

## Rendering mirror surfaces

The next texture mapping method assumes we have a mirror surface, or at least a reflectance function that contains a mirror component. Examples might be a car window or hood, a puddle on the ground, or highly polished metal or ceramic. The intensity of light reflected off each surface point towards the camera depends on what is visible from the surface in the “mirror direction.” We write the formula for mirror reflection

$$\mathbf{r} = 2\mathbf{n}(\mathbf{n} \cdot \mathbf{v}) - \mathbf{v}$$

where  $\mathbf{v}$  is the direction to the viewer and  $\mathbf{r}$  is the direction of the incoming ray. This is essentially the same relationship that we saw in lecture 12 for two rays related by a mirror reflection, except there we had  $\mathbf{l}$  instead of  $\mathbf{r}$  for the direction of the incident ray. (We could have used  $\mathbf{l}$  again here, but I chose not to because I want to emphasize that we are not considering a light source direction. Rather we are considering *anything* that can be seen in that mirror reflection direction.)

Note that since the mirror surface is a finite distance away, the  $\mathbf{v}$  vector will typically vary along the surface regardless of whether the normal  $\mathbf{n}$  varies. Since both the surface normal and  $\mathbf{v}$  can vary along the surface, the reflection vector  $\mathbf{r}$  vector typically varies along the surface as well.

## Ray tracing

To correctly render the image seen “in the mirror,” we would need to cast a ray into the scene from the surface point and in direction  $\mathbf{r}$ . We need to ask what is visible along that ray and what is the intensity of light along that ray. This is called *ray tracing*. If another mirror surface were visible along that ray, then we would need to repeat this process e.g. recursively. In Assignment 3, you will get some experience with the (very) basics of ray tracing.

When we discussed ray casting back in lectures 7 and 8, we saw several methods for speeding it up. We can apply these methods to ray tracing too, but you should appreciate that ray tracing is still computationally expensive. In particular, it cannot be done using the OpenGL pipeline, since it requires access to all the objects in the scene. In practice, vertex and fragment shaders don’t have such access.

## Environment Mapping (Reflection Mapping)

Let’s next consider an alternative method for rendering mirror reflections, called *environment mapping*, which is much less expensive. An environment map is full “360 degree” RGB image of the a scene as seen from some particular position within the scene. More precisely, environment maps define RGB values over a sphere of directions, centered at some scene point. Defining functions on a sphere can be awkward since it is difficult to uniformly sample a sphere. We will discuss two methods for representing environment maps below.

Before we do, let’s look at how environment maps are used. Define an environment map to be a function  $E(\mathbf{r})$  of vector  $\mathbf{r}$  which is the direction of a ray going out into the scene from the position where the environment map is defined. Here is the basic algorithm.

```

for each pixel (xp,yp){
    cast a ray to find the intersection point
    if ray intersects mirror surface at (x,y,z) ){
        compute normal (nx,ny,nz)
        compute reflection direction (rx,ry,rz)
        I(xp,yp) := E(rx,ry,rz)
    }
}

```

This idea of this algorithm was proposed first by Blinn and Newell in 1976. (That paper also introduced the modified Blinn-Phong model.) See the slides for an example image which was shown in that paper. Note that, for this example image, the surface contains both a diffuse component (grey paint) and a mirror component.

### Cube map

How does one represent the environment map  $E(\mathbf{r})$ ? There are two common ways, and both are used in OpenGL. The first is a *cube map*. Here we represent the unit sphere of directions  $\mathbf{r}$  by sampling the six faces of a cube. Each of the six faces of the cube defines a projection plane with a field of view of 90 degrees. To make a cube map using computer graphics we could render six images, each with a 90 degree field of view, and the six view directions being  $\pm x, \pm y, \pm z$ . The faces of the cube would be  $x = \pm 1, y = \pm 1, z = \pm 1$ .

