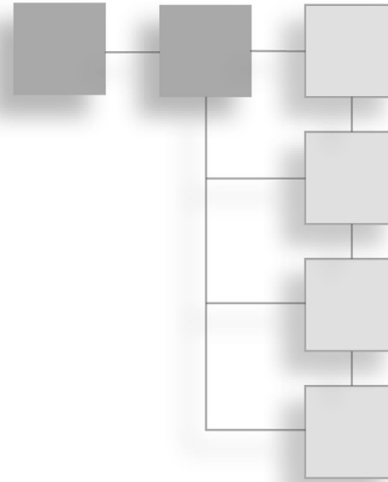


CHAPTER 11

MIRROR, MIRROR, ON THE WALL



In the last few chapters, you learned how lighting is an important component in making realistic graphics. You may remember that not all materials behave the same way or have specific properties. This chapter is the first in a series that addresses how different materials interact with their environment.

In this chapter, you will learn about the essentials you need to know when dealing with materials that are translucent and reflective. I will be covering the topics of reflection and refraction and how they interact. By the end of this chapter, you will be able to render materials that exhibit both translucency and reflectivity.

Although there may be different approaches to doing both reflections and refractions, we will concentrate on using cube environment maps. This method is a great way of representing the captured environment from a specific point within your scene and is easy to use for reflections and refractions.

From Reflections to Refractions

Many materials, as you know, have properties that allow them to either reflect or refract light. Probably the most obvious example is glass, which presents both phenomena at the same time. On the other hand, other materials that have glossy surfaces, such as car paint, also show reflections under the right lighting conditions.

Before we examine the details of how reflection and refraction can be reproduced, I must address two topics. First, I need to explain why cubemap environments are so well suited for such effects. Second, you must build the basic shader that you'll use throughout this chapter.

200 Chapter 11 ■ Mirror, Mirror, On the Wall

So why cubemaps? Cubemaps were covered in Chapter 6, “Blurring Things Up,” but they are worth revisiting.

A cubemap is a set of six textures grouped together forming a cube centered on a single point in space, with each face being a snapshot of the scene along a specific axis as shown in Figure 11.1. Although this may not seem like a natural way of representing an environment from a single point in space, it is a very efficient way for your hardware to do so.

Because of the way a cubemap is formed, looking up the environment map is easily done when given a direction vector to look at. The major axis of the vector is used to find out which face of the cube will be sampled; the remaining two components of the vector are used to access the cubemap face as if it were a regular texture.

note

Because of the nature of cubemaps and environment maps, there is an aspect you will need to keep in mind. Because the environment map represents a snapshot of your environment from a particular point in space, this implies that everything within it is considered to be at an infinite distance from that point; in simpler terms, because the environment is pre-cooked, it will not contain any perspective.

Accessing a cubemap is simply a matter of determining which face to access and then accessing it as a regular texture. This makes the cubemap a natural and easy feature to implement with existing hardware architectures. Another added benefit is how easy a cubemap is to build.

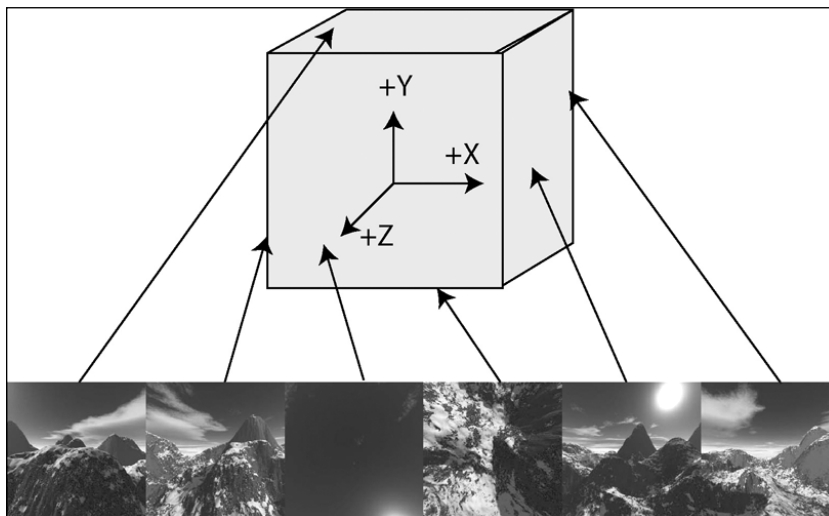


Figure 11.1 Illustration of how a cubemap is formed.

