An Interactive Web Gallery for Goal-based Caustics

Jad-Nicolas Khoury*
Supervisor: N. Thanikachalam
Professor: Dr. Mark Pauly

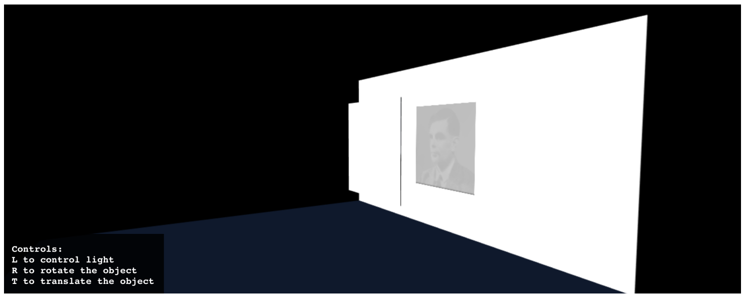
Controls:
L to control light
R to rotate the object
T to translate the object

Figure 1: *Rendering with the caustic lightmap on the client side*

Abstract

Any product or technology, if destined to be distributed, needs good visibility on the market, in particular to the potential buyers.
For a long time now, web technologies and web-design APIs allowed companies of all sizes to display their ideas, products and technologies to virtually any person with an internet connection, in a more intuitive, aesthetic, detailed and simple form, all the while being easier to implement. But for some products, images and text alone don't make justice to the advantages and appeal of the merchandise.
Video streams can help, but implementing a web application offering interactivity to the potential buyers is sometime worth the time and effort. For this reason, Rayform wanted to develop an online WebGallery to showcase some of their technologies.

1 Introduction

The main goal of this project is to develop an interactive web-gallery displaying the real-world effect of Rayform products. These products are either reflective or refractive objects which, when in the right light conditions and position, project as a caustic pattern a pre-defined image.
By the nature of these objects, this implies the implementation of an extremely efficient ray tracing simulation engine which should be able to run in real time. As an additional constraint, the original object meshes should not be accessible by the user, to avoid the possibility of someone reverse-engineering the technology. This implied deporting part of the rendering process to a server that is not accessible to the user.

2 Implementation

The rendering process is distributed as follows:
- The GPU server should access the real caustic objects mesh, run the ray-tracing engine, and stream the caustic pattern obtained to the client side.
- The client side should render the environment, the light, the helpers, and all necessary interface, as well as a low resolution mesh representing the caustic object without the surface modification that encapsulate the caustic technology. Receiving the texture stream from the GPU server, the client side should unwrap it and superpose the texture to the ”screen” on which the caustic pattern is projected.

As this application is supposed to render in real-time, two pipelines have been explored:
First, a synchronous pipeline, based on the assumption that it could be possible to wait for the caustic pattern to arrive on the client side before rendering. The advantage of this implementation is that the caustic pattern shown in the client side correspond exactly as the one resulting from the current scene setup (i.e. object rotation, position ...). The limit being that the whole round-trip time could be too long to get a smooth rendering when displacing or rotating the object or the light.
Second, an asynchronous implementation that should result in a smoother experience. The server side continuously render the caustic pattern, taking the parameters update into account, and streaming it directly on some public link. The client side then continuously read this stream and use it directly as a texture. While being easier to implement and resulting in a smoother experience, this pipeline could induce a delay between applying a transformation to some object of the scene and the impact visibility of this transformation on the caustic pattern. This is currently the pipeline being implemented.

