A Simple Demonstrator Of How The Ray-Tracing Algorithm Functions In WebGL

Gabriel Silva (85129) & Gonçalo Vítor (85119)

***Resumo* – O artigo aqui escrito descreve o trabalho feito pelos autores no desenvolvimento duma aplicação simples, usando WebGL, no âmbito da unidade curricular de Computação Visual, que demostra o funcionamento do algoritmo de Ray-Tracing usado para a criação de imagens com efeitos de iluminação realistas.**

O artigo começa por descrever abreviadamente as ideias por trás do algoritmo Ray-Tracing, seguindo-se uma apresentação da aplicação desenvolvida e explicação de como foram desenvolvidos os pontos mais cruciais da mesma.

***Abstract* – The hereby article depicts the work done by its authors in the development of a simple application, using WebGL, for the curricular unit Computação Visual (Visual Computation), which demonstrates how the Ray-Tracing algorithm works to create images with realistic illumination effects.**

The article starts with a description of the ideas behind the Ray-Tracing algorithm, followed by a presentation of the application created and an explanation of how the more critical aspects where developed.

I. INTRODUCING RAY-TRACING

Images are essentially composed of two things : a geometry (one/several shape/s) and colour. Colour is no more than light beams being absorbed, reflected or transmitted. The colour of an object depends on how much light it absorbs, reflects and transmits. In nature light comes from sources, the most common being the sun, that shoot enormous amounts of light beams in all directions. Those beams may later hit an object that reflects that light into our eyes, making us see the object.

The goal of Ray-Tracing is to give us a way to simulate the reality of how light interacts with objects in a computer environment. The first big problem of it is, obviously, the huge amount of rays produced by light sources, that would make any computation of the sort practically impossible. So, instead of calculating every ray a light source produces and checking which ones hit an object and then a human eye or camera, Ray-Tracing does this in the inverse order. The only points we're interested on are the ones we can see. All the other light beams that don't turn into something we can see don't matter to us. The place to start at, then, is the eye/camera. It's important to have a projected scene in mind between our eye and our image. This scene has a height h and width w, giving it h*w pixels. Ray-Tracing starts by, from the eye, creating rays that hit every pixel of our scene. Our interest is in the rays that both hit the scene and an object of our image. In this case a second ray is cast, from the point were the object was struct to the light source. The first rays mentioned are called primary rays, and the second ones secondary rays.

Primary rays tell us what points should be coloured. Secondary rays are needed to discover what light should be applied to the hit point in the object, this is, what colour it should have. This depends on the surface of the object (if we imagine a sphere with a light right on top of it, its colour isn't uniform, the bottom should be darker, and even in the top side the centre would be brighter than the "sides") and, also, on the existence of other objects that may be in the way blocking the light, causing a shadow.

II. THE APPLICATION

The main aspect of Ray-Tracing is, like the name indicates, the rays created to illuminate an image. This is what the application focuses on. A simple image is presented to the user, where he/she can see 2 spheres over a floor. In front of all this there's a cone, representing a person, where the tip of the cone is the eye, and between

the person and the image, despite not visually represented, is the scene mentioned in the introduction. There are also two different light sources (in the images bellow, the light sources are the two spheres closer to each lateral boundary), that the user can "play" with: the colour of both lights can be changed, as well as their position on the image. The only thing left is to animate everything. Animating starts to create the rays and displays them on screen. Both primary and secondary rays are shown, and the spheres' and floor's colours change as a result of the computed rays. It's important to note, like we'll mentioned below, that since shadows were not taking into account, the floor, for example, having the spheres above it, and the light above the spheres, does not have shadows depicted in itself. It's also possible to reset everything to the initial state to test the result of the animation with different light colours and positions.

