, 0.5); // to the [0,1] range of the texture map

gluPerspective(?, ?, ?, ?); // projector "projection" and view matrices

gluLookAt(lightPosition_x,lightPosition_y,lightPosition_z, At_x,At_y,At_z, Up_x,Up_y,Up_z);

glMultMatrixf((GLfloat *) M_Inverse); //M_Inverse is the inverse of the concatenation of the ModelView and Projection matrices

glMatrixMode(GL_MODELVIEW);

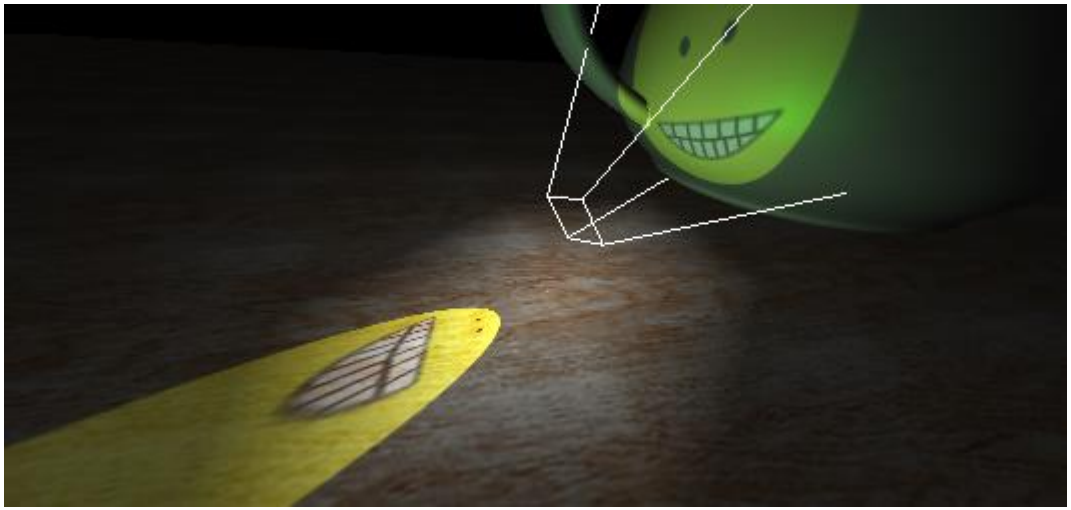
Useful function:

glGetFloatv();

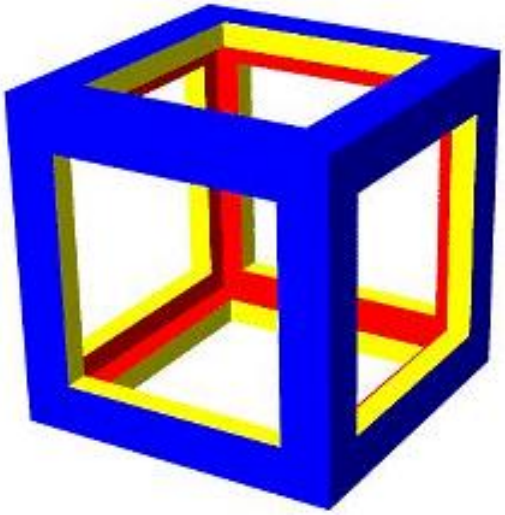
http://www.opengl.org/developers/code/glut_examples/advanced/projtex.c

Artifacts

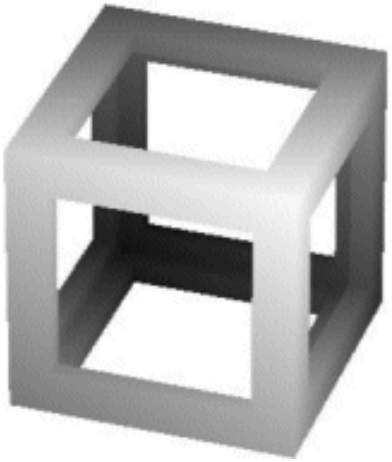
- Projective texture mapping produces a reverse projection as well
- Projective texture mapping 'penetrates through' objects

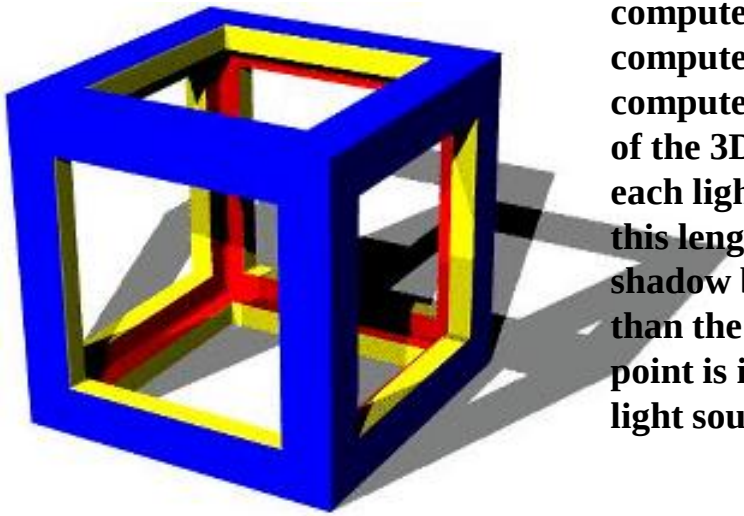


Shadow Maps



Textures can also be used to generate shadows. First, the scene is rendered from the point of view of each light source, but only the depth-buffer values are retained. In this example the closer points are lighter and more distant parts are darker (with the exception of the most distant value which is shown as white for contrast).





As each pixel is shaded (once more shadow mapping assumes that the illumination model is applied at each pixel) a vector from the visible point to the light source is computed (Remember it is needed to compute, $N \cdot L$). As part of normalizing it we compute its length. If we find the projection of the 3D point that we are shading onto each lights shadow buffer we can compare this length with the value stored in the shadow buffer. If the shadow-buffer is less than the current point's length then the point is in shadow and the corresponding light source can be ignored for that point.