2.1 Server Side

2.1.1 Algorithm Description

This section will describe the general caustics simulation algorithm, running on the server side, without going into computational details.
Considering that 30 frames per seconds - all rendering included - is a good objective, this leaves approximately 33ms for the whole loop to run: the client side computations, the transmission of the parameters to the GPU server, the computations of the caustic pattern, the transmission of the resulting texture, the blending of the said texture and finally the full rendering display. For this reason, the Ray-Tracing engine should be extremely efficient. Since it has been specifically designed to compute caustic patterns projected on a screen, it was possible to develop shaders restricted in their applications, but very cost-effective, thanks in particular to OpenGL Geometry Shaders (see Fig. 2). More concretely, rendering the projected caustic pattern is a problem equivalent

*e-mail:jad-nicolas.khoury@epfl.ch

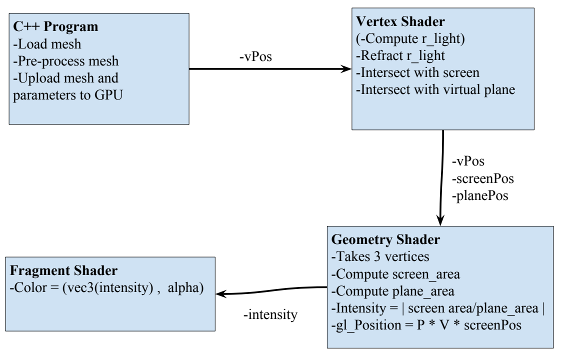

Figure 2: *Overview of the Caustic Simulation Pipeline*

to determining where the mesh focuses light on the screen and where it disperses it. One can think of the refractive surface as an assembly of many converging and diverging triangular lenses.
The first step to design such Ray Engine has been to discretize the rays traced. The algorithm traces exactly one ray per vertex of the refractive mesh. Even though that could seem a drastic compromise, one can notice that in our application, all the rays intersecting the refractive surface in the same triangle will be refracted as parallel rays, and therefore will result in the same ray convergence/divergence (as the triangle is flat by definition).

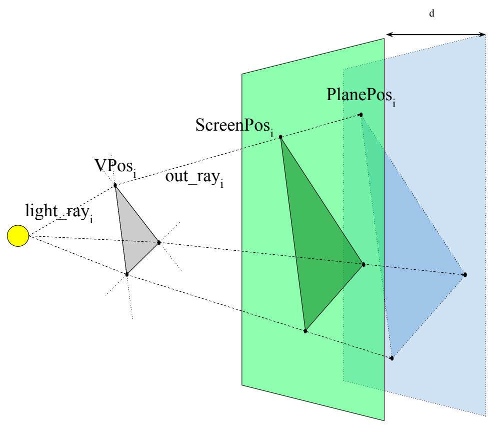

Figure 3: *Visualisation of the caustics simulation algorithm*

The algorithm designed to simulate the converging/diverging of the light rays, has been inspired by [Wallace 2016] and was optimized using Geometry Shaders.
As illustrated on Fig. 3, the Vertex Shader computes for each vertex of the refractive mesh the refracted out-ray. Then, it intersects this resulting ray with a plane representing the screen on which the caustic pattern is displayed (here represented in green), and then on a virtual plane, parallel to the screen, and placed behind it with regard to the light (here in blue). These planes are stored as pairs $(point\_on\_plane\,,\,plane\_normal)$. For each vertex, the Vertex Shader passes the following geometry to the Geometry Shader: the vertex actual position, the point where the out-ray intersect with the screen, and the point where it intersect the virtual plane. The Geometry Shader access the 3 points of each triangle of the mesh, and then use the three passed screen-positions to compute the projected area of the triangle on the screen; and repeats the same with the virtual plane. This algorithm has the advantage of treating each ray and each triangle exactly once. The diverging/converging of the ray by the given triangle is then easily (and roughly) approximated by the ratio of these areas, i.e.:

$$Intensity = |Area_{screen}/Area_{plane}| \qquad (1)$$

This value is then passed to the Fragment Shader where it is used to give a monochrome colour to the triangle on the screen. At first, the virtual plane was supposed to be placed as close as possible to the mesh. Intuitively, comparing area on this plane to the area on the screen would be perfect to deduce the diverging/converging of the light. But this idea is complex to implement first because the refractive surface isn't flat (and the plane should not intersect the mesh), but also because it should be displaced whenever the mesh rotates, translates etc ... Placing the virtual plane close to the screen and behind it allows us to leave it at a fixed position, while never taking the risk that the object could intersect it.
Recall that, for now, the screen is only a pair $(point\_on\_screen\,,\,screen\_normal)$, and the mesh is at its original position (see section 2.1.5). There is therefore nothing to render on the screen! The first approached solution was to render the screen as a grid and displace all the vertices to match the positions of the projected points. But this approach had many problems: first, we would need a correspondance point-on-screen to point-on-mesh, which is virtually impossible. Then, we would need to access each point of the mesh while rendering the grid of the screen, and so pass two objects to the shaders when rendering. But the shaders are applied in the same fashion to the all points passed to them, and this would add further complications. The trick found to answer this problem was to actually never render the screen itself, but rather render the mesh, and then ”cheat” by assigning to the vertex position the computed projected position on the screen. To render the caustic pattern and take into account its position on the screen, a simulated camera is placed in front of the screen, with orthographic projection and adapted parameters.
This algorithm is summarised in Fig. 2.

2.1.2 Refraction Algorithm

Until now, the refraction of the light ray by the caustic surface has been evoked as a simple operation, but it requires two steps in reality. As the caustic object is not just a surface but a glass solid with a flat side and a ”caustic” side, the incoming light-ray has to be refracted twice. A first time as it passes from the air medium (n_1) to the glass medium (n_2), and a second time when it comes out of the object (n_2 to n_1). In the figures discussed, the surface geometry as well as the refraction angles are exaggerated and not to scale, and the vectors are not normalized, in an effort to make them as clear and explicit as possible.
We distinguish two cases in our application:
The simplest one is with directional light, pictured in Fig. 4. The light source is considered to emit parallel light rays, giving uniform light rays across all the mesh vertices. This incoming light ray is passed directly to the vertex shader which first compute the refraction of this ray by the flat side, giving r_{inside}. r_{inside} is then refracted again at the vertex position, using the vertex normal, resulting in the final out-going ray $r_{outside}$.
In the case of Point Light (or Spotlight), light rays are not parallel anymore, and therefore change for each vertex. As seen on Fig. 5 in red, the light ray is computed as $vertex\_pos - light\_pos$. This ray is then refracted on the flat surface, and then refracted again using the vertex normal. But as the figure shows, there is a problem: the ray refracted by the flat surface, shown in purple, does not actually hit the caustic surface at the exact vertex position, but is used nonetheless to compute the outgoing ray. To have the exact

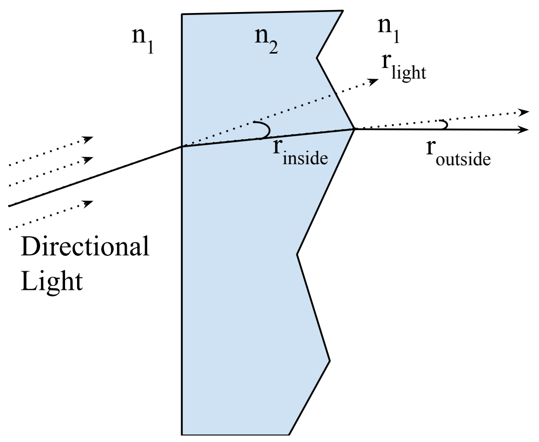

Figure 4: *Representation of the details of the refraction computations with directional light*

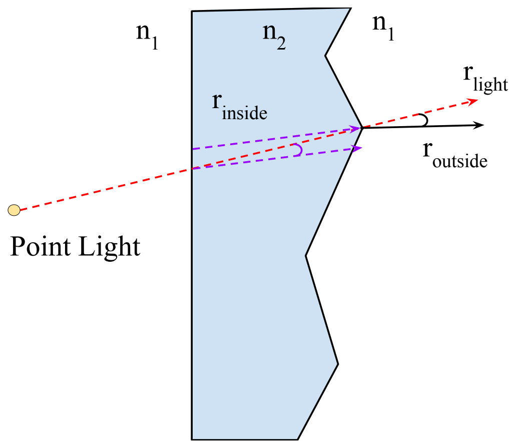

Figure 5: *Representation of the details of the refraction computations with point light*