Building a cubemap from scratch is simply a matter of setting up six cameras with a 90 degree field of view facing all three major axes in both directions. At the end of this chapter, I will show you how you can dynamically construct a cubemap for use in a reflection and refraction shader. The process for building static environment maps, such as `Snow.dds`, generally involves authoring each face separately and using a specialized tool such as the Microsoft Texture Tool to composite them together.

Enough about cubemaps for now, though. Let's set up the basic shader that we will use throughout this chapter.

The basic shader you will use in this chapter is similar to the one developed for the heat haze and depth of field effects. The scene is composed of two passes. The first pass renders the environment cubemap you will use for this chapter to a sphere to create a background environment for the scene. (I recommend using `Snow.dds`, which is included on the CD-ROM source code directory for this chapter.)

The second rendering pass renders a standard teapot object to the scene. For now, you will simply render the teapot using a wood texture, but you will add reflection and refraction to this object later on. See Chapter 6 to learn how the workspace is set up. Right now, let's go back to the first pass and see how the shader code is formed.

When rendering an environment to a sphere in this way, you must take a few things into account. First of all, the sphere model you have is a unit-sized sphere centered on the origin. However, because you will be using this sphere to map an environment onto the camera, this sphere needs to be re-centered around the camera that is rendering it. This can easily be done by offsetting the sphere vertex positions by the camera's position, as defined by the built-in variable, before transforming it.

In addition to centering the sphere at the center of the camera, the environment map represents visual information that is located at infinity, relative to where the camera is located. This has the consequence that the environment will need to be rendered first, and the `ZWRITE` render state must be set to `D3D_FALSE` to avoid writing any depth information to the Z-buffer.

The other item that is needed by this shader is the view direction so the environment map can be sampled properly. Because the 3D sphere we are using is an origin-centered unit sphere, the view direction simply becomes the position of the sphere vertex. The vertex shader code that renders the environment cubemap to a sphere is as follows:

```
float4x4 view_proj_matrix;
float4 view_position;
struct VS_OUTPUT
{
    float4 Pos: POSITION;
    float3 dir: TEXCOORD0;
```

202 Chapter 11 ■ Mirror, Mirror, On the Wall

```
};

VS_OUTPUT vs_main(float4 Pos: POSITION)
{
    VS_OUTPUT Out;

    // Center environment around camera
    Out.Pos = mul(view_proj_matrix, float4(Pos.xyz + view_position, 1));
    Out.dir = Pos.xyz;

    return Out;
}
```

Within the pixel shader for this pass, all you have to do is read in the view direction passed in from the vertex shader and use it as a texture coordinate to look up the environment map. Doing so yields the following pixel shader code:

```
sampler Environment;
float4 ps_main(float3 dir: TEXCOORD0) : COLOR
{
    return texCUBE(Environment, dir);
}
```

The second pass needed for this basic shader simply renders a model with a texture applied to it. No point in describing shader code here because this is as simple as it gets.

With this shader compiled and running, you should get results similar to the one shown in Figure 11.2. This template shader has been included on the CD-ROM as `shader_1.rfx` in the directory for this chapter and is in fact very similar to the template shader developed in Chapter 7, “It’s Getting Hot in Here.”

You are now all set up and ready to start shading. So let’s start with the first topic at hand: reflections.

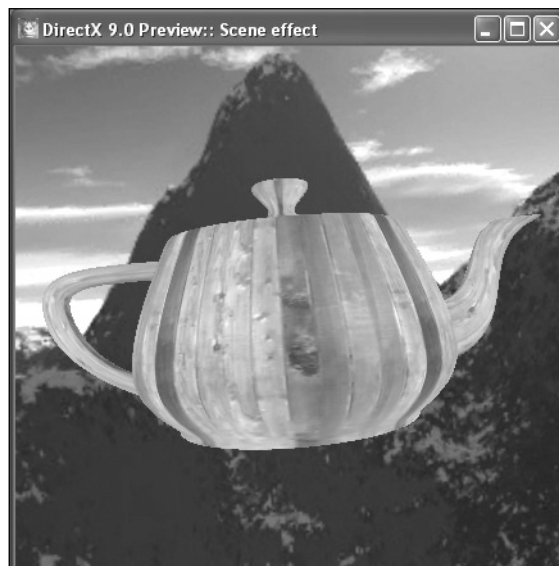


Figure 11.2 Rendering output for this chapter’s template shader.

