Cube Mapping and GLSL

Notes for a MSc Course in Computer Graphics
University of Minho
António Ramires

1 The Equations for reflection and refraction

In this chapter we're going to explore some lighting effects such as reflection, refraction and chromatic dispersion.

Reflection is the change of direction at an interface between two different surfaces such that the light returns to the medium where it started from.

To compute the reflected direction we must take into account the incident light direction and the normal to the surface. We can also think about reflection on the opposite direction, considering the incident direction as the view direction such as in Figure 1.

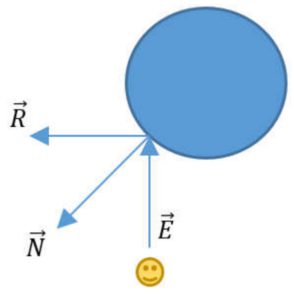

Figure 1 – Reflection.

The reflection direction, $\vec{R}$, can be computed as in Eq. 1.

$$\vec{R} = 2\left((\overrightarrow{-E}) \cdot \vec{N}\right)\vec{N} + \vec{E} \qquad \text{Eq. 1}$$

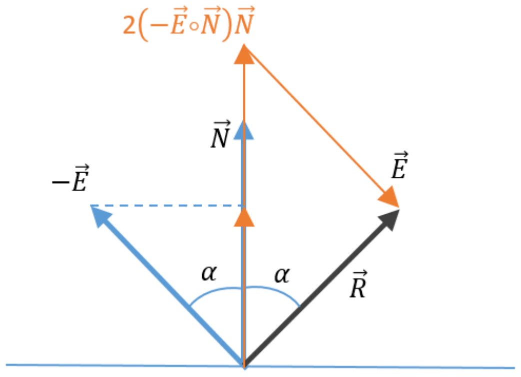

Figure 2 – Computing the reflection vector

In non-opaque materials we also have refraction. When light enters a new medium at an oblique angle it's direction is changed according to the respective refractive indices. The refractive index of an optical medium is a quantity n that describes how light propagates through that medium, and is defined as in Eq. 2

$$n = \frac{c}{v} \qquad \text{Eq. 2}$$

Where c is the speed of light in vacuum, and v is the speed of light in the medium.

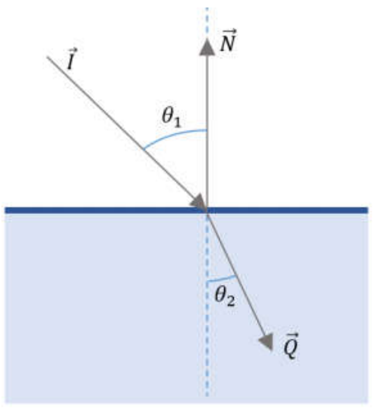

Figure 3- Refraction

In Figure 3, vector $\vec{I}$ represents the light's incident direction and $\vec{Q}$ the new direction after entering the bottom medium. Snell's law establishes the relation between the two angles θ_1 and θ_2, and the refractive indices of both mediums, n_1 and n_2 (Eq. 3).

$$\frac{n_2}{n_1} = \frac{\sin\theta_1}{\sin\theta_2} \qquad \text{Eq. 3}$$

To compute $\vec{Q}$ we start by computing a vector in the same plane as $\vec{I}$ and $\vec{N}$. Using this vector and $\vec{N}$ we can easily compute $\vec{Q}$ using some trigonometry.

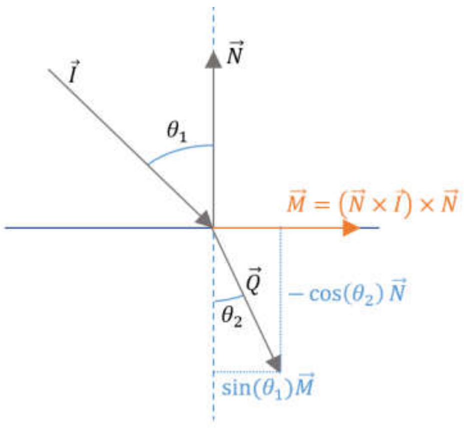
$\vec{I}$
$\vec{N}$
θ_1
$\vec{M} = (\vec{N} \times \vec{I}) \times \vec{N}$
$\vec{Q}$
$-\cos(\theta_2)\,\vec{N}$
θ_2
$\sin(\theta_1)\vec{M}$

Figure 4 – Computing the refraction direction

As shown in Figure 4, computing $\vec{Q}$ can be done with Eq. 4

$$\vec{Q} = \sin(\theta_1)\,\vec{M} - \cos(\theta_2)\,\vec{N} \qquad \textit{Eq. 4}$$

The above figures (Figure 3 and Figure 4) consider the initial medium with a smaller refractive index compared to the second medium. The light bends towards the normal. When the opposite occurs, for instance light traveling from water to air, the light bends away from the normal.