inside-ray, we would need to compute the intersection point of the light ray on the flat surface. As we will see section 3.3, this was not feasible at the moment, and since the distance from the mesh to the light is very large compared to the thickness of the refractive material, and even more large compared to the size of one triangle, we consider this error acceptable.
When Rotating the mesh, we also apply the adapted transformation to the ”object normal”, representing the normal of the flat side used in the first refraction computation. When the object normal points toward the screen then the refraction algorithm takes this fact into account by first refracting on the caustic surface and then on the flat surface. Fig. 6 shows the impact of rotating the mesh on the caustic pattern simulated.

2.1.3 Reflection Algorithm

The ray-tracing algorithm is easily modified to obtain the reflection pattern of a reflective mesh. The refraction computation at each vertex is simply replaced by a simple reflection of the light vector using the vertex normal.

2.1.4 Post Processing

The obtained image shows the result of the caustic pattern, but does not really show the contrast seen in real life. In particular, the Ray-

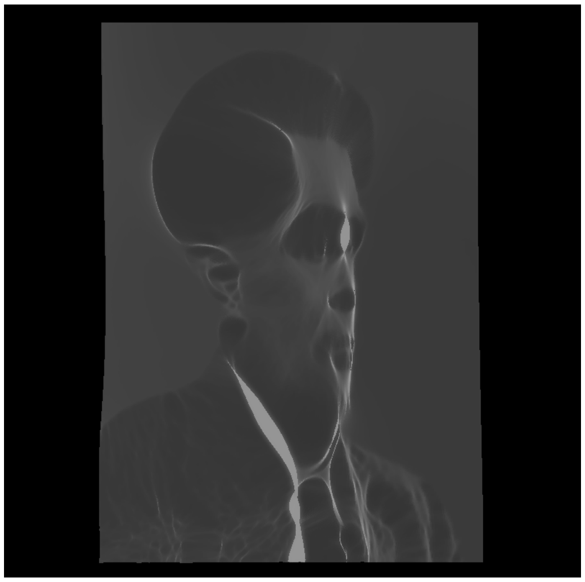

Figure 6: *Caustics with post processing, when the mesh is rotated by $\pi/16.0$. Light mode = directional.*

Tracer projects triangles to the screen with an intensity given by the aforementioned formula, and an alpha designating the transparency. Then, when two projected triangles overlap, the local value of the FrameBuffer can exceed 1. When using default FrameBuffers, values exceeding 1 are clamped to 1 before being stored, leading to loss of information in the high luminance regions.
This is a standard problem when rendering light patterns and can be partially fixed by artificially manipulating the contrast and adding some effects. A natural first step to answer this issue is to store the values of the RayTracer-computed caustic pattern in a floating point FrameBuffer. These non-clamped value, called High Dynamic Range values, are used in post-processing algorithms. Fig. 8 shows the comparison with and without processing.
Fig. 7 shows the post-processing pipeline implemented. A first FrameBuffer is created to store the original HDR image rendered by our RayTracer. This FrameBuffer is then passed to the post-processing class. The main challenge of implementing the presented post-processing pipeline has been to efficiently manage all these FrameBuffers which all require a different set of shaders, some even requiring two passes. In order to optimise performance, all these shaders and their corresponding shader programs are instantiated and used in a single class called PostProcessing.
At each rendering loop, this class binds a High Clamp F.B. and pass it the HDR image. A first shader program is used to clamp all values below 1, keeping therefore only the high luminance regions.
This High Clamp F.B. is passed by the same class to the Blur shader program, that applies a two-pass gaussian filter and stores the result in the Blurred F.B. The last shader program, called LDR, takes in the blurred high-clamped values as well as the original HDR image. It superposes the blurred image to the HDR one to give the impression of ”blooming” from high-values regions, and then apply a standard gamma-exposure HDR algorithm to convert the resulting superposition into a LDR image that can be displayed by OpenGL without loss of information. Fig 9 shows the result of adding blooming to the HDR image before applying the HDR function that converts it to a LDR image.
As an additional optimisation step, using FrameBuffers for the post-processing makes it possible to render and compute the caustic pattern only when necessary. In particular, we check at each frame if some parameter has changed, and only render if it is the case. If no

