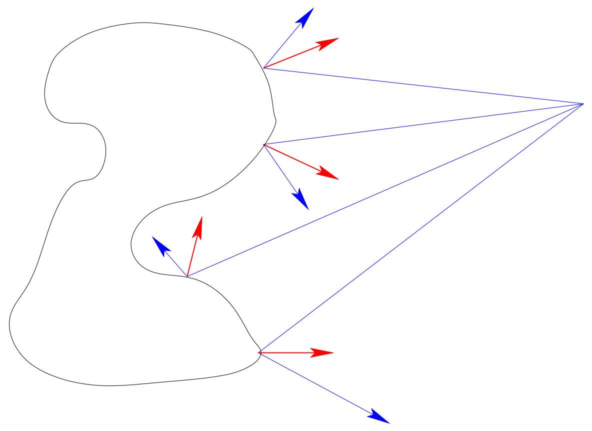


Figure 1: Eye rays reflected from an object. The reflected rays are shown as blue arrows. Red arrows are the normal vectors. Given the normal vectors, one can compute the reflected rays using the formulas we used to do Phong shading. Note that environment mapping will not produce correct results for rays which have to bounce from the object several times before escaping to the environment (like one of the rays shown above).

Environment Mapping

1 Geometry of reflection

Consider an ideally reflective object and a ray through a pixel. What color should one use for that pixel? The same as the color incoming to the reflection point from the direction of the reflected ray (see Figure 1). Therefore, if one is able to precompute and somehow store the colors incoming towards the reflective object along different rays, rendering would be really simple.

In environment mapping, one stores the colors arriving at a 3D object from different directions in one or more textures. Typically, one assumes that colors incoming to different points along parallel rays are the same. This means that we need to keep a color for each incoming ray *direction* rather than for every possible incoming ray. Below we discuss two specific examples.

2 Spherical environment map

Imagine a photograph (or a synthetic image, e.g. rendered using ray tracing) of a mirror sphere. This image stores colors incoming along rays of all possible directions (Figure 2). It is directly used as texture for spherical environment mapping. Such a texture might look like the one shown in Figure 3. To simplify things, let's say that the sphere is exactly inscribed into the (square) photograph.

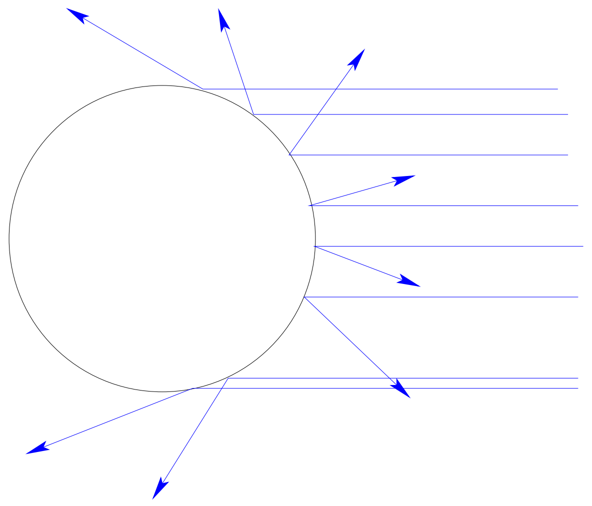
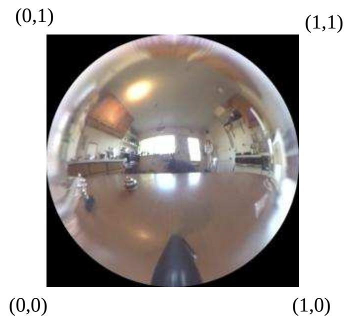


Figure 2: A mirror sphere. Any direction is a direction of a certain reflected horizontal ray. Spherical environment mapping is based on mapping colors from a photograph of a reflective ball onto a 3D object. If the photograph is taken from infinite distance, colors incoming from all directions are present on that photograph. Of course, there are two caveats here: (1) the distance between the viewpoint and the sphere is finite – a typical photograph is a perspective image rather than a parallel projection based image; in particular, some directions are not represented on the image; we'll just sweep this problem under the carpet here, saying that if the photograph is taken from far away, the projection is close to parallel; (2) the photograph will not really store colors for every ray, but only at rays through pixels and the density of sampling of the space of directions will not be uniform; we'll just hope that interpolation will solve this problem.



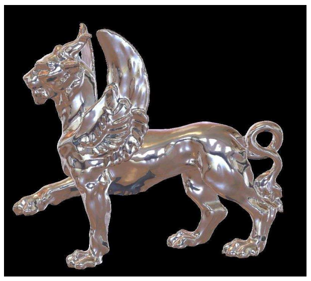


Figure 3: An example of a texture for use with spherical environment mapping and a 3D model with this spherical environment mapping.