The ray  $\mathbf{r}$  intersects the cube at some position:

$$\frac{1}{\max\{|r_x|, |r_y|, |r_z|\}}(r_x, r_y, r_z)$$

and note that one of the coordinates has value either 1 or  $-1$ . Thus, the cube is  $2 \times 2 \times 2$ . The six images that define the environment could be written, say,

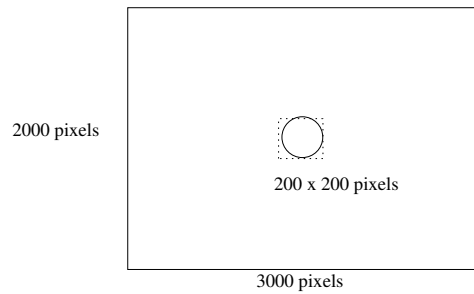
$$E_{z+}(r_x, r_y), E_{z-}(r_x, r_y), E_{y+}(r_x, r_z), E_{y-}(r_x, r_z), E_{x+}(r_y, r_z), E_{x-}(r_y, r_z).$$

To create the cube map of the environment, you render six images from the chosen scene point. (Here we assume that the scene does not have any mirror surfaces in it.) You can render these six images using OpenGL in the standard way using methods we have discussed up to now and which you are familiar with from the first two assignments (plus lighting and material).

### Sphere map

A second method for obtaining environment maps is to use a real image of an environment, rather than rendering the environment. This is analogous to using real images for texture maps  $T(s_p, t_p)$ , rather than using computer generated textures.

With a *sphere map* (also known as a *disk map*), the directions of the unit sphere are represented by a disk. The environment map is defined by the image in this disk. The disk map typically uses a digital photographic image of a real scene that is reflected off a real mirror sphere, located in the scene. In the figure below, the dotted square contains a cropped image of a mirrored sphere in a



real scene. See the example of the floating dog that I discussed in the lecture, and the example of the Terminator 2 movie that I mention later.

Recall how the environment map is used. For each reflection direction, we would like to look up a value in the environment map. But now the environment map is parameterized by points on an image of a sphere which is a texture. So we need a correspondence between reflection directions  $\mathbf{r}$  and “texels”  $(s_p, t_p)$  on the image of the sphere. In particular, we want a *mapping* from the vector  $\mathbf{r}$  to texture coordinates  $(s, t)$  or  $(s_p, t_p)$ . How to obtain this correspondence?

Assume the projection of the mirror sphere into the image is approximately orthographic. Then the direction of the viewer is approximately  $\mathbf{v} = (0, 0, 1)$  for all points on the small mirror sphere. A (unit) sphere has the nice property that the unit surface normal is identical to the position of a point on the sphere. Let’s normalize the image of the sphere so that the square containing it is  $[-1, 1] \times [-1, 1]$  and let  $(s, t)$  be the parameters of this  $2 \times 2$  ‘normalized’ square. Then, for any point  $(s, t, \sqrt{1 - (s^2 + t^2)})$  on the surface of the unit sphere mirror, the unit surface normal is

$$\mathbf{n} = (s, t, \sqrt{1 - (s^2 + t^2)}).$$

As we saw in lecture 12 (see also slides of this lecture near the beginning), a reflection vector  $\mathbf{r}(s, t)$  is computed using:

$$\mathbf{r} = 2(\mathbf{n} \cdot \mathbf{v})\mathbf{n} - \mathbf{v}.$$

So

$$(r_x, r_y, r_z + 1) = 2\sqrt{1 - (s^2 + t^2)} (s, t, \sqrt{1 - (s^2 + t^2)}) \quad (*)$$

Taking the magnitude of both sides (and remembering that  $\mathbf{n}$  is a unit vector) we get:

$$\sqrt{r_x^2 + r_y^2 + (r_z + 1)^2} = 2\sqrt{(1 - (s^2 + t^2))}$$

Substituting into the right hand side of Eq. (\*) and rearranging gives:

$$(s, t) = \frac{1}{\sqrt{r_x^2 + r_y^2 + (r_z + 1)^2}} (r_x, r_y)$$

This is the mapping we seek. For any  $\mathbf{r}$ , we can calculate the texture coordinates  $(s, t)$  that correspond to  $\mathbf{r}$ .