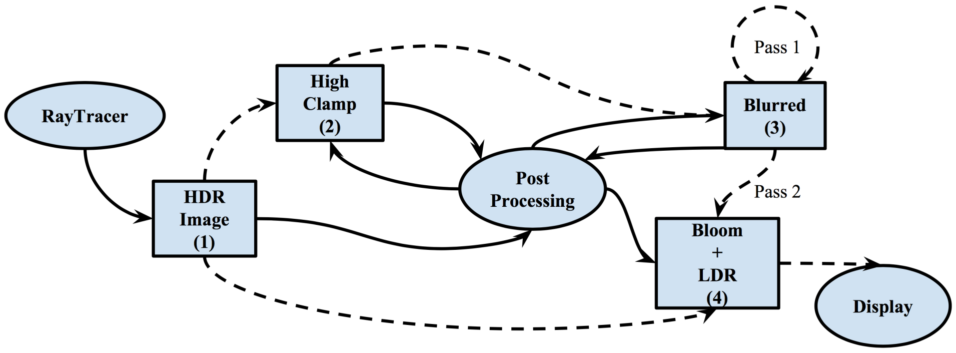

Figure 7: *Post Processing Pipeline*

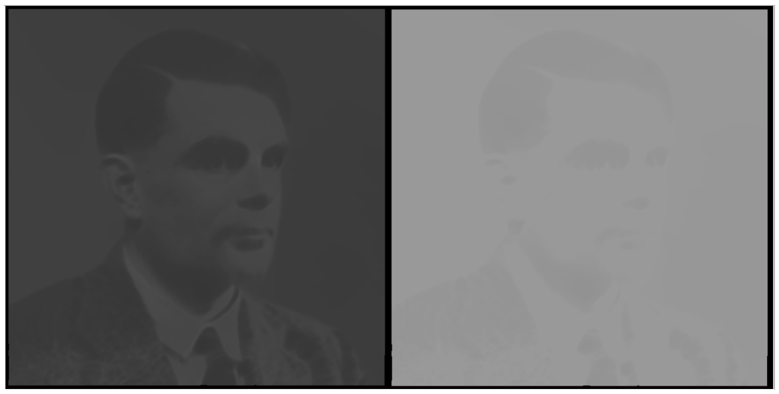

Figure 8: *Caustic comparison with (left) and without (right) post-processing. Here,* $\alpha = 0.6$, $Exposure = 5.2$, $\gamma = 0.05$

parameters have changed, we just display the FrameBuffer containing the result of the previous rendering + post-processing.

2.1.5 Mesh Pre-Processing

A few steps have been added after the loading of the mesh, such that the server rendering setup is compatible with the setup displayed on the client side.
First, we assume that all the meshes are more or less of a rectangular shape. Some of the input mesh are centred on zero, some have vertex position stating at zero... So the first preprocessing step added is to compute position of the four "corners" of the mesh, and deduce the "center" of the mesh. We then add to the MVP matrix computation an initial translation in order to center this mesh on zero.
As these rectangles don't all have the same area, we also use these results to compute its area and rescale the mesh to have it of size approximately 100x100.
As each mesh has its own focus distance (i.e. the distance from the screen such that the caustic pattern is the exact desired image), when loading a mesh we also load a set of corresponding parameters, including this distance, that we multiply by the scale found in the previous pre-computing step.
As a result, when loading a mesh, the program also automatically scales it, center it, and translate it to its focus position relative to the screen.

