Normalmap based Refraction Shader

Julia Angerer Parzival Legmaier Andreas Pointner

Abstract

As part of the course *Shader Programming* this shader project was developed, which deals with refraction based on normal maps. Normally this effect is implemented using cubemaps, this approach however utilizes the framebuffer and a normal map to refract the scene. Both methods are not physically correct but efficient and optical plausible enough for realtime applications. The option to use the framebuffer with a normal map for a refraction has the advantage that in *Deferred Rendering*, the needed information is mostly already available.

1 The original idea

The refraction describes the bending of light that occurs when light travels from one medium to another one with a different refractive index, as an example from air to water or glass [1].

To nearly achieve physically correct refraction raytracing is needed, but this would be computationally intensive for realtime applications. The higher the raytracing-depth, the more expensive the calculation gets.

For realtime applications other techniques are normally used that recreate refraction effects that look convincing enough in comparison to the physical correct approach.

1.1 Idea

There are several different approaches to calculate refraction for realtime applications. An often used method is to use a cubemap which supplies the refraction data.

Another approach, which is realized in this project, is to use the framebuffer as a source for the refraction. Based on a normal map the texture of the framebuffer is distorted, so different effects can be achieved by using various normal maps.

The advantage of the normal map based refraction is that the whole scene can be refracted. Since the cubemap is pre-calculated, this method can only refract objects which are already in the particular cubemap.

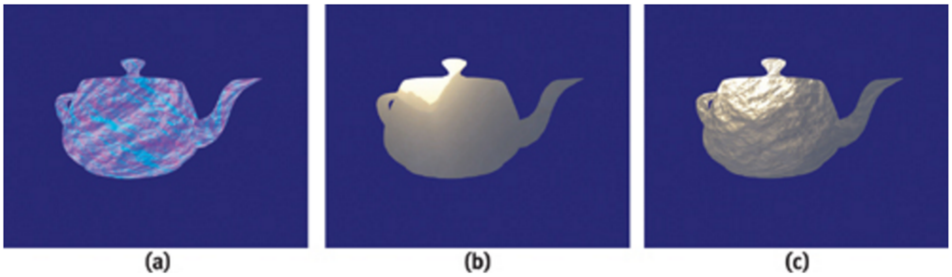


Figure 1: This figure shows the visualization of the rendering steps: (a) Fetch the normal map, (b) Fetch the framebuffer texture, (c) Perturb the framebuffer texture based on the x and y components of the normal map.

[1]

1.2 Planned implementation

A basic OpenGL-project that can handle input, load models and shaders, draw a skybox and render to texture in order to create a framebuffer need to be set up. For the basic structures of the different parts of the implementation the website *LearnOpenGL* [2] was used.

The shader part of this project will be implemented with help of *GPU Gems 2* [1] which gives a detailed description of the general idea shown in figure 1.

2 Implementation

The implementation of this shader project divides into two parts, the first one being the setup of the OpenGL base project. The second part is the implementation of the actual shaders used to achieve the refraction effect seen in figure 2.

2.1 Base project

The following steps were implemented:

- **Basic code structure**: This includes the basic code structure (update loop, rendering, input handling, ...).
- **Model loading pipeline**: To easily load models automatically based on their material files with the corresponding textures (diffuse, specular, normal) a model loading pipeline was established.
- **Shader class**: To handle shaders separately a shader class was defined in order to initialize, compile and use different vertex and fragment shaders.

- **Camera**: To view the whole scene a free-fly camera system was implemented, so the user can easily look at the refraction effect.
- **Skybox rendering**: In order to make the refraction look more conceivable a skybox is always rendered in the background. The skybox is also used to reflect the scene.
- **Framebuffer**: The framebuffer class defines methods to initialize, bind and unbind the buffer easily.

2.2 Shader

Standard shader For all non refracting objects a standard shader was defined which handles blinn or phong lighting, with or without textures. Additionally the shader implements gamma correction and calculates light attenuation.

Skybox shader The skybox is rendered without lighting and is always drawn to the background.

Framebuffer shader The whole scene is rendered on a quad without lighting with the framebuffer shader.

