Computer Graphics 02561 - Project Report

Alberto Zanatta - s182264

21st December 2018

Contents

1 Introduction

As my final project, I chose to complete the course's last assignment and further expand on it by implementing some other related rendering techniques.
My work was therefore concerned with understanding the basic principles behind Environment Mapping and use such concept to render chrome-like and glass-like objects by simulating related physical phenomena such as reflection, refraction, the Fresnel effect and chromatic dispersion.

1.1 Environment mapping

Scenes can feature a collection of either reflective or refractive objects featuring different kind of materials. To make them appear life-like lighting-related phenomena such as reflection and refraction need to be simulated realistically.
Unfortunately, accurately simulating these phenomena requires ray-tracing, which is computationally very expensive and as a result not suitable for real-time applications [2].
Environment mapping is a technique that approximates the results of ray-tracing allowing to create the illusion of reflections and refraction. As it is can be implemented using texture mapping supported by graphics hardware, it can obtain such lighting results in real time.

1.2 Environment maps

The environment surrounding an object is assumed to be infinitely distant and is encoded in an omnidirectional image called **environment map**.
Environment mapping is basically the process of pre-computing and encoding an environment map and then sampling texels from it during the rendering of an object [3].
The steps involved are quite straightforward. First the environment map is loaded, then for each vertex/fragment the reflective/refractive ray is computed and used to access the environment map and retrieve the shade of the fragment.
One method to represent an environment map is through a **cubemap**, a collection of six disjoint, squared textures representing the faces of an imaginary cube enclosing the object to be rendered. Each face can be considered as a 90 degree snapshot of the environment with square aspect ratio taken by a camera placed at the center of the object and aligned along the directions of the world axes.
Unlike traditional 2D textures, a cube map - supported by virtually all graphics card - is sampled using a three component vector representing a direction.

1.3 Shortcomings and advantages

As environment mapping assumes the environment to be infinitely distant and environment maps are sampled using only directional vectors, moving the object across the scene does not lead to any variation in rendering results. This can lead to minor visual artifacts when the surroundings the environment map encodes are not sufficiently distant from the object [1].
Furthermore, through this technique self-reflections and inter-object reflections cannot be achieved. As a consequence, it works well if there is a single reflective/refractive object in the scene.
Despite these limitations, in practice environment mapping allows to create plausible and convincing lighting effects as many of its visual artifacts are typically difficult to spot at first glance and are often overlooked.

2 Method

First of all, I completed all the four parts of Worksheet 10 to get confident with the basics of environment mapping and its implementation.

2.1 Load and access and environment map

To initialize a cube-map correctly, each of the six squared textures representing the surrounding environment need to be uploaded to the right face.
As in a WebGL application images are loaded asynchronously and independently of the order in which they were originally instantiated, I decided to store within each image object a reference to the cube-map and to the cube-map face it belongs to - the latter as an integer index ranging in the $[0, 5]$ interval.
When the `onload` callback is invoked, the image is retrieved from the `event` variable. Then, the cube-map reference is used to activate the right texture unit and the face integer is used to put the image into the right cube-map's orientation.
The resulting three-dimensional environment texture is sampled using the world space normals of the sphere object placed at the center of the scene. In the vertex shader I take advantage of the fact that for a sphere centered at the origin, the normal at a point p is simply p. As a result, the vertex position can be used as a direction for the sampling of the environment map.
Furthermore, as we are only concerned with the *direction* of a vector when accessing the cube-map and not its magnitude, no normalization is required.
I extended this part by adding the possibility to choose between two different cube-maps and to make the camera spin around over time.
The results of this implementation are shown in figure 1 and in figure 2.

2.2 Draw the environment in the background

To show the surrounding environment on the background of the rendered scene, I drew a quad very close to the far end of the camera's view frustum using clip-coordinates.
Then, I transformed the clip coordinates back into world direction vectors and used them to look up the cube-map and texture the background plane accordingly.
This transformation is encoded in the M_{tex} matrix calculated by multiplying the inverse of the projection matrix used to render the sphere and the rotational part of the view matrix.
The inverse of the projection matrix allows to transform clip coordinates into eye coordinates; finally, the second part of the transformation - which involves the inverse of the rotational part of the view matrix - allows to retrieve the direction vector pointing from the origin in eye space - that is where the camera is placed - towards a specific position in the background quad. We do not include the translation part of the matrix because we are not interested in the world space coordinates of such position in the background, but only in the direction vector which points towards it from the eye.
Direction vector that is then used to look up into the cube map.
The vertex shader for this part is the shared by both the sphere and the background quad.
Naturally, when drawing the background both the model-view and the projection matrices are set to the identity matrix - as vertex positions are already in clip space; when drawing the sphere it is the M_{tex} matrix which is initialized as the identity.
The results of this implementation are shown in figure 3 and in figure 4.
The possibility to spin the camera around the sphere allows to further check for the correctness of the code.