2.2 Client Side

The first approach to render the environment on the client side was to hardcode it using WebGL. But it quickly became obvious that using javascript to write OOP rendering code was complicated and unstable. After 2 weeks of trying to implement my own WebGL API, it seemed that it would be faster to use an existing API even if that meant learning yet another tool. The client side rendering has therefore been implemented using the ThreeJS API.
The first innovative feature implemented was to modify the ThreeJS control class to allow the program to manage the controls of the scene objects in a state machine. This way, the user can only access to one set of parameters at a time, and the rendering program is always in a pre-defined controlled state: No controls, Light translation, Object translation, or Object rotation. The control class has also been modified to offer setter functions to display or not, the control helpers, in function of the said control activation state. Fig. 10 shows the result on the client side. The rendering loop then detects if some parameter has changed and if that's the case, send to the GPU server the new set of parameters.
The next challenge was using the incoming texture as an additive lightmap, i.e. as a light map that is superposed to the scene lightning done by ThreeJS. To do so, the solution found was to add a virtual plane just in front of the rendered box representing the screen, and tweak its parameters such that it acts like a transparent texture, superposing itself with additive blending. The only constraint of this approach seems to be that the resulting displayed caustic patterns does not appear in the refractive texture of the box representing the caustic mesh (as seen in Fig. 1).
Regarding the hosting of the client side, I used my DigitalOcean

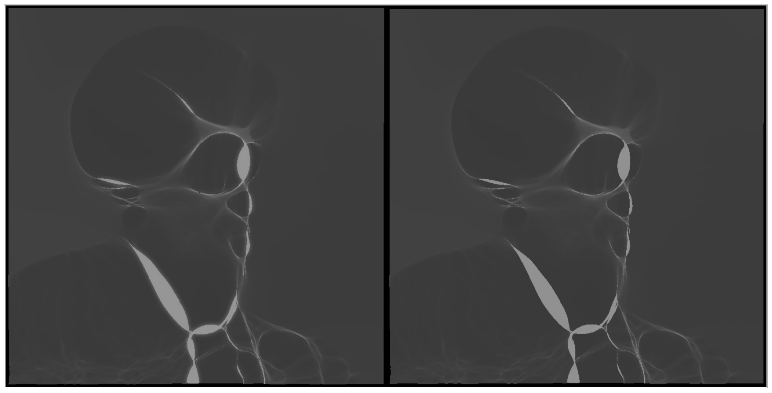

Figure 9: *Caustics comparison with (left) and without (right) bloom effect. The mesh is purposely off-focus to display the triangles overlap.*

credits, generously proposed by the Github Student Pack. It runs on Ubuntu 16.04 and the virtual host is setup using nginx.

2.3 Linking the two sides

As of this day, this part is not functional yet. Both sides are prepared to be linked, from parameter change detection in the client side to parameters update function in the GPU server side. But the last features still remain to be implemented, and more problematically, some are barely documented. More precisely:

1. How to wrap the OpenGL FrameBuffer normally destined to the screen in a video stream
2. How to get this said stream accessible from a public link (that will be used as texture source in the client side)
3. How to launch an AWS instance from the client side
4. How to make the OpenGL code listen to some parameter changes that will be thrown by the client side. (This step seems to be the most problematic)

Once this is done, the whole WebGallery will be up and running.

3 Future Work

3.1 WebGallery Homepage

A simple feature that could be implemented would be a homepage for the webgallery, proposing the different meshes that are implemented in the openGL side. When launching the rendering with a particular mesh, the scene will come loaded with appropriates parameters.

3.2 Client Side aesthetics

There is no wondering if the client side could be made more aesthetically pleasant. This could be achieved using physical lightning, ambiant occlusion, some GUI menus, a better looking environment etc ... And, as mentioned before, fixing the displayed object refraction.