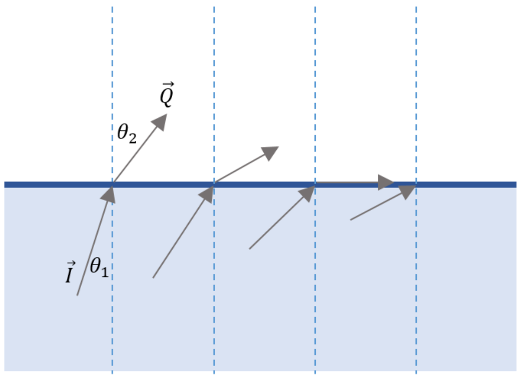

Figure 5 – Total Internal Reflection

Moving to a less dense medium can cause what is known as Total Internal Reflection, i.e. no light leaves the first medium, and there is no refraction ray, as in the rightmost ray in Figure 5. As shown in Figure 5, as the incidence angle moves away from the normal, the outgoing direction gets closer to the tangent of the surface. When the outgoing direction matches the surface's tangent, the incident direction is at the Critical Angle. This angle can easily be computed using Snell's law, since the critical angle implies that θ_2 is 90 degrees and both refractive indices are known.

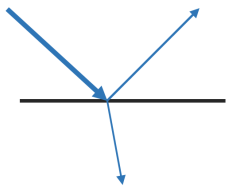

Figure 6 – Reflection plus refraction

When light hits a transparent medium, part is reflected, and if the critical angle is not achieved, part is refracted. Determining the percentage of light that goes each way can be computed with Fresnel equations.

These equations are relatively complex and account for different wave lengths and light polarization. Several good approximations, lighter to compute, assume that light is not polarized and ignore the wavelength spectrum.

One such approximation to compute the percentage of reflected light was proposed by Schlick, see Eq. 5.

$$F = f + (1-f)\left(1 - \vec{V}\cdot\vec{N}\right)^5 \qquad \textit{Eq. 5}$$

Where $\vec{V}$ is the vector from the point to the eye, $\vec{N}$ is the surface normal, and f, the reflectance of the material when the angle of incidence is zero, can be computed using

$$f = \frac{\left(1 - \frac{n_1}{n_2}\right)^2}{\left(1 + \frac{n_1}{n_2}\right)^2} \qquad \textit{Eq. 6}$$

Where $n_1$and n_2 are the refractive indices for the mediums on both sides of the interface.

Another approximation was proposed in the book Cg Tutorial, see

$$F = \max(0, \min(1, bias + scale \times \left(1 - \vec{V}\cdot\vec{N}\right)^{power}) \qquad \textit{Eq. 7}$$

The second term will be zero when $\vec{V}$ and $\vec{N}$ coincide, therefore $bias$ represents the reflectance of the material when the incident vector is collinear with the normal vector. The $scale$ and $power$ variables allow for a broad variety of effects.

2 Cube Mapping

A cube map can be seen as a set of six textures that define the walls of a cube. A 3D texture coordinate can be used to access the relevant texel, as a vector pointing from the center of the cube and intersecting one of its walls.

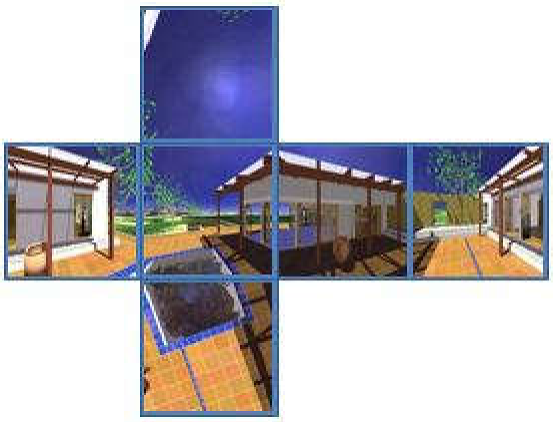

Figure 7 - The six walls of a cube

Formally speaking, the textures are not defined as appears above, but rather as in the following figure:

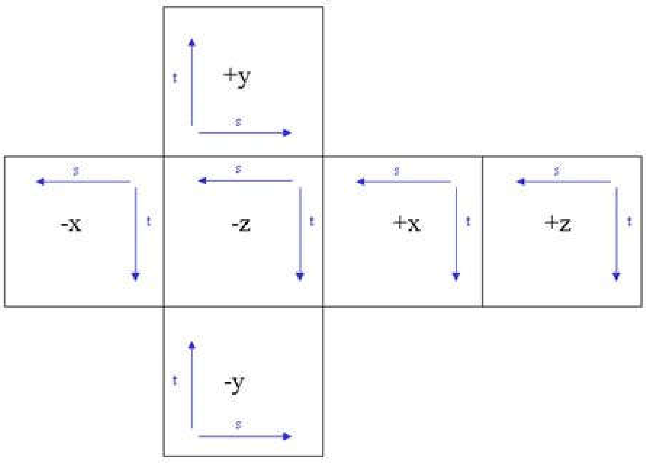

Figure 8 - Cube map texture orientation

2.1 OpenGL setup

First we have to create a cube map texture. The texture requires six square images, one for each side of the cube. To distinguish these textures, OpenGL defines a specific target for each one.

The face targets can be defined in an array as follows:

```
GLenum faceTarget[6] = {
        GL_TEXTURE_CUBE_MAP_POSITIVE_X,
        GL_TEXTURE_CUBE_MAP_NEGATIVE_X,
        GL_TEXTURE_CUBE_MAP_POSITIVE_Y,
        GL_TEXTURE_CUBE_MAP_NEGATIVE_Y,
        GL_TEXTURE_CUBE_MAP_POSITIVE_Z,
        GL_TEXTURE_CUBE_MAP_NEGATIVE_Z
    };
```

The texture creation is similar to the 2D case. The following routine shows how to load and create a cubemap.

```
unsigned int loadCubeMapTexture(
std::string posX, std::string negX,
                              std::string posY, std::string negY,
                              std::string posZ, std::string negZ) {
    ILboolean success;
    unsigned int imageID;
    GLuint textureID = 0;
    std::string files[6];

    files[0] = posX; files[1] = negX;
    files[2] = posY; files[3] = negY;
    files[4] = posZ; files[5] = negZ;

/* Texture name generation */
    glGenTextures(1, &textureID);
    /* Bind the texture as a cube map */
glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);

    /* set the texture parameters */
    glTexParameteri(GL_TEXTURE_CUBE_MAP,
GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP,
GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP,
GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_CUBE_MAP,
GL_TEXTURE_WRAP_T, GL_REPEAT);

    // generate an il image
    ilGenImages(1, &imageID);
    // bind it
    ilBindImage(imageID); /* Binding of DevIL image name */

    // for each cube map face do ...
    for (int i = 0; i < 6; ++i) {
        ilEnable(IL_ORIGIN_SET);
        ilOriginFunc(IL_ORIGIN_LOWER_LEFT);
        success = ilLoadImage((ILstring)files[i].c_str());

        if (!success) {
            // something has gone wrong
            ...
            ilDeleteImages(1, &imageID);
            return 0;
        }
        /* Convert image to RGBA */
        ilConvertImage(IL_RGBA, IL_UNSIGNED_BYTE);

        /* Create and load images to OpenGL */
        glTexImage2D(faceTarget[i], 0, GL_RGBA,
                        ilGetInteger(IL_IMAGE_WIDTH),
                        ilGetInteger(IL_IMAGE_HEIGHT),
                        0, GL_RGBA, GL_UNSIGNED_BYTE,
                        ilGetData());
    }
```

```
        /* unbind the texture */
        glBindTexture(GL_TEXTURE_CUBE_MAP,0);

        /* Release memory used by image. */
        ilDeleteImages(1, &imageID);
        // return the texture ID
        return textureID;
}
```

3 Reflections simulated with cube maps

The texture coordinate to access a cube map is a 3D vector, or a direction, which will provide a point in the cube, i.e a 2D texture coordinate in the respective texture. When using a cubemap to create environmental reflections the vector we are looking for is a reflection vector. This requires both the normal vector and the incident vector, the vector from the camera to the point of the surface.

The incident vector can be computed in either world or camera space. In here we are going to perform these computations in world space. To compute this vector the shader requires to know the camera position, the position of the surface point, and the normal, all in world space. This is pretty much the same information as in the lighting shaders, and the vertex shader is very similar, except that in here we're going to work in world space just to show that it is also an option.