2.3 Render the reflective surface

After implementing the loading of the cube-map and the drawing of the background, I worked on the rendering of the sphere's reflective surface.

When we look at a reflective, chrome-like object what we see is not the object itself but rather how it reflects the surrounding environment.
A simple and effective - even not physically accurate - way to frame the problem is the following.
When looking at a point on a reflective surface the **incident ray (I)** pointing from the eye towards such position is reflected into the **reflected ray (R)** according to the **surface normal (N)** and cast in the environment. The equation for the calculation of such ray is the following: $R = I - 2 * N * dot(N, I)$ and is implemented by a standard `GLSL ES` function, `reflect`.
In my shaders the actual reflection calculations are performed within the fragment shader.
The vertex shader is responsible for assigning the normal direction and vertex position in world coordinates to two varying `vec3` variables - called `normal` and `position` respectively. The interpolated values are then passed to the fragment shader where the eye position in world coordinates is stored within a uniform `vec3` variable named `eye`.
Here, first of all I normalize the interpolated normal vector - a step that even though not strictly necessary, does not impact on the correctness of the final result.
Then the `incident` vector is calculated through a simple subtractions between the `position` and `eye` varyings and passed as a parameter to the `reflect` function together with the normalized normal.
The resulting reflected vector is used to look up the environment map and retrieve the fragment's color shade.
Both the `incident` and the `reflection` vectors do not need to be normalized. Regarding the former, this is due to the fact that we normalize the normal vector before passing it to the `reflect` function. Regarding the latter, as it is used to look up a cube map, we are interested only in its direction and not in its magnitude.
Keeping the eye, the fragment position and the interpolated normal in world coordinates rather then in eye coordinates allows to avoid cumbersome transformations from eye space back to world space - as the querying of the cube-map always requires world space directions.
Finally, performing the reflection calculations on a per-fragment basis allows for a higher quality final result, even though it is computationally more expensive.
The results of this implementation are shown in figure 5 and in figure 6.

2.4 Bump mapping

Bump mapping is a technique used to achieve a greater superficial detail for an object without increasing further its polygon count [8]. It usually relies on normal-maps used to perturbate the surface normals of the object when performing lighting calculations.
To implement such technique I first queried the normal map using the right `u` and `v` values to get a `vec4` texel. Then I remapped all the components of the texel in the range [-1, 1] to obtain the actual perturbated normal.
Finally, I used such normal instead of the real one for calculating the reflected vector, eventually used to look up the environment map. The results of this implementation are shown in figure 7 and in figure 8.

2.5 Expanding the project

After completing all the four parts of the worksheet, I decided to expand on the project by implementing other interesting physical phenomena that could be simulated using environment mapping.

2.5.1 Refraction

The physical phenomenon Refraction can be framed in the same way as reflection.
When the **incident ray (I)** originating from the observer's eye meets the surface of a refractive object whose material has a different density than that of the medium containing I, it is *refracted* and its direction changes into the **refracted ray (R)** in accordance to Snell's law. This equation states that

the sin of the angle between I and the **surface normal (N)** multiplied by the index of refraction of the medium containing I is the same as the sin of the angle between R and N multiplied by the index of refraction of the medium containing R.

The shader implementation The shader program implementing refraction shares the same vertex shader as the shader program implementing reflection.
Within the `sphere-refraction-fragment-shader`, I use the `GLSL ES refract` function to compute the refracted ray used to query the cube map. The function requires three parameters: the incident ray, the surface normal and the ratio of the refractive indices of the two media involved.
The incident ray is calculated in the same way as for reflection but it is normalized before passing it to the `refract` function because of the way it is implemented.
Within the shader, the uniform `float` variable `etaRatio` is the one storing the ratio between the index of refraction of air- the medium containing I -, 1.0003, and the index of refraction of Crown glass, - the material of the refractive sphere, 1.52. [5].
The results of this implementation are shown in figure 9 and in figure 10.

