

Texture mapping

1 What is it?

In the most basic form, texture mapping allows to put color patterns provided by 2D images onto 3D objects. From programmer's perspective, this is done by providing texture coordinates for each vertex. For each triangle of the 3D object the texture applied to it is copied from the triangle bounded by points given by the texture coordinates (see Figure 1). More precisely, texture coordinates are linearly interpolated when the triangle is scan-converted. Then, on the pixel processing stage, they are used to look up color from the texture image. Texture coordinates $(0, 0)$, $(0, 1)$, $(1, 0)$ and $(1, 1)$ correspond to the corners of the image. There are several ways to handle the case of one or both coordinates being outside of the range from 0 to 1. For example, one can use only the fractional part of each coordinate to do texture lookup or they can both be clamped to $[0, 1]$ (see 'Repeating and Clamping Textures' in Chapter 9 of the OpenGL programming guide).

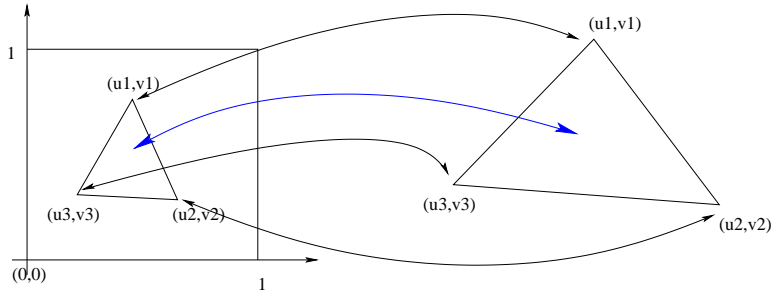


Figure 1: Applying a texture (left) to a 3D triangle (right). (u_i, v_i) are the texture coordinates of the vertices. The contents (colors) of the image specified as the texture are copied to the 3D triangle. Note that this requires stretching or shrinking the texture, particularly if the two triangles differ a lot in shape or size.

OpenGL allows to use up to 4 texture coordinates and they can be transformed using a 4×4 texture matrix (which works a lot like transforming vertex coordinates with the modelview matrix). Eventually (i.e. after this transformation), only two texture coordinates are used.

2 What can be done with textures

2.1 Complex color patterns on geometrically simple objects

So far, we could only define objects of varying color by specifying a different color (material) for each of the vertices. This means that for simple geometrically objects with complicated color patterns (think about e.g. a wooden table) we would need to use a lot of vertices. Textures allow to apply complex color patterns to 3D objects without increasing the number of polygons used.

2.2 Billboards

Textures allow for several tricks that are common in computer games or, more generally, real-time rendering.

Billboards, for example, are textures put on rectangles that are made rotate to always face the viewer. They are good for fast rendering of symmetric objects (that appear more or less the same from any direction). Trees are a good example. To model a tree using a billboard, one can find a nice tree and take a picture of it (alternatively, we might also want to model a tree geometrically and render it using an expensive photorealistic renderer). This yields a picture of a tree that we will use as texture.

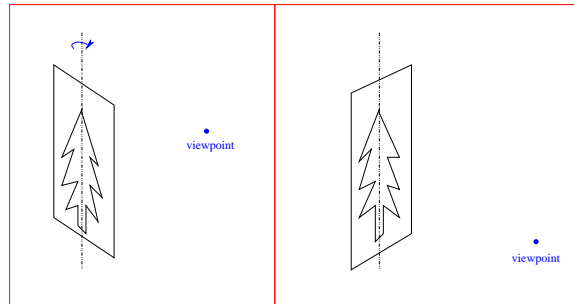


Figure 2: A billboard pretending a tree. It rotates around the axis so that it always faces the viewer.

A tree can be rendered as a textured rectangle, that will rotate depending on the location of the viewpoint (so that the tree always faces the viewer, see Figure 2). Of course, we need to take care about a few details. The rectangle with the tree texture can only be rotated around the tree's axis. Also, some pixels (the ones outside the tree's outline) should ideally be rendered as transparent. Of course, this works only for viewpoints on the ground. A forest modeled with billboards wouldn't look well from above.

2.3 Shadows on a plane

One can draw shadows on a plane (think of it as a rectangular table) using two rendering passes. In the first pass, a texture for the plane is generated by rendering the scene in front of the table with the viewpoint set to be the light source. All objects can be rendered in the same dark color on a light background, without any illumination. In the second pass, the scene is rendered in a usual way except for that the texture obtained in the first pass is applied to the table (see Figure 3).