The vertex shader will transform vectors and points to provide the data for the fragment shader.

```
#version 330

uniform mat4 PVM;
uniform mat4 M;
uniform vec3 camWorldPos; // world space


in vec4 position; // local space
in vec4 normal;   // local space

out vec3 normalV;
out vec3 eyeV;

void main () {
    // compute normal in world space
    normalV = normalize(vec3(M * normal));
    // compute position in world space
    vec3 pos = vec3(M * position);
    // compute look direction in world space
    eyeV = normalize(pos - camWorldPos);

    gl_Position = PVM * position;
}
```

Regarding the fragment shader we need to compute the reflection vector and perform a texel fetch using a cube map sampler. The shader itself is straightforward.

```
#version 330
```

```
uniform samplerCube texUnit;

in vec3 normalV;
in vec3 eyeV;

out vec4 outColor;

void main() {

	// normalize vectors
	vec3 n = normalize(normalV);
	vec3 e = normalize(eyeV);

	// compute reflection vector
	vec3 t = reflect(e, n);

	// access cube map texture
	vec3 ref = texture(texUnit, t).rgb;

	outColor = vec4(ref, 1.0);
}
```

The result from this shader could be something like the following figure.

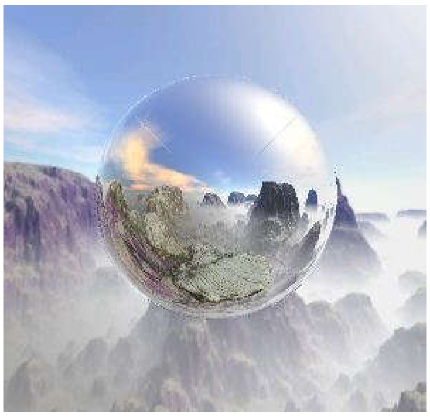

Figure 9 - Cube map providing environmental reflections to a sphere

3.1 Transparency

A transparency effect can be added as a cheap way to simulate glass. The goal is to keep the reflections but blend them with the background using simple transparency.

In OpenGL we have enable blending and set the blending function parameters which can be achieved by adding the following two lines on the application setup:

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

In the shader itself there is only a small change related to the output. Instead of writing

```
	outColor = vec4(ref, 1.0);
```

we write

```
        outColor = vec4(ref, opacity);
```
where `opacity` controls the mixture between the reflection color and the background. Notice that for the mix to occur properly the background must have been drawn before the transparent objects.

We can create a slightly more refined approach determining the opacity as a function of the dot product between the normal and the eye vector. If the reverse eye vector coincides with the normal then the surface will be highly transparent, and as the vectors diverge the transparency effect vanishes.

In the vertex shader the effect is computed as:

```
        float opacity = dot(-e,n);
        outColor = vec4(ref, opacity);
```

This latter approach will look slightly more realistic, as the following figure shows:

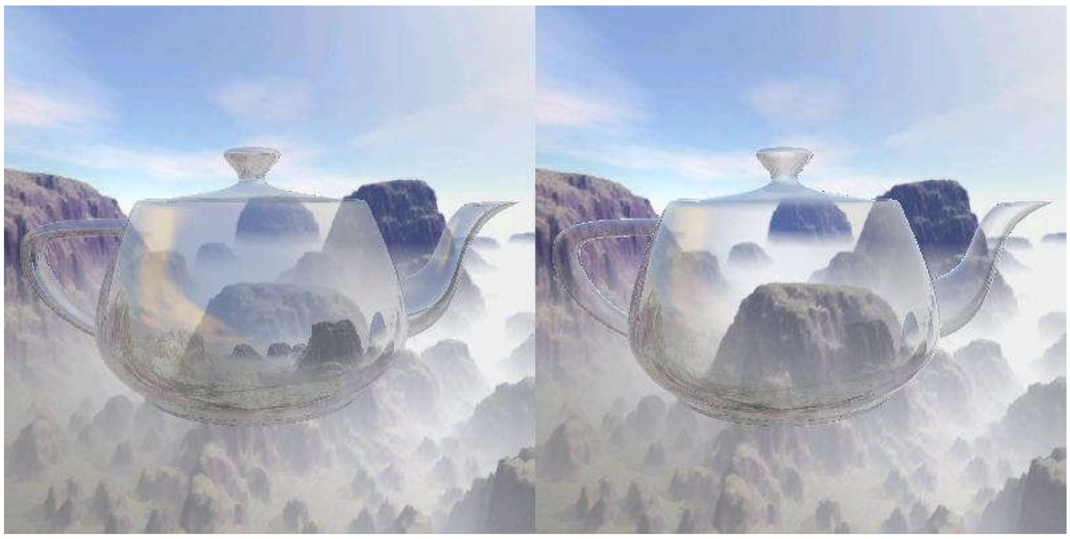

Figure 10 - Transparency factor. Left: constant level; Right: modulated by dot product