Reflections

You may recall our discussion on specular lighting in Chapter 10, “Shiny Little Pixels.” Light from a source bounces off a polished surface onto the viewer. Although this may seem different from reflection, it is the same process. The specular lighting equation emulates the same phenomenon but from the point of view of a single light. The reality is that all visible objects in a scene can be seen as a source of light and treated in the same way, especially when you are dealing with highly glossy surfaces.

The most obvious example of a reflective material is a mirror, which reflects every ray of light it encounters almost perfectly. But some more subtle examples, such as water or glass, also exhibit reflections under the right set of circumstances.

Reflections occur when a ray of light emanating from a source of light (or another lit object) hits the surface of the reflective material and is bounced towards the viewer. As Figure 11.3 shows, the basic concept behind reflection is simple. The angle between the incident light angle and the surface normal is the same as the angle between the reflected ray and the surface normal. Figure 11.3 also shows how the reflected vector can be calculated from the incident light vector and surface normal.

note

Remember that HLSL has built-in functions for both reflection and refraction. Unless you have specific needs, you should use those built-in functions because it gives the compiler a better understanding of what you are trying to accomplish and results in more optimized code.

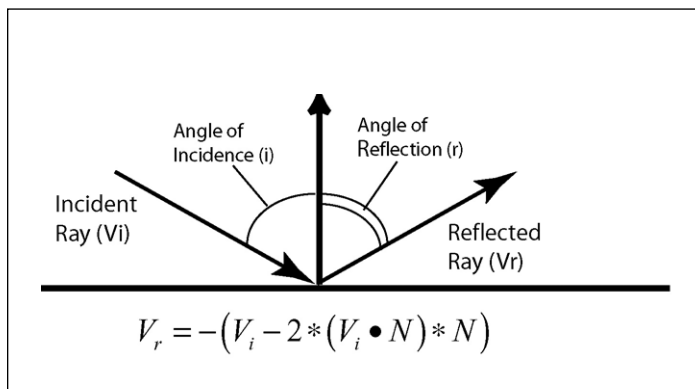


Figure 11.3 How rays of light reflect off a surface towards the viewer and the standard reflection equation.

204 Chapter 11 ■ Mirror, Mirror, On the Wall

With this information, rendering the scene with reflection on your teapot should be fairly straightforward. However, there is one aspect you must consider. In the preceding paragraphs, I discussed how light comes from a source, such as a scene object, bounces off your reflective surface, and heads toward the viewer. Actually, when rendering such a scene, you need to do this process upside-down. Because you cannot practically consider all sources of light in your scene, which would mean all pixels of your environment map, you need to start from your camera and trace the reflection in its reverse path to see what gets reflected.

To do this in a shader is simply a matter of determining the camera-to-object vector, and performing the reflection from this vector to get the source of the reflection from which to look up the environment map. The camera-to-object vector can be defined as the difference between the vertex position and the camera position (through the use of the built-in `view_position` variable).

For this shader, you only need to do the reflection calculations on the vertex shader to maximize shader efficiency. You will be asked to repeat the same process per-pixel in the exercises at the end of the chapter. After you calculate the reflection vector, it simply needs to be passed to the pixel shader, where it will be used to read from the environment map. Doing so yields the following vertex shader code:

```
float4x4 view_proj_matrix;
float4 view_position;
struct VS_OUTPUT
{
    float4 Pos:        POSITION;
    float2 TexCoord:   TEXCOORD0;
    float3 Reflect:    TEXCOORD1;
};

VS_OUTPUT vs_main(float4 inPos: POSITION, float3 inNormal: NORMAL,
                  float2 inTxr: TEXCOORD0)
{
    VS_OUTPUT Out;

    // Compute the projected position and send out the texture coordinates
    Out.Pos = mul(view_proj_matrix, inPos);
    Out.TexCoord = inTxr;

    // Compute the reflection vector
```

```

    Out.Reflect = -reflect(view_position-inPos,inNormal);

    return Out;
}