Of course, there are several technicalities that need to be taken care of. First, the objects below the table wouldn't cast a shadow and therefore they should not be rendered in the first pass. Thus, one may want to use the table's plane as the back clipping plane during the first pass.

Then, in order to make things look well, we would typically blend the texture obtained in the first pass with the table's natural texture or color.

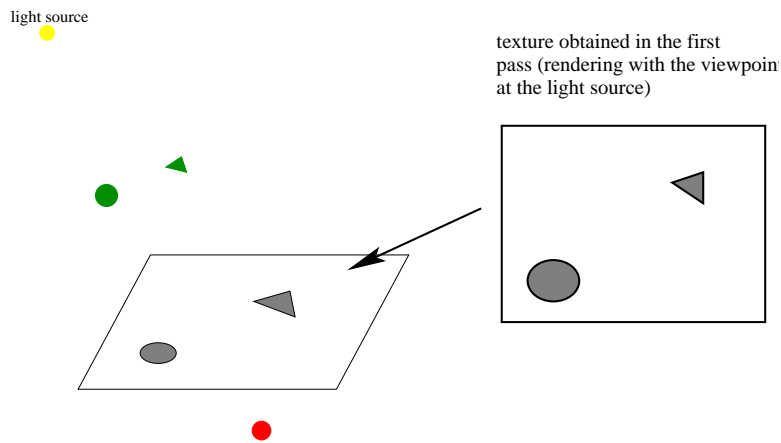


Figure 3: A two pass algorithm for shadows: in the first pass, one renders the scene with the table surface acting as the screen and with the viewpoint at the light source. Note that the red sphere below the table does not cast shadow. Using the table's plane as the back clipping plane ensures that it is not rendered in the first pass.

2.4 Reflections from mirrors

A two pass approach similar to shadow rendering can easily be applied to flat and rectangular mirrors. To obtain the texture for the mirror, one renders the scene from the viewpoint symmetrically on the other side of the mirror. Then, the scene is rendered again with the texture obtained in the first step applied to the mirror. This time, the front clipping plane has to be set to the mirror's plane (since we don't want to see reflections of objects on the other side of the mirror).

2.5 Environment mapping

2.5.1 Geometry of reflection

Let's look at an ideally reflective object. Consider 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 4). Therefore, if one is able to precompute and somehow store the colors incoming towards the reflective object along different rays, rendering of the image would be really simple.

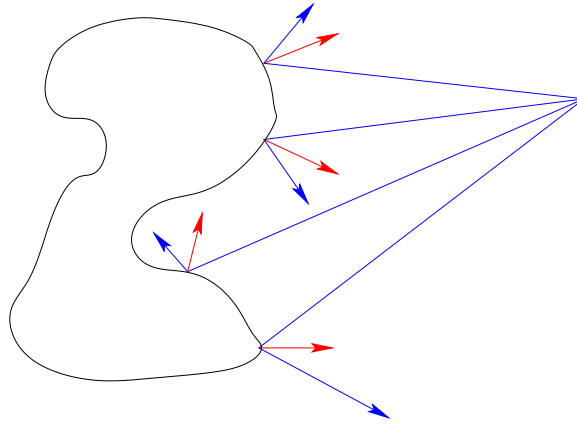


Figure 4: Eye rays reflected from an object. The reflected rays are shown as blue arrows. Red arrows are the normal vectors. Given the normal vectors, one can compute the reflected rays using the formulas we used to do Phong shading. Note that environment mapping will not produce correct results for rays which have to bounce from the object several times before escaping to the environment (like one of the rays shown above).

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 along parallel rays are the same (which is a lot like assuming that the objects that form the environment are large and distant). 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 approaches.

2.5.2 Spherical

One takes a photograph (or renders an image, e.g. using ray tracing) of a mirror sphere. This image stores colors incoming along rays of all possible directions (Figure 5). It is directly used as texture for spherical environment mapping. Such a texture might look like the one shown in Figure 6. To simplify things, let's say that the sphere is exactly inscribed into the (square) photograph.