4 Refractions simulated with cube maps

When light is transmitted through an object, it changes direction when it changes medium. For instance, when a light ray moving in the air enters a glass volume, the ray changes direction. We can “invert” the problem, and imagine what happens to the rays that “comes out of our eyes”.

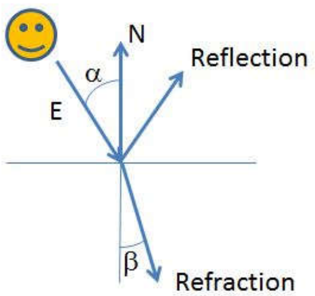

Figure 11 - Refraction and reflection.

Fortunately, GLSL has a function to compute the refraction vector based only on E and N vectors, as well as the coefficient $eta = \frac{n_1}{n_2}$. This vector can then be used to sample the texture.

```
// compute refraction vector
vec3 trefr = refract(e, n, eta);
// access cube map texture
vec3 refr = texture(texUnit, trefr).rgb;
```

So we have two colors, one for the reflection and another for the refraction. Each effect can be seen in Figure 12.

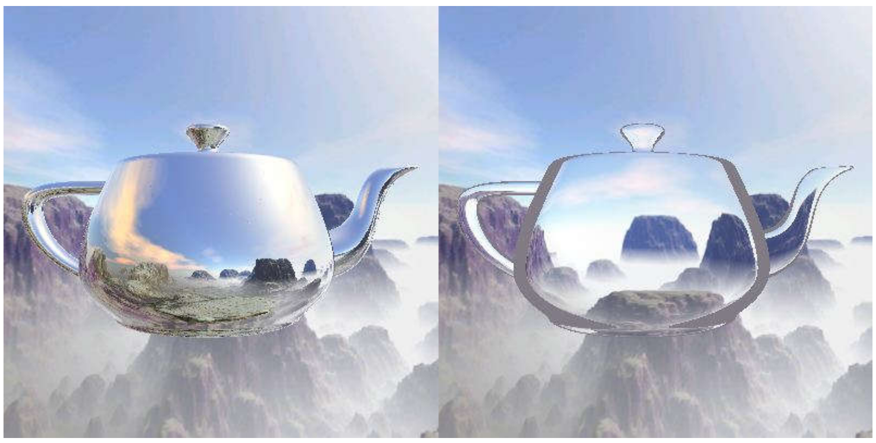

Figure 12 - Left: reflection; Right: refraction

Function `refract` gives us either a unit vector with the direction of the refracted ray, or a null vector, when the incident angle is past the total reflection angle. Looking at the GLSL spec we can see that a constant k is being computed as

$$k = 1 - \mathrm{eta}^2 * (1 - (\mathrm{N} \cdot \mathrm{E})^2) \qquad \text{Eq. 8}$$

Displaying k as an intensity provides the result in Figure 13. The value of k is also suitable for combining the refracted and reflected colors.

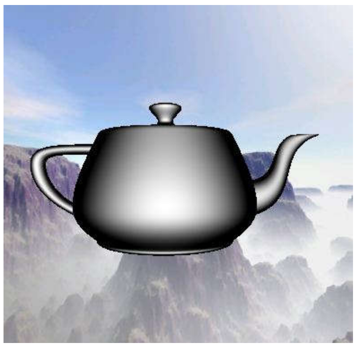

Figure 13 - Value of k as an intensity

The following shader uses Schlick approximation to simulate the Fresnel effect.

```
#version 330

uniform samplerCube texUnit;
uniform float f = 0.05;
uniform float power = 2.0;
uniform float bias = 0.3;
uniform float scale = 2.0;
uniform float eta = 0.66;

in vec3 normalV;
in vec3 eyeV;

out vec4 outColor;

void main() {

      // normalize vectors
      vec3 n = normalize(normalV);
      vec3 e = normalize(eyeV);

      // compute reflection vector
      vec3 trefl = reflect(e, n);

      // compute refraction vector
      vec3 trefr = refract(e, n, eta);

      // access cube map texture
      vec3 refl = texture(texUnit, trefl).rgb;
      vec3 refr = texture(texUnit, trefr).rgb;

      // Schlick approximation
      float f =  ((1.0 - eta) * (1.0 - eta)) /
                        ((1.0 + eta) * (1.0 + eta));
      float schlick = f + (1 - f) * pow(1 + dot(-e,n), power);


      outColor = vec4(mix(refr, refl, f), 1);
}
```