Comments and assumptions Storing the ratio of the refractive indices instead of their single values saves the fragment program from having to calculate it for each fragment.
Finally, the way refraction is simulated is not physically accurate. In fact, it does not take into account the second refraction of the incident ray - the one happening when passing from the glass sphere into the open air again. Fortunately, as the phenomenon is quite complex the end result is still convincing and life-like [1].

2.5.2 The Fresnel effect

The physical phenomenon Transparent objects such as glass are both refractive and reflective. The amount of reflected light and the amount of refracted light both depend on the angle between the incident vector and the surface normal and can be accurately computed using the Fresnel Equations - which quantify the *Fresnel effect* [4].
Implementing the Fresnel effect allows to increase the realism of rendered objects, as it allows to achieve a more realistic mixing of refraction and reflection.

The shader implementation The shader program implementing the Fresnel effect shares the same vertex shader as the shader program implementing reflection.
Within the fragment shader both the reflected and the refracted rays are computed and used to query the cube-map.
The reflected color - `cReflected` - and the refracted color - `cRefracted` - are then combined in the following way:
`gl_FragColor = reflectionCoefficient * cReflected + (1.0 - reflectionCoefficient) * cRefracted`
Where `reflectionCoefficient` is a value ranging in [0, 1] computed as:
`float reflectionCoefficient = max(0.0, min(1.0, fresnelBias + fresnelScale * pow(1.0 + dot(incident, nNormal), fresnelPower)))`
These two formulas combined allow to achieve a significantly less complicated yet very convincing empirical approximation of Fresnel equations.
The results of this implementation are shown in figure 11 and in figure 12.

Comments and assumptions Computing the reflection coefficient requires the specification of three different values : the fresnel scale, the fresnel bias and the fresnel power. By default they are set to 1, 0 and 5 respectively [7], but can be tuned using three different sliders available when loading the project's WebGL application.

2.5.3 Chromatic dispersion

The physical phenomenon When a ray of white light enters or exits a refractive object, it is split into several rays as its color components are refracted along slightly different directions according to their wavelength. This phenomenon is called *chromatic dispersion*.

The shader implementation Within the fragment-shader of the program implementing chromatic dispersion, the uniform `vec3` variable `etaRatios` stores the ratio of the refractive indices for red, green, and blue light.
The computation of the reflected vector and reflected color is left unchanged.
Then, for each main color component - red, green and blue, the program performs the following steps:

- computes the refracted direction of the color, given the normalized incident vector, the surface normal and its own ratio of indices of refraction;
- uses such refraction vector to perform a cubemap lookup and sample a texel;
- extracts the texel component corresponding to the color and use it to initialize the same component in the refracted color.

Finally, the program calculates the reflection coefficient to blend the reflected and refracted colors into the shade of the fragment.
The results of this implementation are shown in figure 13 and in figure 14.

Comments and assumptions The above-mentioned shader program simulates the chromatic dispersion effect only for the red, green and blue components of light as they are the standard color components in digital images. However, this phenomenon affects all the wavelengths of the incident illumination.
The values used as the refractive indices for red, green and blue light in Crown glass were found at this page [6].
The same page suggests that in air there is no meaningful difference between the indices of refraction of different light wavelengths.

2.5.4 The final application

All the shader programs discussed in the previous sections have been integrated into a single WebGL application which consists of both a `.html` file - `worksheet10app.html` - and a `.js` file - `worksheet10app.js`.
The program allows to seamlessly change between:

- two different cubemaps, one depicting a mountainous, canyon environment and one set within the courtyard of a house;
- four different normal maps to perturbate the normals of the sphere placed at the center of the scene;
- four different shader programs implementing reflection, refraction, Fresnel effect and chromatic dispersion.

Furthermore, three sliders allow the user to tune the parameters affecting the computation of the reflection coefficient in the Fresnel and chromatic dispersion programs.
I think this program combines effectively what I have learned throughout the whole course in terms of interactive programs and graphical applications.
I decided to have a dedicated shader for the drawing of the background and a single vertex-shader for

all the programs affecting the rendering of the sphere. Naturally, the fragment shaders are different, but all rely on the same code for sampling the normal map and calculate the perturbated normals. In order to keep the shaders as simple as possible, this 'utility' code has been moved inside another script tag in the html document and separately compiled and added to each program in the javascript file.
The way the code is written allows to easily add and remove cubemaps and normal maps by specifying their source files before the definition of the `init` function. No more than eight cube-maps/normal maps should be loaded at the same time.
The visible results of this implementation are shown in figure 15.