To see what texture coordinates we need to use with spherical environment mapping, notice that the direction of the reflected ray (assuming that the viewing direction is the same as the direction from which the image of the sphere was taken and the viewpoint is far away, so that the rays from the viewpoint are almost parallel) depends solely on the normal vector at the point where the ray hits the object. Therefore, the color at a vertex v of the object should be the same as the color we see at the point of the sphere which has the same normal vector. Assuming that the sphere is centered at the origin and has unit radius and the unit normal at v has coordinates $[n_x, n_y, n_z]$, we need to use the color we see at the point (n_x, n_y, n_z), which projects (assuming that the projection direction is along the z-axis) to (n_x, n_y). Of course, the range for n_x and n_y is $[-1, 1]$ and the range of texture coordinates is $[0, 1]$, and therefore we need to scale n_x and n_y so that they are in $[0, 1]$. To do this, we add one to each coordinate and then divide it by 2. To sum up, the texture coordinates we need to use for a vertex v with unit normal $[n_x, n_y, n_z]$ with spherical environment map are

$$\left[\frac{1+n_x}{2}, \frac{1+n_y}{2}\right].$$

The scaling can be elegantly handled using the *texture transformation matrix*. In graphics pipeline, there are 4 texture coordinates which can be transformed using 4×4 matrices much like vertex coordinates (eventually, the first two texture coordinates are used in the case of 2D textures). Using texture matrices one can also apply the environment map to a rotating object: the trick is to use normals as texture coordinates and use the texture matrix to rotate the texture coordinates in the same way as the vertices are rotated and to scale them to $[0, 1]$.

Graphics APIs usually provide various ways of generating texture coordinates automatically based on viewpoint, normals, light location and other parameters. Automatic texture generation means that the programmer does not have to specify them explicitly but they are computed in some specific way from other vertex attributes or view or light data. Automatic texture generation can handle most of the 'natural' scenarios of using texture to model lighting or reflections (in particular, environment mapping).

3 Cubical

In cubical environment mapping, one first records colors incoming towards a point P somewhere inside or near the object from every direction $\vec{d}$. We will think of this color as being painted onto the cube's surface at the point where the ray that starts at P and has $\vec{d}$ as the direction vector intersects the cube's surface (Figure 4). Cube environment map can be thought of as six images (which we obtain on the faces of the cube by applying the above procedure). If just one texture is desired, one can cut the cube along some edges and flatten it onto the plane as shown in Figure 5.

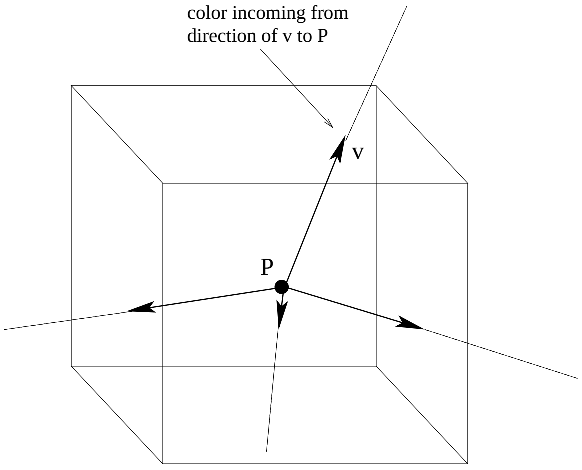


Figure 4: Cubical texture mapping.

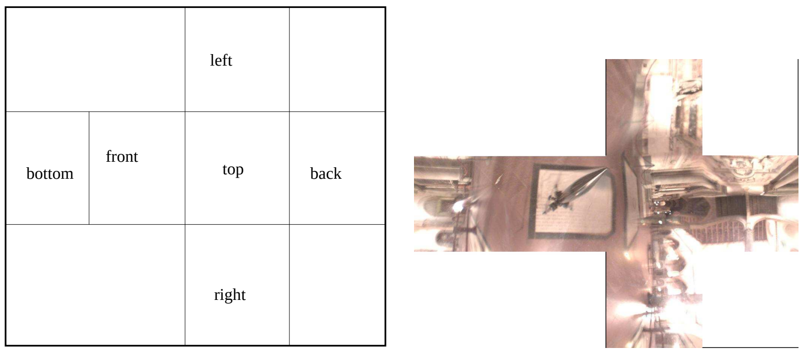


Figure 5: Flattening the cube environment map.

Cubical environment map is more flexible (there is no problem with applying it to views from different directions, for example) but it its hardware implementation is slightly more complicated. One possible scenario is as follows. For each vertex, use texture coordinates equal to the coordinates of a unit vector pointing in the direction symmetric to the view vector about the normal of that vertex (this is the direction of the incoming color we want to use). Send this texture coordinate to the rasterizer and let it linearly interpolate to fragments. At the fragment processing stage, convert the interpolated normal to 'real' texture coordinates (in the texture space) and look up color from texture. The conversion of normal to coordinates in the image basically maps that normal to the corresponding point on the cube and finds the corresponding point on the texture. Formulas for this conversion are not hard to find.