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 a set of texture coordinates for each vertex. In OpenGL, there can be up to four of them and they can be transformed using 4×4 matrices just like the vertex coordinates. Eventually (for 2D textures, since 3D texturing is also possible), the first two matter, and are used to look up color from the texture image. Typically, one thinks of them as being chosen from the interval from 0 to 1 ($(0,0)$, $(1,0)$, $(0,1)$ and $(1,1)$ corresponding to corners of the texture). There are several ways to deal with texture coordinates out of this range. For example, they can be clamped to $[0,1]$ or just their fractional part can be used.

Roughly speaking, 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).

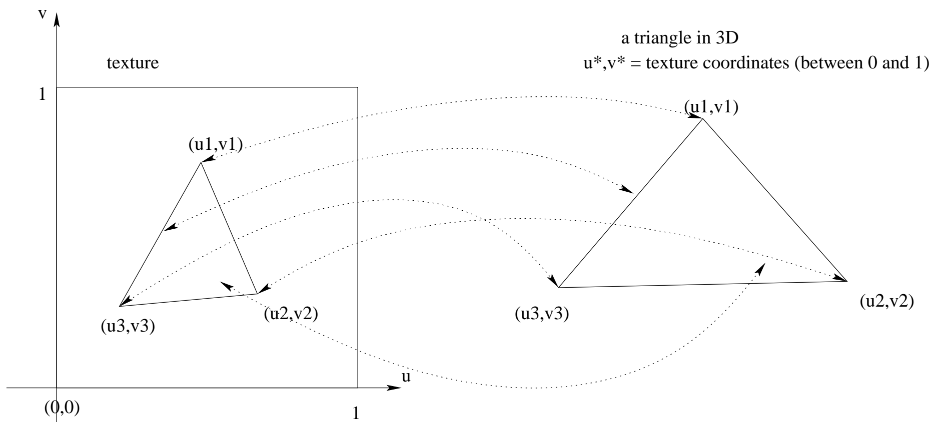


Figure 1: Applying texture to a triangle. To determine the color of a point on the triangle one linearly interpolates the texture coordinates from vertices and then uses the interpolated values as an index into the texture image.

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. This allows to considerably speed up the rendering.

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 bollboard, 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.

Now, a tree can be rendered as a textured rectangle, that can rotate depending on the location of the viewpoint (so that the tree always faces the viewer). 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 (see 2). Also, the pixels in the texture that are the background ideally should not be drawn.

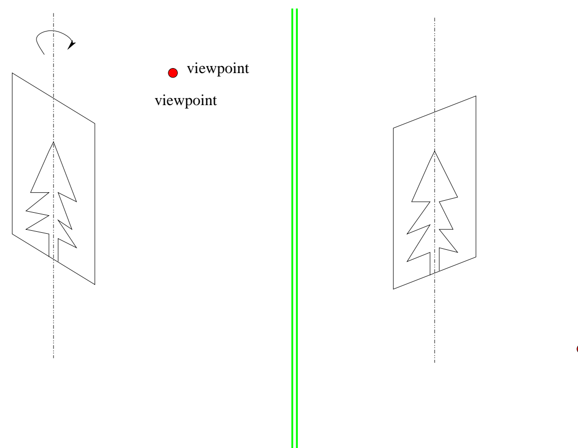


Figure 2: Billboard: an image rotates around the horizontal axis so that it always faces the viewpoint.

2.3 Shadows on a plane

One can draw shadows on a plane (e.g. table surface, flat ground etc) using two rendering passes (Figure 3). 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 at the light source. The projection matrix should be set in such a way that

the table surface exactly overlaps with the screen. The plane of the table can be used as the clipping plane to avoid drawing objects that are below the table (like the red sphere on the Figure – such objects obviously do not cast shadows on the table!). 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 surface (see Figure 3).

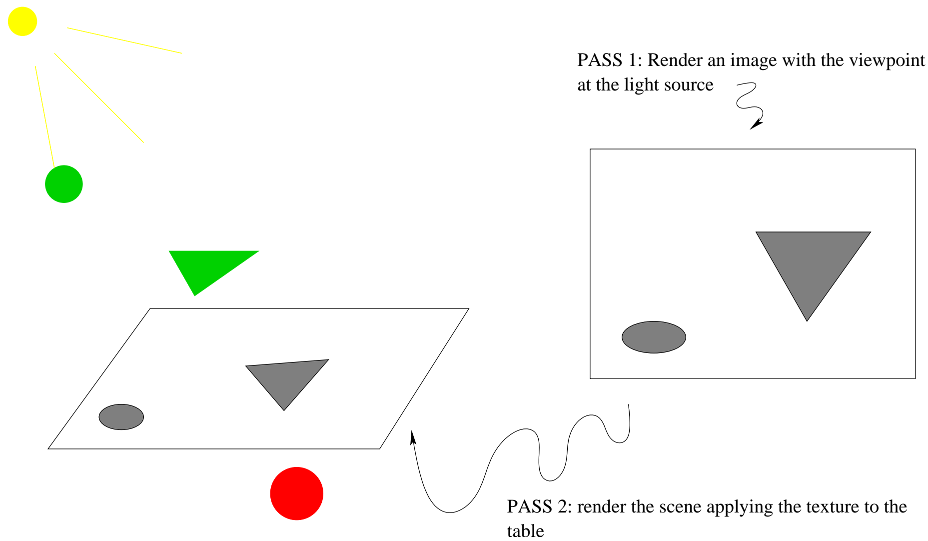


Figure 3: A two pass algorithm for shadows on planar surfaces.