3 Results

3.1 Load and access the environment map

Figure 1: Texturing of the unit sphere using the 'Courtyard' environment map

Figure 2: Texturing of the unit sphere using the 'Landscape' environment map

3.2 Draw the environment in the background

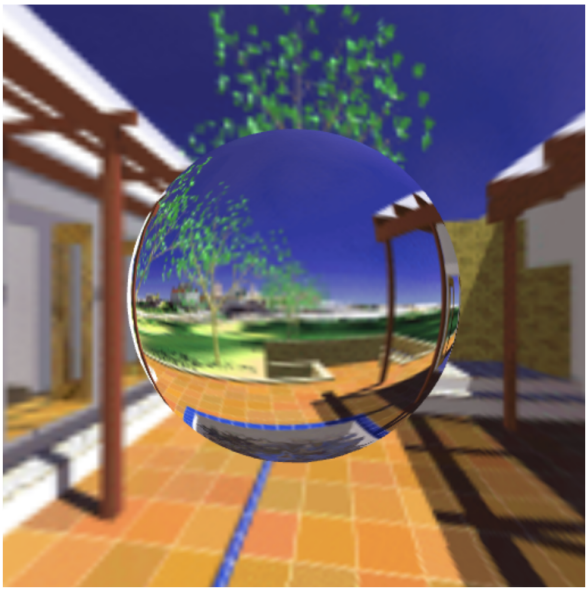

Figure 3: Background texturing using the 'Courtyard' environment map

Figure 4: Background texturing using the 'Landscape' environment map

3.3 Render the reflective surface

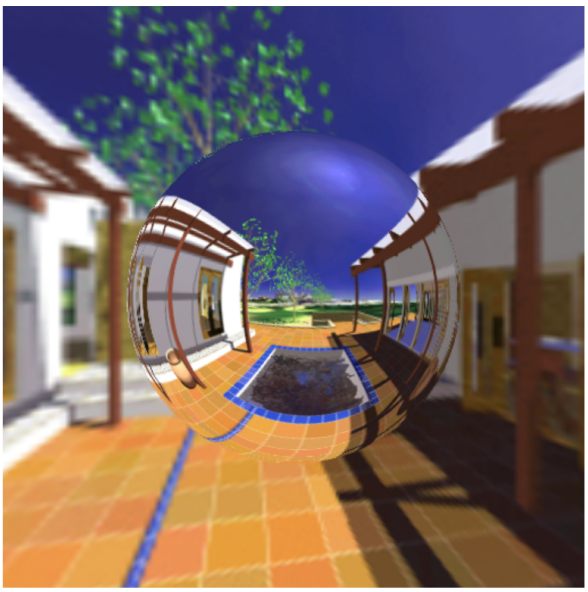

Figure 5: Reflection using the 'Courtyard' environment map

Figure 6: Reflection using the 'Landscape' environment map

3.4 Bump mapping

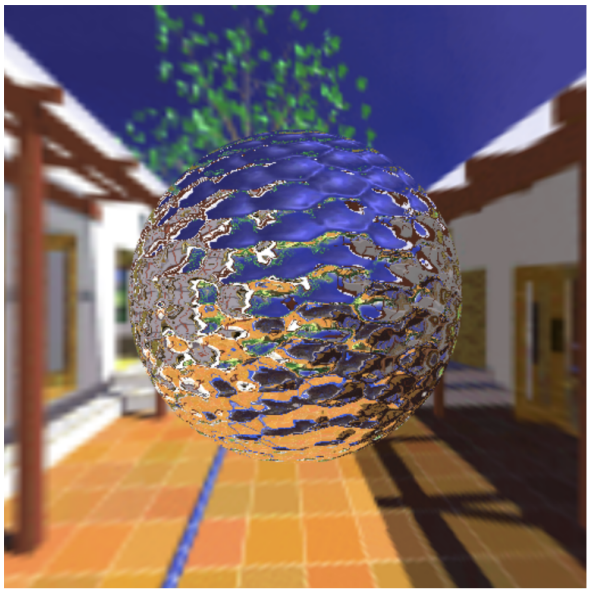

Figure 7: Bump mapping using the default worksheet normal map within the 'Courtyard' environment