Refraction shader The basic structure and the lighting is based on the standard shader. In order to achieve the refraction effect several inputs are needed:

- Framebuffer texture
- Normal map of the model (material file)
- Refraction parameters (strength, chromatic dispersion)

Based on the x and y components of the normal map the framebuffer texture is perturbed to achieve the refracted look (as seen in program 1).

After the calculation of the refraction color the lighting information, the gamma correction and the light attenuation is added. Because the refraction is based on the specific normal map and the lighting is not, the lighting looked incorrect. To resolve this issue new normals based on the normal map were computed in the fragment shader using a tangent based calculation.

To make the whole effect more convincing a normal map based chromatic dispersion was added. This method was achieved by shifting the framebuffer by a small value in different UV directions.

```
1     // Final refraction color (With chromatic dispersion)
2     vec3 refractionColor = vec3(texture(framebuffer, newUV -
      chromaticDispersion).r, texture(framebuffer, newUV).g, texture(
      framebuffer, newUV - chromaticDispersion).b);
```

Program 1: Sampling of the normal map and the framebuffer and computing the final refraction color by perturbing the UV-coordinates of the framebuffer based on the normal map.

```
1     // Sample normal map
2     vec4 normalMap = texture(texture_normal1, UV.xy);
3
4     // Refraction
5     vec2 bumpUV = vec2(gl_FragCoord.x / width, gl_FragCoord.y / height);
6
7     // Unpack from [0..1] to [-1..1]
8     vec4 bump = 2.0 * texture(texture_normal1, UV.xy) - 1.0;
9
10    // Displace texture coordinates
11    vec2 newUV = bumpUV + bump.xy * refractionStrength;
12
13    // Non pertubated framebuffer texture
14    vec4 frameTexture = texture(framebuffer, bumpUV);
15
16    // Final refraction color (With chromatic dispersion)
17    vec3 refractionColor = vec3(texture(framebuffer, newUV -
      chromaticDispersion).r, texture(framebuffer, newUV).g, texture(
      framebuffer, newUV - chromaticDispersion).b);
18
```

2.3 Interaction parameters

The demo project has following input parameters:

- Camera controls: WASD
- Light-X: F/G
- Light-Y: H/J
- Light-Z: K/L
- Refraction strength: R/T
- Refraction/Color blending: N/M
- Chromatic dispersion: C/V

3 Evaluation

The plan was successfully executed, but it took a little longer than expected. The anticipated amount of 30 hours turned to about 40 hours but in this process we achieved more than actually planned (chromatic dispersion). Still there are a few remaining problems that could be resolved.