A few points of clarification are perhaps needed here. First, in OpenGL, one uses texture coordinates  $(s, t) \in [0, 1] \times [0, 1]$ . To obtain such coordinates, one needs to map a bit more, namely

scale by  $\frac{1}{2}$  and add  $\frac{1}{2}$ . This gives the formulas for  $(s, t)$  as a function of  $r_x, r_y, r_z$  that are found in the OpenGL specification and in some OpenGL/graphics textbooks.

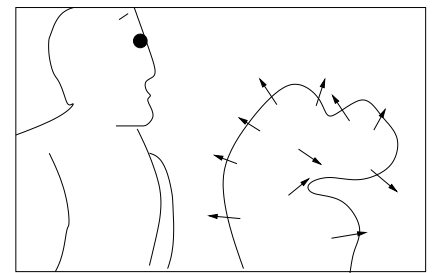
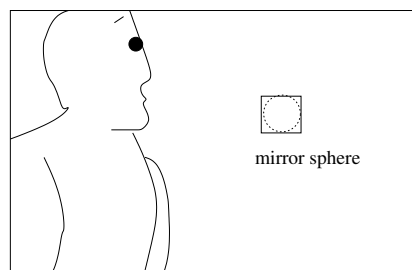
Second, you might get the impression from the above derivation of the mapping that, since we are assuming a constant  $\mathbf{v} = (0, 0, 1)$  direction, the reflection vector  $\mathbf{r}$  depends only on the surface normal  $\mathbf{n}$ . This impression is not correct. The mirror reflection vector  $\mathbf{r}$  depends both on the surface normal  $\mathbf{n}$  and on the viewing direction  $\mathbf{v}$  in the particular scene. The viewing direction  $\mathbf{v}$  on the mirror surface being rendered typically will not be constant on the surface.

### Example 1 (special effects in cinema: Terminator II)

Earlier I mentioned that the disk/sphere method uses a real mirror sphere and a real image of it. This gives an environment map texture image  $E(s_p, t_p)$ . In class, I discussed an example of how this was used in several scenes in the film Terminator II. (See see [www.debevec.org/ReflectionMapping/](http://www.debevec.org/ReflectionMapping/).) In the scene shown below, a computer graphics generated creature is inserted into the scene. The surface of the creature is a smooth mirror which seems to reflect the surrounding scene. The way this was done is illustrated in the figure below. The original scene was filmed with an actor and a real environment, and a mirror sphere was placed in the scene. In each frame of the film, the image in the mirror sphere served as the environment map for that frame.

Using these environment maps, a separate animation was created. The animation only consisted of the creature changing shape and moving over time. This was created entirely using computer graphics, of course, namely the environment maps were used to render a mirror reflection of the scene on the surface of the creature. Because the creature's surface is curved, the mirror reflection is distorted, like the reflection in a fun-house mirror. Finally, the animation of the moving creature was "inserted" into the film (covering up the mirror sphere).

Notice that the reflection of the actor's full face can be seen in the head of the creature, even though only part of the actor's face is visible in the original image (since his head is turned). This is exactly what should happen if there were a creature in the scene.



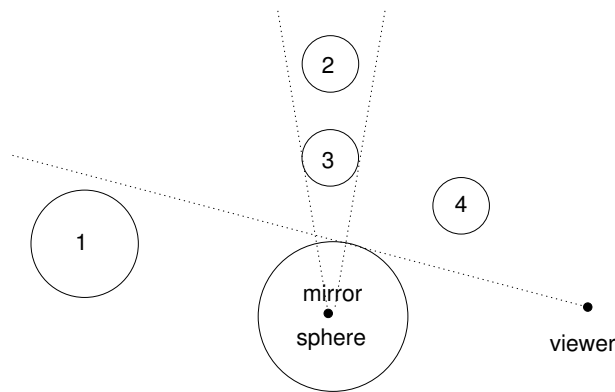
It is important to realize that environment mapping is only an approximation. It assumes that the intensity of a ray arriving at the mirror surface depends only on the direction of that ray and not on the position on the surface where the reflection occurs. This assumption is reasonable if that the environment that is being reflected is far away from the object and that there is nothing intervening between the object and its environment. However, if this condition is not met, then the rendering is not "correct". But generally it doesn't matter – people looking at it cannot tell.