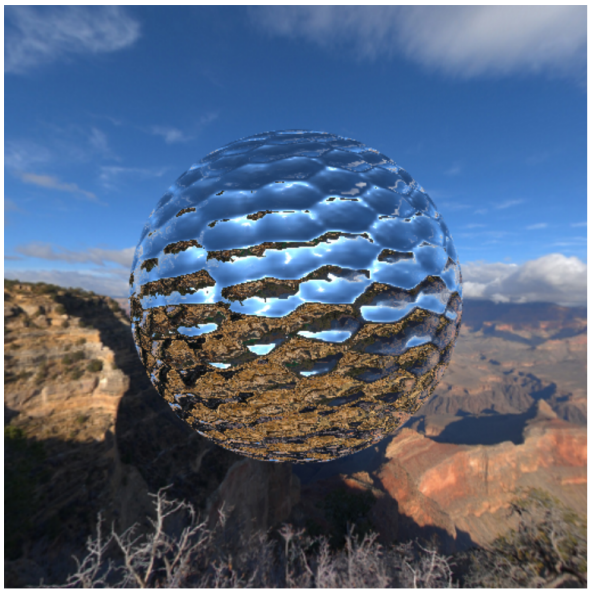

Figure 8: Bump mapping using the default worksheet normal map within the 'Landscape' environment

3.5 Refraction

Figure 9: Refraction using the 'Courtyard' environment map

Figure 10: Refraction using the 'Landscape' environment map

3.6 The Fresnel effect

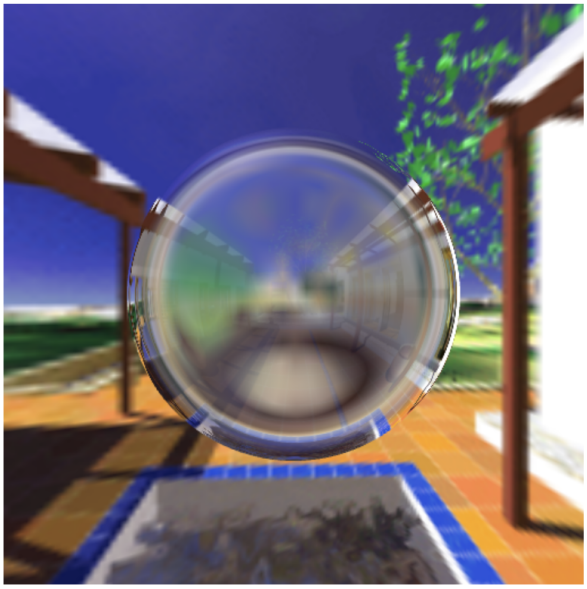

Figure 11: Fresnel effect using the 'Courtyard' environment map

Figure 12: Fresnel effect using the 'Landscape' environment map

The values used for the three Fresnel parameters are the following: Fresnel Scale = 4, Fresnel Bias = 0.1 and Fresnel Power = 5.

3.7 Chromatic dispersion

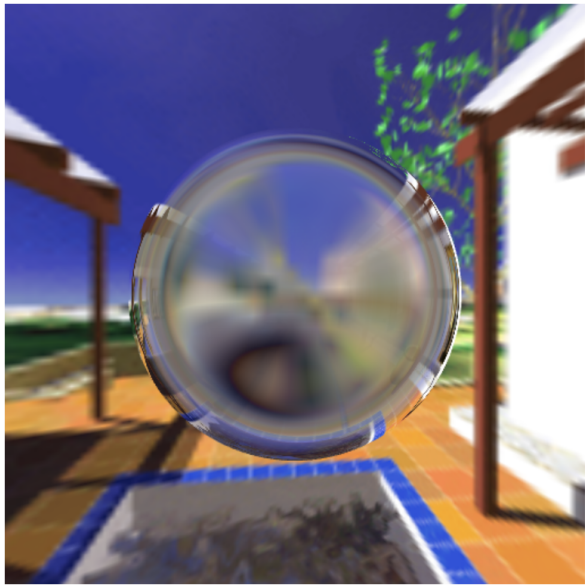

Figure 13: Chromatic dispersion using the 'Courtyard' environment map

Figure 14: Chromatic dispersion using the 'Landscape' environment map

The values used for the three Fresnel parameters are the following: Fresnel Scale = 4, Fresnel Bias = 0.02 and Fresnel Power = 5.