```

Notice the negative sign in front of the `reflect` function call. Because you are tracing the reflection from its destination to its source, you need to invert the resulting vector from the `reflect` function so that it points in the right direction within the environment cube-map. The pixel shader code simply needs to take in the reflection direction vector and sample the environment map by using the `texCUBE` function. Taking this environment value, proper reflection can be done by adding it to the object's texture to create glossiness on it. Following is an example of how this can be done, assuming a material with 40 percent reflectivity:

```

sampler Wood;
sampler EnvMap;
float4 ps_main(float2 inTxr: TEXCOORD0,float3 inReflect: TEXCOORD1) : COLOR
{
    // Output texture color with reflection map
    return 0.6*tex2D(Wood,inTxr)+0.4*texCUBE(EnvMap,inReflect);
}

```

With this shader compiled and running, your output should look similar to that shown in Figure 11.4. The complete version of this shader is included on the CD-ROM as `shader_2.rfx`.



Figure 11.4 Screenshot of the reflection shader in action.

Refraction

On the other end of the spectrum is the refraction effect. Translucent materials, such as glass, let rays of light through their surface. However, as I mentioned in Chapter 7, these rays of light are affected by the differences in density between the two media they cross, causing the rays of light to be deviated. This deviation is defined by Snell's Law, described through illustration and equation in Figure 11.5.

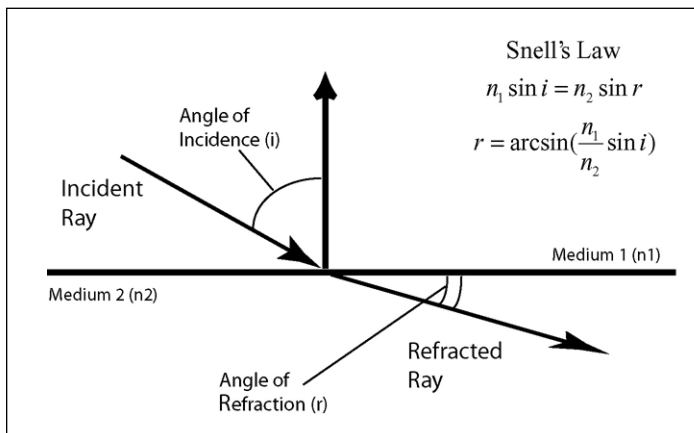


Figure 11.5 How refraction happens, and a description of the Snell's Law equation.

As you can see from the equation in Figure 11.5, the relationship between entering and exiting angles is dependent on the ratio of the index of refraction between the two media involved. The index of refraction is defined as the ratio between the two surface densities, that is, $IOR = n_1/n_2$. Table 11.1 summarizes the IOR (index of refraction) for many common materials.

Keep in mind that the IOR generally varies slightly in relationship to the color of light. The values given in Table 11.1 assume a midrange yellow-colored light. In the second exercise at the end of this chapter, I will ask you to expand on the refraction shader to consider the color of light in the refraction equation.

Table 11.1 Refraction Indexes for Various Materials

| Material | Refraction Index |
|------------|------------------|
| Air | 1.00 |
| Ice | 1.31 |
| Alcohol | 1.32 |
| Water | 1.33 |
| Plastic | 1.46 |
| Plexiglass | 1.51 |
| Glass | 1.52 |
| Emerald | 1.58 |
| Mercury | 1.62 |
| Ruby | 1.76 |
| Diamond | 2.42 |

With this, writing a refraction shader should be a simple matter of taking our previous shader and substituting the reflect function call with the refract one. However, at the time of this writing, the refract function does not always work as expected. Because of this, we will take the long way to solving the problem.

The first step in determining the refraction is to take the view vector and apply a dot product with the surface normal. As you may remember, the dot product of two vectors essentially gives you the cosine of the angle between the two vectors. However, to solve Snell's equation, you need the sine and not the cosine of the angle. This can be resolved by using the following identity: $\text{sine} = \sqrt{1 - \text{cosine}^2}$.