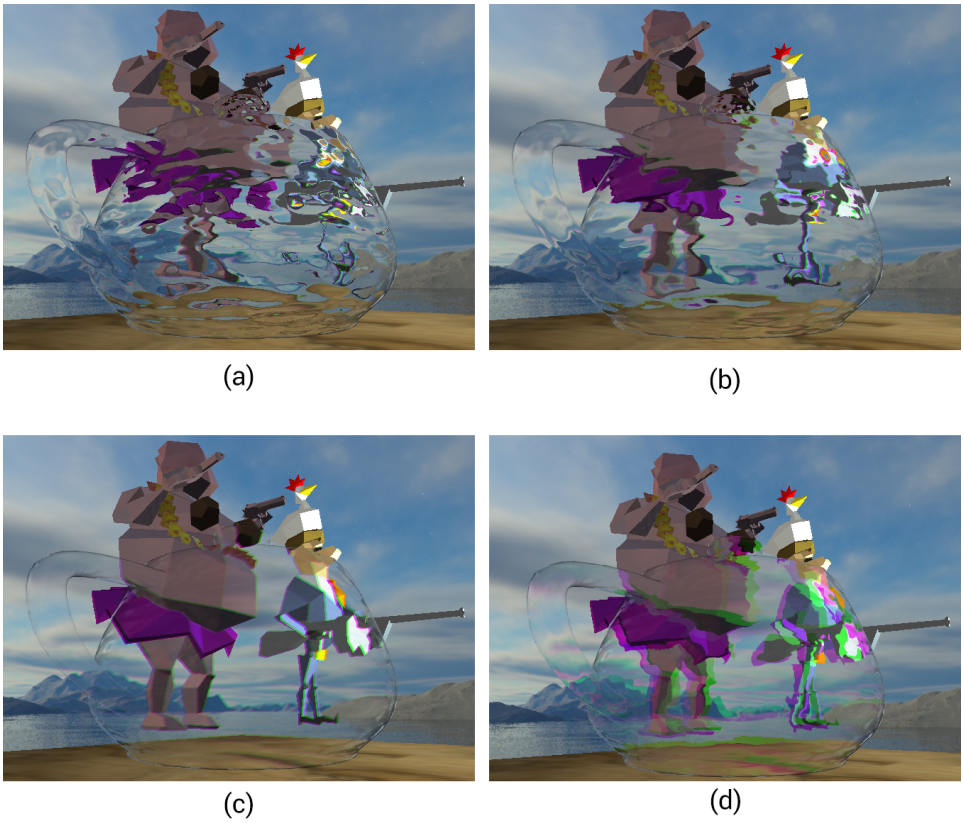


Figure 2: This figure shows different refraction strengths in (a), (b), (c) and chromatic dispersion in (d).

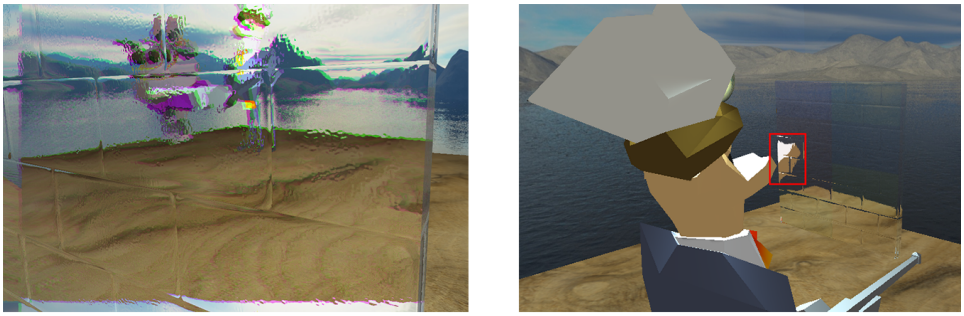
3.1 Accomplishments

The implementation of the base project according to the website *LearnOpenGL* [2] worked out well. The most time was spent on the framebuffer class, because the depth buffer was lost after unbinding. The depth buffer was crucial because all refractive objects were drawn after the framebuffer and without the depth they always remained in the foreground.

The solution was to write the depth manually into the framebuffer with the function `glBlitFramebuffer()` before unbinding.

```
1  glBindFramebuffer(GL_READ_FRAMEBUFFER, framebuffer);
2  glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);
3  glBlitFramebuffer(0, 0, width, height, 0, 0, width, height,
     GL_DEPTH_BUFFER_BIT, GL_NEAREST);
```

Due to the course *Shader Programming* lighting was no problem to implement as well as the skybox and the framebuffer shader. The refraction shader was as estimated the hardest part because of the following factors:



(a) (b)

Figure 3: This figure shows different artifacts: (a) Border artifacts because the tiling of the framebuffer texture, (b) Foreground artifacts

- Loading normal map (modify model loading pipeline)
- Loading framebuffer texture (general render to texture problem)
- Locating the UV-coordinates of the framebuffer (due to deprecated GLSL documentation we thought that glFragCoord values are given in a relative way, while in fact they are given absolute)
- Tangent space calculation for lighting (if the lighting is not based on the normal map too, it looks incorrect)

3.2 Remaining problems

However, the above mentioned problems could be solved while others still remain:

- Multisampling in the framebuffer (implementing multisampling in a custom framebuffer is rather complicated and was not solved due to time problems)
- Border artifacts (because the framebuffer texture is tiled, it is possible to see corresponding opposite edges of the texture in the refraction when parts of the refractive objects are cut by the viewport, as seen in figure 3 (a))
- Foreground artifacts (In the framebuffer all objects but the refractive ones are drawn, so in the refraction process it is not possible to distinguish between objects behind and in front of the refractive one, as seen in figure 3 (b). Actually there is a solution for the problem [1] which was not implemented successfully due to time problems.)

References

[1] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005 (cit. on pp. 1, 2, 6).

[2] Joey de Vries. URL: http://learnopengl.com/ (visited on 01/11/2016) (cit. on pp. 2, 5).