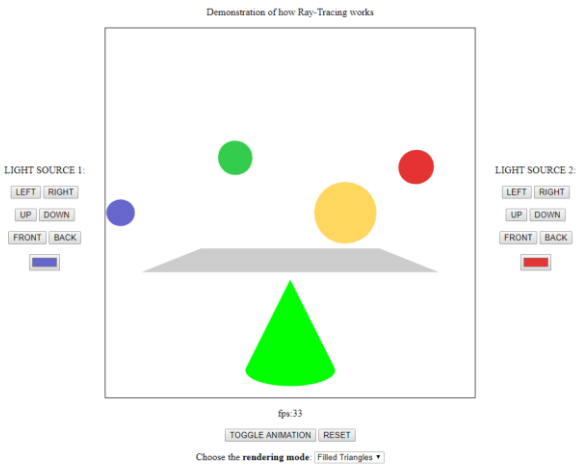

Figure 1: Initial state

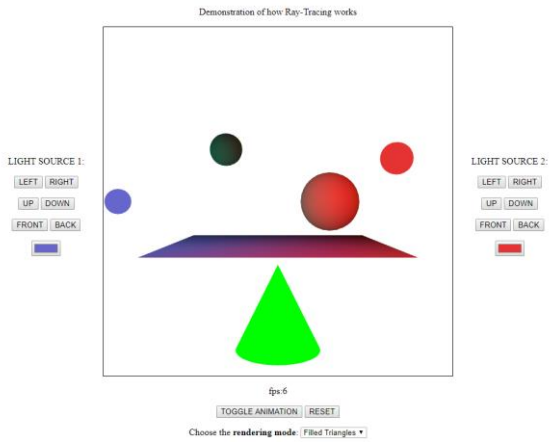

Figure 2: Result of the animation

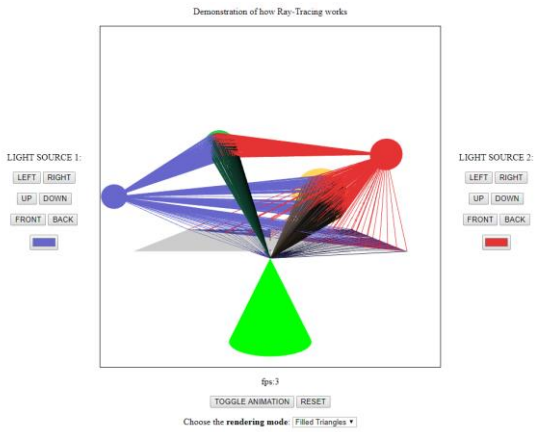

Figure 3: The rays created

III. BUILDING THE APPLICATION

To build the application, a couple things were simplified given that first: some aspects of Ray-Tracing require large computational efforts; and secondly: we're only trying to illustrate how the algorithm works, and not trying to implement it ourselves. The first thing simplified is the type of light taken into consideration to colour objects. Reflected and transmitted light was ignored, both to simplify the problem and make it computationally less expensive. Calculating shadows was also not dealt with, since it demands that for any ray of light we check if it intersects any point of all the other objects on its way to the light source, which asks for a large effort.

The code produced was built on top of what was provided in the practical classes, although a lot things were changed and added.

The draw in the canvas is made using a perspective projection, so we get a more realistic view of the image.

To facilitate the calculation of the rays a couple things were changed. When creating a ray, we link, for example, any point of one of the spheres to the cone's tip. That is very simple assuming we have the real coordinates of every point. However, the way we learned in classes, objects were created given relative positions, and after that, with the wanted translation and scaling factors, translation and scaling matrices were created and applied to our shaders. The problem with doing it this way is that we never have access to the real coordinates of any object.

To solve this, we decided to apply the translations and scaling factors to the objects as they are created, instead of when they are drawn.

We already know how to calculate rays, but how did we draw them? We are constantly drawing triangles, but when we do in fact compute rays of light, we keep a count of how many were created, and we add them to the model's vertices and colours in question (we calculate rays for each model separately). This way we can, then, draw everything as triangles minus the amount of rays calculated and proceed to draw the rest as lines.

Now we can both calculate and draw the rays. The only left is to give the rays an appropriate colour. Again, as mentioned before, because shadows were neglected, secondary rays will never be shadow rays, meaning they always have the same colour as the light source itself (this is easily visualize in **Figure 3**), making it very simple to give colour to these rays.

The primary rays, however, aren't so simple. Because there's no shadows, the colour of the primary ray is the colour of the object that light is hitting. So to calculate the colour we start by iterating over the objects points and calculate the Euclidean distances between that point and the light source centre point. We do this to keep the minimum and maximum distances. We are going to use them to build a "illumination coefficient". How? Simple: the difference between the maximum and minimum distance is the amount of divisions on our coefficient scale. Imagine max – min = 10. This means we need to have 10 steps from 0 to 1 (the minimum and maximum values of the coefficient). The important thing to know now is that the closer a point is to a light source the more colourful it is. This means that when the object is at the minimum distance the value of the coefficient should be 1. The inverse happens to the maximum distance (we must get the value 0). Continuing the example: now lets imagine our point is the at the minimum distance (we must get the value 1). In this case, maximum – ourPointDistance = 10. From here we can easily understand that our coefficient is calculated using the following expression :

(Max – Dist) * (1/divs),

Where Max is the maximum distance, Dist is our points distance to the light source, and divs is the amount of divisions in our scale, given by the difference of the maximum distance with the minimum distance.

Now that we have our coefficient, for every point, we simple multiply the objects original colour with our lights colour and multiply that with the coefficient calculated for that same point (this multiplications are made for the red, green and blue properties of the colour).

The last thing we did was adding the second source of light. By doing so, we made it so that each object is hit by two lights, creating another problem: how do you calculate light now? We found a simple way solve this. Let's imagine a sphere that is purely white. Whatever light hits it, the colour you see in the sphere is the colour of that light. Now let's put two lights on the scene, one is bright green (meaning the green aspect of this light's colour is high) and the other is dark green (meaning the green aspect of this light's colour is low). How do we see the sphere? In dark green? No. We'll see the sphere as green as the bright light's green was. So the way we select the colour to give each point's red, green and blue property is by choosing whichever colour calculation from both lights gave the highest value.

VI. CONCLUSIONS

Even tough we were not supposed to build the Ray-Tracing algorithm, we still would've liked to show how it works using a more complete demonstration, by using at least shadows. That did prove both hard and computationally expensive. Still, we feel like we built an interesting application, were we both illuminate a scene from scratch by our own, and at the same, demonstrate a bit of how Ray-Tracing works.

Both members of the group contributed equally to development of the project.

REFERENCES

https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing , accessed throughtout the month of november, 2019;