With this result, you can use your refraction indices to deduce the angle of the exiting ray. However, this angle only gives us a direction, not a vector! Because the refracted ray will be in the same plane as that formed by the surface normal and the incident ray of light, you can determine a 3D basis from the two vectors and then use the sine/cosine of the refracted ray angle to define the refracted ray vector.

The following code shows how this can be done:

```
float4x4 view_proj_matrix;
float4 view_position;
struct VS_OUTPUT
{
    float4 Pos:        POSITION;
    float2 TexCoord:   TEXCOORD0;
    float3 Refract:    TEXCOORD1;
};

VS_OUTPUT vs_main(float4 inPos: POSITION, float3 inNormal: NORMAL,
                  float2 inTxx: TEXCOORD0)
{
    VS_OUTPUT Out;

    // Compute the projected position and send out the texture coordinates
    Out.Pos = mul(view_proj_matrix, inPos);
    Out.TexCoord = inTxx;

    float3 viewVec = normalize(view_position - inPos);

    // Compute the reflection vector using Snell's Law
    // the refract HLSL function does not always work properly
    // n_i * sin(theta_i) = n_r * sin(theta_r)

    // sin(theta_i) : Determine the sine of the incident vector
```

208 Chapter 11 ■ Mirror, Mirror, On the Wall

```

float cosine = dot(viewVec, inNormal);
float sine = sqrt(1 - cosine * cosine);

// sin(theta_r) : Determine cosine of the refracted vector
// Note that the saturate(x) function is equivalent to
// using clamp(0,1,x). Also, 1.14 is the IOR for this
// shader.
float sine2 = saturate(1.14 * sine);
float cosine2 = sqrt(1 - sine2 * sine2);

// Determine the refraction vector be using the normal and tangent
// vectors as basis to determine the refraction direction
float3 x = -inNormal;
float3 y = normalize(cross(cross(viewVec, inNormal), inNormal));
Out.Refract = x * cosine2 + y * sine2;

return Out;
}

```

On the pixel shader side, all you need to do is sample the environment map and output the color as the following code does:

```

sampler Wood;
sampler EnvMap;
float4 ps_main(float2 inTxr: TEXCOORD0, float3 inRefract: TEXCOORD1) : COLOR
{
    // Output texture color with reflection map
    return texCUBE(EnvMap, inRefract);
}

```

With this shader compiled and running, your output should look similar to the one shown in Figure 11.6. The complete version of this shader is included on the CD-ROM as `shader_3.rfx` in the directory for this chapter. You may have noticed the gray border on the sides of the object. This is a natural phenomenon that I will address in the next section.



Figure 11.6 Rendering output for the final refraction shader.

Walking Hand in Hand

As you have seen from the rendering for the refraction shader, there are regions where no refraction occurs, and the result is a grayish color. The reason behind this result is not a coding error or similar glitch, but a natural phenomenon that occurs when dealing with refraction.

If you look at a container of water, such as an aquarium, dead-on, you will see through the water without any difficulty. However, if you look at the same container from a shallow angle, it will not be transparent anymore and will start behaving more like a mirror.

This phenomenon happens because refraction will stop happening past a certain angle, called the *critical angle*. At this angle, the refraction angle is equal to 90 degrees, and any incident rays past this angle exhibit a phenomenon called *total internal reflection* (or TIR), which in essence means that the surface will then behave as a mirror instead of being transparent. Figure 11.7 shows the transition of a refractive material towards TIR.

As you can see, because of total internal reflection, reflection and refraction actually go hand in hand, and this is exactly the next shader you will be writing. But before you can do so, you need to find out how the reflection and refraction combine.

We know that refraction will stop happening when the critical angle is hit or in other words, when the refraction angle is equal to 90 degrees. Because you already have the sine and cosine of the exiting angle, you can use this value to determine the correct ratio of reflected and refracted environment. Because the sine of this angle is already in the zero to one range, it makes a great candidate to be used as a blending factor.