3.8 The final application

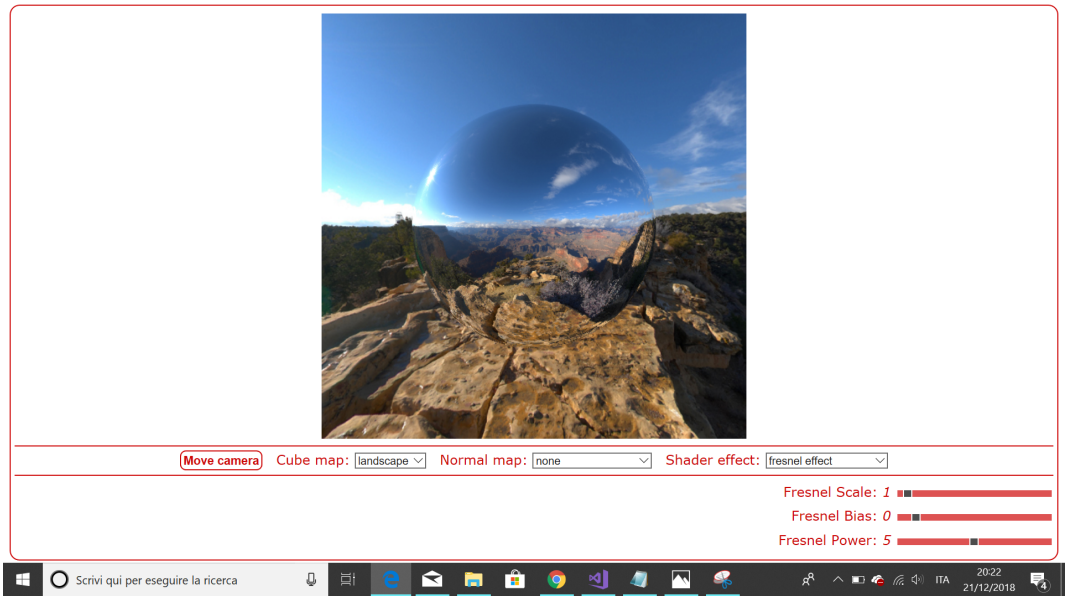

Figure 15: Screenshot of the complete application

4 Conclusion

Most of the results are aligned with what I expected.
In part 1 and 2 of the final worksheet the spinning of the camera shows that the cube-map is set up correctly and that the background quad is textured using the right direction vectors to query the cube-map - it moves as the eye rotates around the central sphere.
In the third section of the assignment the sphere rendered using the reflection shader program matches both the feedback-picture in the pdf file detailing the instructions for the worksheet and the appearance of a chrome-like object reflecting the environment it's facing.
Finally the normal-mapping implemented as part of the final part of the worksheet correctly gives the impression of a 'waving' surface which is more detailed, complex and dramatic than real smooth surface of the central sphere.
The rendering obtained via the refraction shader program seems to correctly distort the environment behind the refractive sphere. However, the final result is quite dull and lacking the expected clarity. I think the main reason for that is the low quality of the textures that make up the environmental maps.
Also the Fresnel effect looks correct to me. As expected, the area of the sphere which is directly facing the camera is almost perfectly refractive, while along the borders - where the angle between the incident ray and the surface normal grows bigger - it is possible to see the surrounding environment being reflected.
Finally it is also possible to spot the subtle changes in the image caused by the implementation of chromatic dispersion. Comparing 13 against 14 shows that in the latter there are some visible red-ish and green-ish shades of color across the refracted image within the sphere which are the results of the three main color components - red, green and blue - being refracted along slightly different directions.

References

[1] The cg tutorial. chapter 7 - environment mapping techniques. http://developer.download.nvidia.com/CgTutorial/cg_tutorial_chapter07.html.

[2] Computer graphics: Environment apping and mirroring. http://www.inf.ed.ac.uk/teaching/courses/cg/lectures/slides9

[3] Environment mapping. https://fenix.tecnico.ulisboa.pt/downloadFile/1970943312277245/2%20-Environment%20mapping.pdf.

[4] Introduction to shading (reflection, refraction and fresnel). https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-shading/reflection-refraction-fresnel.

[5] Refractive index. https://en.wikipedia.org/wiki/Refractive_index.

[6] Torinoscienza. indice di rifrazione. http://archivio.torinoscienza.it/glossario/indice_di_rifrazione_5290.html.

[7] Unity products:amplify shader editor/fresnel. http://wiki.amplify.pt/index.php?title=Unity_Products:Amplify_Shader_

[8] Dave Shreiner Edward Angel. *Interactive computer graphics*. Pearson, 2015.