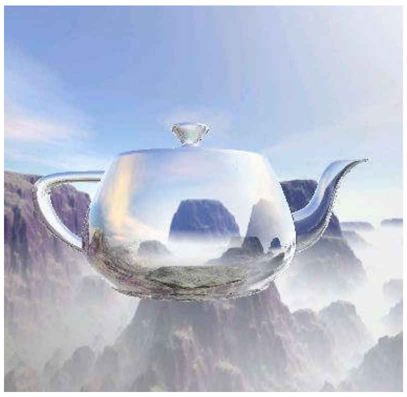

Figure 14 - Reflection + refraction

5 Chromatic Dispersion

Different wavelengths, colours, have different refractive indexes. Hence we can assign a different coefficient ratio to each of the RGB colors. This would provide a different refraction vector for each color. Using each vector as a texture coordinate we sample the cubemap to retrieve the respective component. The GLSL code could be as follows:

```
#version 330

uniform samplerCube texUnit;
uniform float f = 0.05;
uniform float power = 2.0;
uniform float bias = 0.3;
uniform float scale = 2.0;
uniform float eta = 0.66;

in vec3 normalV;
in vec3 eyeV;

out vec4 outColor;

void main() {

      // normalize vectors
      vec3 n = normalize(normalV);
      vec3 e = normalize(eyeV);

      // compute reflection vector
      vec3 trefl = reflect(e, n);

      // compute refraction vector
      vec3 trefrRED = refract(e, n, eta);
      vec3 trefrGREEN = refract(e, n, eta + 0.01);
      vec3 trefrBLUE = refract(e, n, eta + 0.02);

      // access cube map texture
      vec3 refl = texture(texUnit, trefl).rgb;
      vec3 refr;
      refr.r = texture(texUnit, trefrRED).r;
      refr.g = texture(texUnit, trefrGREEN).g;
      refr.b = texture(texUnit, trefrBLUE).b;

      // simple version
      float fresnelApprox = 1 - pow(dot(-e,n), power);

      // schlick approximation
      float f =  ((1.0 - eta) * (1.0 - eta)) /
                     ((1.0 + eta) * (1.0 + eta));
      float schlick = f + (1 - f) * pow(1 - dot(-e,n), power);

      // cgTut empirical approximation
      float R = max(0, min(1, bias + scale * pow(1.0 - dot(n,-e),
power)));

      outColor = vec4(mix(refr, refl, R), 1);
      // debug - show the reflection coefficient
      //outColor = vec4(vec3(fresnelApprox), 1);
}
```

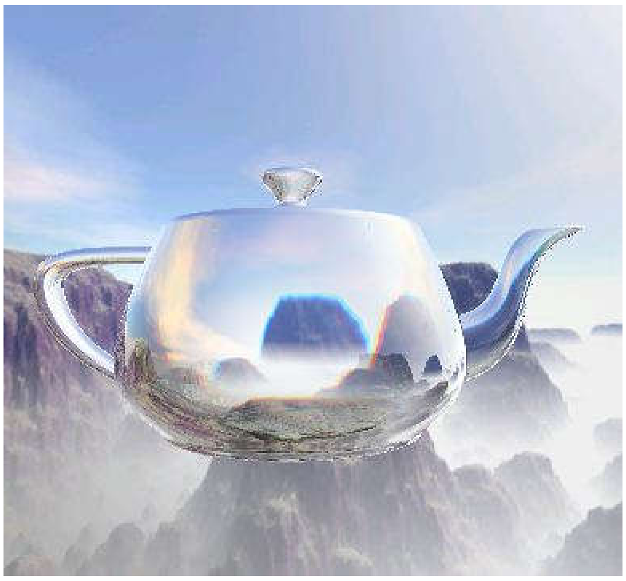

Figure 15 - Chromatic dispersion