## Example 2

Consider the scene shown below which consists of a mirror sphere, a set of matte spheres 1 to 4, a viewer position (small black disk), and a position from which the environment map is originally computed. (If the discussion below is not enough for you, see the images used in the slides which added further 'lines of sight'.)

If the scene were rendered using ray tracing, then which of the matte objects would be visible, either directly or in the mirror? Sphere 1 would not be visible, neither directly nor in the mirror. Sphere 2 would be partly visible<sup>1</sup> in the mirror, but not directly (occluded by 4). Sphere 3 would be partly visible directly (i.e. only partly occluded by 4) and would be visible in the mirror. Sphere 4 would be visible directly and in the mirror.

If environment mapping were used, then the visibilities would be different. Sphere 1 would be visible in the mirror (but still not visible directly). Sphere 2 would not be visible either in the mirror (since it would not be seen in the environment map) or directly (since it would be occluded by 4). Sphere 3 would be partly visible directly, and would be visible in the mirror. Sphere 4 would be as before.



## Environment mapping and the OpenGL pipeline

Ray tracing cannot be done using OpenGL graphics pipeline that we have been discussing. The reason is that, to know the RGB value of a reflected ray (off a mirror) from a point  $(x, y, z(x, y))$ , you need to find closest object in the scene along that reflected ray. However, finding the closest object requires access to a list of all the objects in the scene. The vertex and fragment processors do not have such access. They process the vertices and fragments individually using information about normals, surface attributes, texture coordinates. Similarly, the clipper processes the primitives and polygons individually.

<sup>1</sup>To see why, consider the ray that goes from the rightmost point on sphere 2 and passes near the rightmost point on sphere 3. This ray would reflect back and pass between sphere 3 and 4. Now consider a second ray which goes from the rightmost point of sphere 2 and touches close to the rightmost point of the mirror sphere. This ray would reflect to a ray that goes below the viewer. Somewhere between the above two rays from sphere 2 there must be a ray from sphere 2 that would reflect to the viewer. From this reasoning, you should be able to see that there are points on sphere 2 that are seen by the viewer.

The vertex and fragment shaders do have access to texture maps, of course, and this is what allows them to do environment mapping. So at what stages of the pipeline is environment mapping carried out?

There are two answers that should come to mind, which are similar to what we have been discussing with other methods in the last few lectures. The first answer is analogous to smooth shading. Here, the vertex processor would compute the RGB values for the vertices of a mirror polygonal surface – by computing the reflection vector and looking up the RGB value in an environment map. The rasterizer would then interpolate the reflection vector  $\mathbf{r}$  across the fragments of the polygon. The fragment processor would then use the interpolated reflection vector for each fragment to lookup the RGB value in the environment map.

This first answer is *not* what OpenGL does. Why not? The problem is essentially the same here as it was for smooth shading with specular highlights. In the case of a mirror polygon, you want to know what is visible at each point in the interior of the mirror but this depends on the reflection vector at each point. If you just interpolate the RGB values from the vertices, then the RGB values of all interior points will just be a linear combination of the vertex RGB values – and this would just give a smooth blurry mush, not a mirror reflection.

The second method (which is what OpenGL does use "under the hood") is that the vertex processor computes the reflection vectors of the vertices, and passes these to the rasterizer which now interpolates the reflection vectors across the fragments. The fragment processor is then responsible for computing the RGB values, based on the reflection vectors.

## Refraction

At the end of the lecture, I briefly discussed the problem of refraction whereby light passes through a transparent medium such as water or glass. In this case, the light direction changes at the interface between air and glass/water for reasons you learned about in your U0 physics course. The result is that the images of objects under water or behind glass are distorted. Ray tracing and environment mapping methods can be applied to render images in these situations. The basic idea is to trace rays of light (backwards) from the eye to the surface of these glass/water objects, through the objects, and then possibly out of the objects into the world. In the latter case, environment mapping can be used since one can just lookup the RGB value of the ray from the environment map  $E(\mathbf{r})$ .

To do this "properly", one considers the basic physics of refractions, e.g. Snell's law which you learned in U0 physics. In the lecture, I gave a few example images and videos which you should check out.