3.3 Physically correct refraction with Pointlight

To resume the problem, when refracting the ray in the object with point light, we use as r_{inside} the result of refracting the vector $v_{pos} - light_{pos}$ on the flat surface, but this ray does not intersect the second face of the object exactly in v_{pos}. One could therefore look to compute the exact r_{inside}. This would mean looking for $intersect_{pos}$ such that the result of refracting the light ray $intesect_{pos} - light_{pos}$ on the flat surface, give the vector $v_{pos} - intesect_{pos}$.
The refracted out-going ray is computed as follow:

$$k = 1 - eta^2 \left[1 - (\boldsymbol{n} \cdot \boldsymbol{I})^2\right] \tag{2}$$

$$\boldsymbol{r_{out}} = \begin{cases} \boldsymbol{0} & \text{if } k < 0 \\ \boldsymbol{I} * eta - \boldsymbol{n}\left[eta(\boldsymbol{n} \cdot \boldsymbol{I}) + \sqrt{k}\right] & \text{otherwise} \end{cases} \tag{3}$$

Where $\boldsymbol{I}$ is the incidence vector, $\boldsymbol{n}$ is the normal of the surface, $eta = n_1/n_2$, and $\boldsymbol{r_{out}}$ is the resulting refracted ray.
Noting $\boldsymbol{l} = light_{pos}$, $\boldsymbol{p} = intersection_{pos}$, $\boldsymbol{v} = v_{pos}$, we can use:

- $\boldsymbol{I} = \boldsymbol{p} - \boldsymbol{l}$
- $\boldsymbol{r_{out}} = \boldsymbol{v} - \boldsymbol{p}$
- $\boldsymbol{n} \cdot \boldsymbol{I} = \boldsymbol{n} \cdot (\boldsymbol{p} - \boldsymbol{l}) = \boldsymbol{n} \cdot \boldsymbol{p} - \boldsymbol{n} \cdot \boldsymbol{l}$

And we get the equation system:

$$\begin{cases} k = 1 - eta^2 \left[1 - (\boldsymbol{n} \cdot \boldsymbol{l})^2 - (\boldsymbol{n} \cdot \boldsymbol{p})^2 + 2(\boldsymbol{n} \cdot \boldsymbol{l})(\boldsymbol{n} \cdot \boldsymbol{p})\right] \\ \\ \boldsymbol{n}\left[eta(\boldsymbol{n} \cdot \boldsymbol{p}) + \sqrt{k}\right] - \boldsymbol{p}(1 + eta) = eta\left[\boldsymbol{n}(\boldsymbol{n} \cdot \boldsymbol{l}) - \boldsymbol{l}\right] - \boldsymbol{v} \end{cases} \tag{4}$$

Under condition that $\boldsymbol{p} \in object\_plane$
Since we have all parameters in this equation except $\boldsymbol{p} \in \mathbb{R}^3$, this computation seems expensive and hard to implement. Nevertheless, it could be possible to find an optimized and/or approximation of this computation. If possible, we would get a function that, given a light origin, a plane and a destination, find the path of the light ray from the source to the destination while taking into account the unknown refraction happening at the plane.
Another step toward making the refraction more physically realistic would be to use the Fresnel term to blend reflection and refraction when the object rotates.

3.4 Synchronous implementation

As of the writing of this report, the asynchronous implementation of the parallel algorithm is in progress, as it seems easier and more efficient. But implementing a synchronous pipeline would provide a more realistic feeling since applying a transformation to some object of the scene would impact instantly the caustic pattern display.