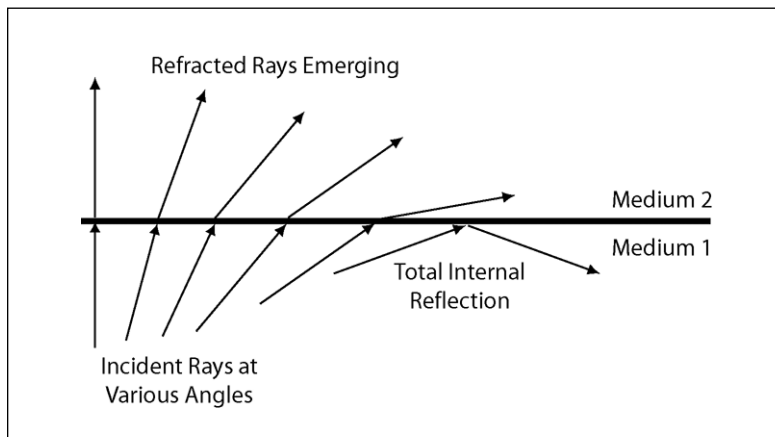


Figure 11.7 Illustration of how the total internal reflection phenomenon happens and the relationship between reflected and refracted light.

210 Chapter 11 ■ Mirror, Mirror, On the Wall

After you determine the blending factors for both reflection and refraction, which will also be passed to the pixel shader through one of the `TEXCOORD` variables, combining the two effects is simply a matter of putting the two code bases together and passing the blend factors, the reflection vectors, and the refraction vectors on to the pixel shader. Once the changes are done, you should end up with the following vertex shader code:

```
float4x4 view_proj_matrix;
float4 view_position;
struct VS_OUTPUT
{
    float4 Pos:        POSITION;
    float2 TexCoord: TEXCOORD0;
    float3 Refract:    TEXCOORD1;
    float3 Reflect:    TEXCOORD2;
    float2 Factors:    TEXCOORD3;
};

VS_OUTPUT vs_main(float4 inPos: POSITION, float3 inNormal: NORMAL,
                  float2 inTxr: TEXCOORD0)
{
    VS_OUTPUT Out;

    // Compute the projected position and send out the texture coordinates
    Out.Pos = mul(view_proj_matrix, inPos);
    Out.TexCoord = inTxr;

    float3 viewVec = normalize(view_position - inPos);

    // Compute reflection
    Out.Reflect = reflect(-viewVec, inNormal);

    // Compute the reflection vector using Snell's Law
    // the refract HLSL function does not always work properly
    //  $n_i * \sin(\theta_i) = n_r * \sin(\theta_r)$ 

    //  $\sin(\theta_i)$ 
    float cosine = dot(viewVec, inNormal);
    float sine = sqrt(1 - cosine * cosine);

    //  $\sin(\theta_r)$ 
    float sine2 = saturate(1.14 * sine);
    float cosine2 = sqrt(1 - sine2 * sine2);
```

```

// Determine the refraction vector be using the normal and tangent
// vectors as basis to determine the refraction direction
float3 x = -inNormal;
float3 y = normalize(cross(cross(viewVec, inNormal), inNormal));
Out.Refract = x * cosine2 + y * sine2;

// Determine proper reflection and refraction factors through
// a Fresnel approximation. (x = reflect, y = refract)
Out.Factors.x = sine2;
Out.Factors.y = (1 - sine2);

return Out;
}

```

The pixel shader for this combined reflection/refraction shader needs to take in three values. The first two are the lookup vectors for the environment cubemap lookup for the reflection and refraction, and the last value is the blending factors to use. Once you have looked up both environment values, combining them is a straightforward process.

Keep in mind that if you are applying a texture to your object, the refraction environment map value for refraction needs to be modulated with the object texture value because the texture has the effect of tinting the rays of light as they traverse the object. The following is an example pixel shader code detailing how the refraction and reflection can be combined:

```

sampler Wood;
sampler EnvMap;
float4 ps_main(float2 inTxr: TEXCOORD0, float3 inRefract: TEXCOORD1,
               float3 inReflect: TEXCOORD2, float2 inFct: TEXCOORD3) : COLOR
{
    // Output texture color with reflection map
    // Note the addition of 0.4 to the reflection/refraction
    // results to ensure a certain amount of ambient lighting
    return inFct.x * texCUBE(EnvMap, inReflect) +
           (inFct.y * texCUBE(EnvMap, inRefract) + 0.4)
           * tex2D(Wood, inTxr);
}

```

With this shader compiled and running, your output should look similar to the one shown in Figure 11.8. The complete version of this shader is included on the CD-ROM as `shader_4.rfx` in the directory for this chapter.