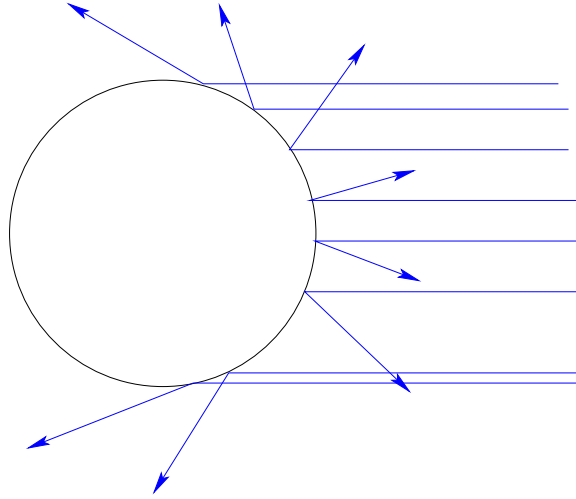


Figure 5: A mirror sphere. Any direction is a direction of a certain reflected horizontal ray. Spherical environment mapping is based on mapping colors from a photograph of a reflective ball onto a 3D object. This is possible because (if the photograph is taken from far away) colors incoming from all directions are present on that photograph (of course, there are two caveats here: first, a typical photograph is a perspective image rather than a parallel projection based image: the rays towards the viewpoint will not, therefore, be parallel; however, for images taken from far away, they will be *almost* parallel). And then, the photograph will not really store colors for every ray, but only at rays through pixels. But we'll not worry about this, since colors along rays other than through pixels can be obtained through interpolation.

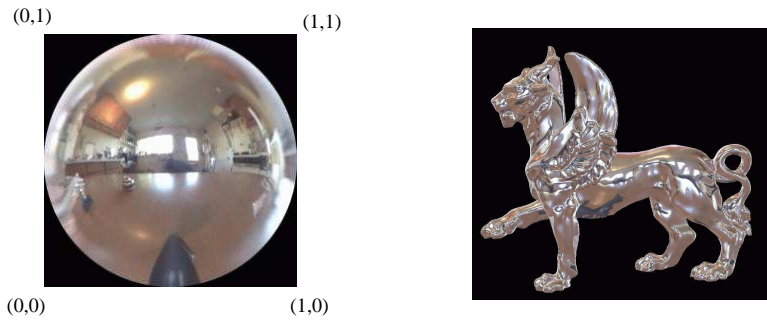


Figure 6: An example of a texture for use with spherical environment mapping and a 3D model with this spherical environment mapping.

To see what texture coordinates we need to use for 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) 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

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

2.5.3 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 7). 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 8.

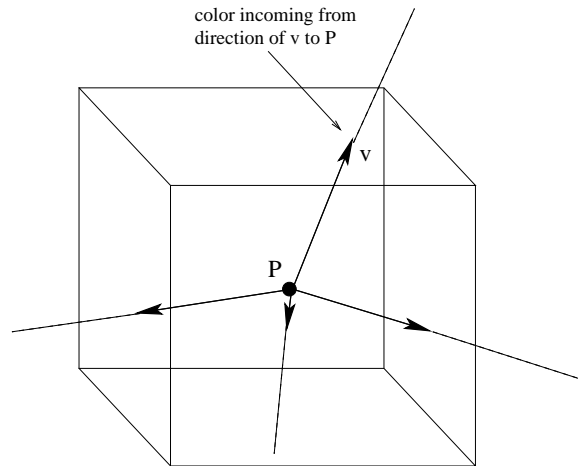


Figure 7: Cubical texture mapping.

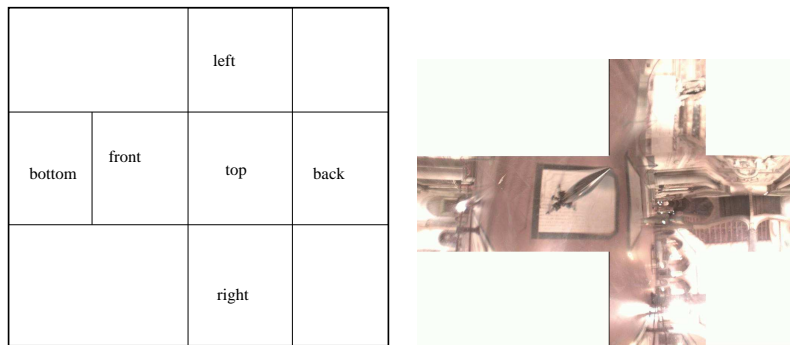


Figure 8: Flattening the cube environment map.