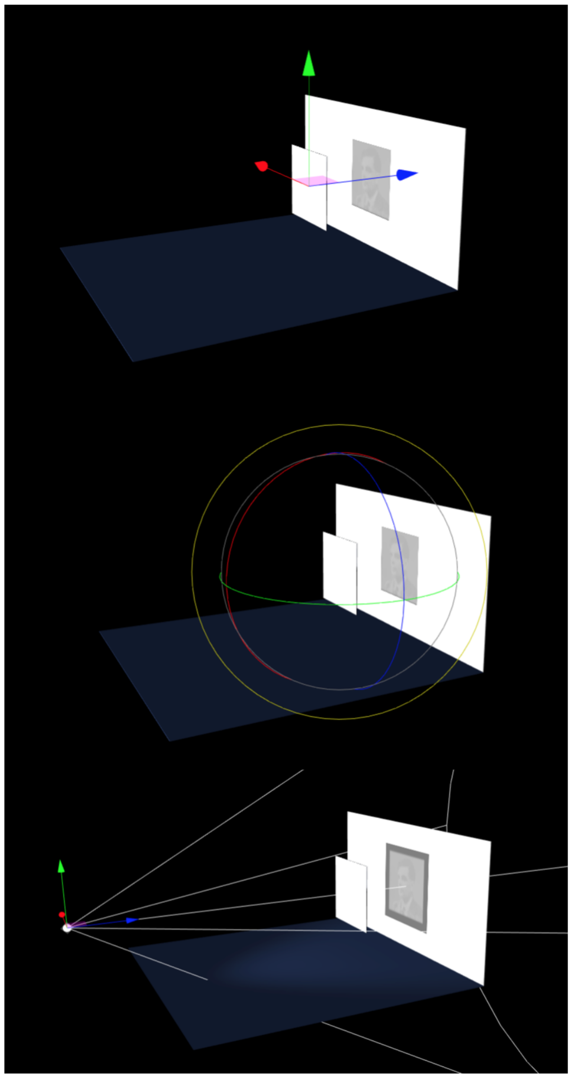
Figure 10: *The three control modes of the client side. Top image: Object translation, Middle image: Object rotation, Bottom image: light controls*

3.5 Genericity

The ray-tracing program could be made even more generic. First, some computations could be modified such that we can give it any mesh, screen parameters, and light position and automatically use those parameters to initialize the scene at optimized position, as well as taking into account the screen orientation. As shown in this report, a great amount of work has already been done in this direction, but debugging passes forced to hard-code some of the values to ensure stability.
The algorithm could also be adapted to project the refracted rays onto any kind of surface: a room populated with objects, a sphere, a bottle of perfume etc ...

3.6 Multipass refraction and colors

By the discretisation of the refraction algorithm (Fig. 3), multipass refraction is not possible. As before, when refracting the ray a first time, it is very unlikely that the resulting refracted ray intersect the following caustic surface in a vertex, and therefore the current algorithm cannot compute the second refraction. It is possible nonetheless, that by solving the problem exposed in section 3.3, we could as well find a way to implement multipass refraction, i.e. simulate the caustic pattern done by multiple caustic surface.
Some of the caustic objects developed by RayForm happen to also project colors in their caustic pattern. If the input 3D objects contain information about some coloration of the mesh triangles, it could be possible to adapt the algorithm to take these colors into account when rendering the caustic pattern.

3.7 Local HDR

For now, the HDR algorithm implemented is a classic gamma-exposure global HDR function. A local histogram-based HDR could be implemented instead in order to get an even more realistic feeling.

4 Conclusion

Working on the WebGallery has been extremely exciting and interesting. Beginning a project of this scale with nothing provided but the meshes and some global objectives, while being very challenging, has allowed me to implement features on my own and in my own style. Some of the presented algorithm have taken days of pen-and-paper designs and tests, and a lot of trials and errors. Unfortunately, it also meant spending an immense amount of time debugging, and trying out implementations that never worked, and so were deleted. As of this day, the client side and the OpenGL part are functional and almost ready to link, but I miss just a few week to make it work as it is supposed to do. Since the presentation will take place a week after the due date of this report, I hope I will be able to show a functioning and linked WebGallery, and if it is not the case, I will continue to work on this project at least until both sides are linked.
Nevertheless, if all the people that allowed me to work on this project agree, I would love to continue working on this as an Optional Project for the Spring 2017 semester.

Acknowledgements

Thanks to Niranjan Thanikachalam who supervised me during this project, helped me design some of the algorithm and was the intermediate between the school, Rayform, and me.
I also would like to thank Dr. Pauly, who created this project for me, and to Yuliy Schwartzburg, CTO of rayform, who provided me some guideline as well as allowing me to work in pair with Rayform.

References

WALLACE, E., 2016. Rendering realtime caustics in webgl. https://goo.gl/yfM8OF.