2.4 Reflections from mirrors

A two pass approach can easily be applied to flat 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.

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. 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.

In environment mapping, one stores the colors arriving at a 3D object from different directions in one or more textures. Typically, one color per direction of

the reflected ray is stored and used (even though the incoming color also depends on where the reflection happens). Below we discuss two specific approaches.

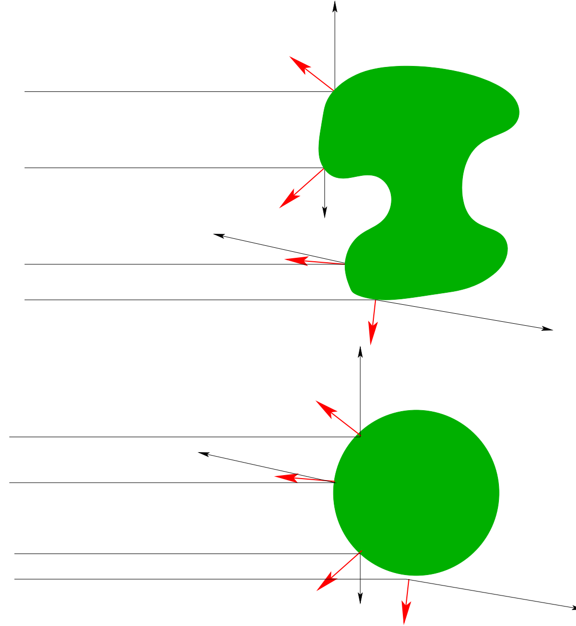


Figure 4: Geometry of reflected rays.

2.5.2 Spherical

If we are far away from the reflective object that we want to render, the eye rays that hit it are almost parallel. To simplify things, let us assume that they actually are exactly parallel. Then, the direction of the reflected ray depends only on the normal vector at the point hit by the eye ray. In figure 4 we show the reflected rays from two objects. Notice that the reflected rays are parallel if the normals at the reflected points are the same.

In spherical map, one assumes that the colors incoming along reflected rays are the same as the colors reflected from a mirror sphere placed more or less at the location of the reflective object to be rendered. One takes a photograph (or renders an image, e.g. using ray tracing) of a reflective sphere. This image stores colors incoming along rays of all possible directions (Figure 5). It can be directly used as texture. In fact, if the viewing direction is along the z-axis, the texture coordinates are the same as the x- and y- coordinates of the normal vector after scaling to $[0, 1]$.

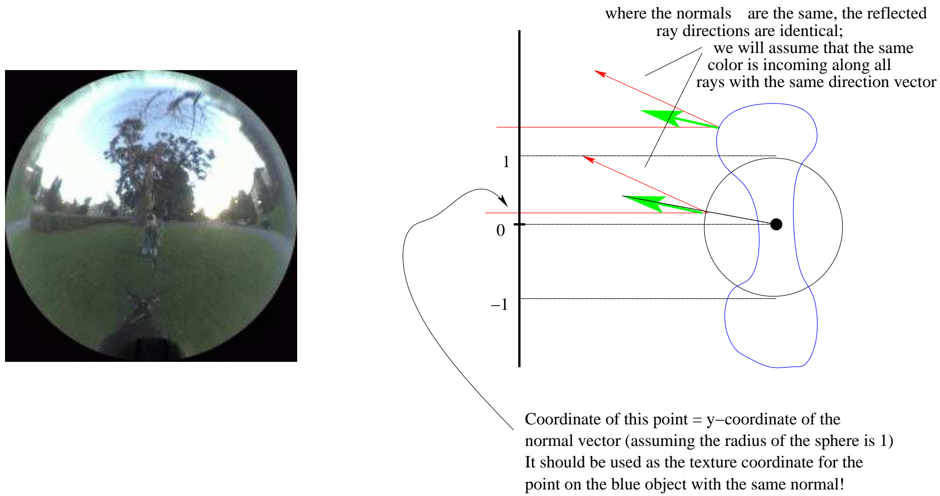
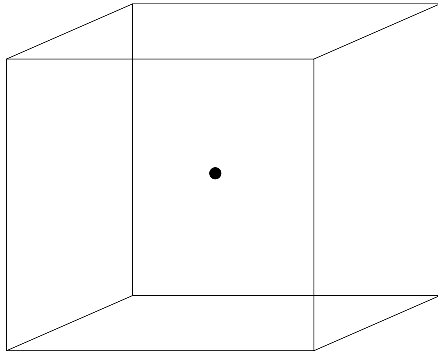


Figure 5: Image of a reflective sphere (left). Why can (the world) x and y coordinates be used as texture coordinates? (right)



Building a cubical environment map:
Put a viewpoint inside a cube
render six images, each with screen
overlaping with one of the cube's faces

Example (the six images unfolded):

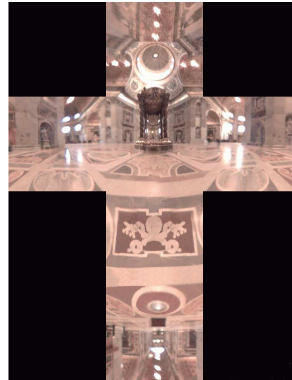


Figure 6: Cubical Environment map.

2.5.3 Cubical

Cubical environment map stores the incoming colors as six images, taken from a viewpoint somewhere at a point inside (in the center) of the object. Color for a ray will be approximated with the color for a ray with the same direction but

incoming towards that point (see Figure 6).