Figure 11.8 Rendering for the combined reflection and refraction shader.

Building Dynamic Environment Maps

All these reflection and refraction shenanigans are nice for static scenes with one object, but when your scene gets more dynamic or contains many objects, our current scheme falls short. Because the environment map contains only a static, prebuilt scene, any reflections or refractions done with it will not contain any other objects in your scene. Doesn't make much sense to do refraction on a teapot if you will not see the elephant right behind it, right?

The common solution to this problem is to use a dynamic cubemap instead of a static one. In this section I will briefly review how this can be achieved. Unfortunately, at the time of this writing, RenderMonkey does not support using cubemaps as render targets, so you will not be able to implement a shader with this technique.

Because a cubemap is essentially a collection of six textures, building a cubemap dynamically requires filling those textures one-by-one. When rendering with DirectX, each face of a cubemap can be accessed as an individual texture, which, in turn, can also be used as a render target. So the overall process is to render your scene six times, once for each face of the cube, setting up the camera so that it matches the point of view from that particular cubemap face.

note

At the time of this writing, cubemap textures are a special texture format that can only be read from .DDS files. This means that you cannot directly render to these textures, or you will need to use specialized tools such as the DirectX Texture Tool to author such textures. Also note that these textures can only be accessed through the use of the `texCUBE` HLSL functions because you need to tell the hardware you want to use a cubemap.

Because of the nature of a cubemap, rendering a face is a simple process. The camera needs to be positioned at the point in space where you want to build your environment map and must face the direction that matches the particular face you are rendering. You may want to refer to Figure 11.1 to see how cubemap faces correspond to a specific axis direction in world space.

The only other setting required for your camera to render cubemap faces is the field of view angle. The field of view defines the angle of the viewing frustum cone that the camera defines in space. Because all cubemap faces are of equal size, you need to set the FOV for your camera to 90 degrees. Doing so ensures that the edges for each cubemap face properly correspond and that the resulting environment appears seamless.

One last consideration when creating dynamic environment maps is performance. Because you must render each face individually, your scene needs to be rendered six times every time you update the cubemap. This may become a performance issue for some applications, and you may need to avoid updating the environment map every frame and try to spread the cost over time as much as possible.

It's Your Turn!

There you have it, your very own reflection and refraction shaders. The following exercises will ask you to expand on those shaders to try out your own shading skills. And as always, the solutions to these exercises are in Appendix D.

Exercise 1: DOING IT ALL PER-PIXEL

Starting with the combined reflection/refraction shader developed a few pages ago, modify it to do all of its operations on a per-pixel basis. This task should be simple and familiar by now, especially considering that per-pixel lighting was the topic of Chapter 10, so no hints will be given on how to perform this.

Exercise 2: COLOR-BASED REFRACTION

As mentioned earlier, the index of refraction, or IOR, for a particular material varies in function based on the color of the light that passes through the object. So far, you have assumed a constant IOR and have ignored this fact.

For this exercise, you are asked to implement a refraction shader which considers the color dispersion due to the variation of the IOR based on the color of light. To do this, start off with the per-pixel shader developed in the previous exercise and adapt it so that a different refraction vector will be calculated for each color component (red, green, and blue) and sample the environment once for each color component. Do not focus on trying to correctly determine an IOR for each color; simply use three nearby values.

What's Next?

As you learned in this chapter, the interaction of light with translucent materials has many aspects for you to consider, the main two being reflection and refraction. Although the concepts behind those two effects are simple, they require you to render components of your scene that are not necessarily easy to access.

This is where environment maps save the day! By estimating a full environment from a specific point in your scene, you can take advantage of those powerful effects and significantly enrich your renderings. However, when dealing with more dynamic environments, you will have to take advantage of the nature of cubemaps and dynamically build an environment map by rendering your scene to each face of the cubemap.

Now that you are on the topic of the interaction of light with surface materials, the next chapter will discuss the topic of *Bi-Directional Refractance Functions*, or BDRFs. These functions help define the properties for materials such as velvet, where the relationship between the lighting and viewing angle cannot be described in terms of simple diffuse and specular lighting.