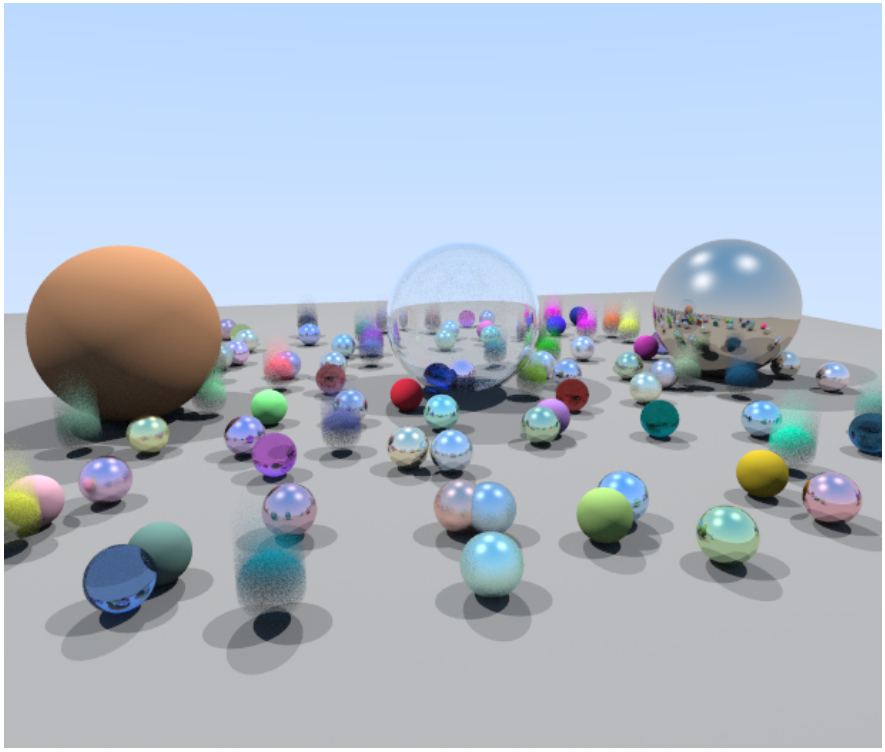

GLSL Ray Tracer

P3D 2nd Assignment - May 2021

Daniel Gonçalves [98845]

Diogo Silva [87652]

Henrique Gaspar [98879]

Table of Contents

1 Main

For this assignment the team created a **Progressive RayTracer** using GLSL. Rather than taking multiple samples per pixel in each frame, we create a feedback loop using a fragment shader and feed all previous results to the current frame. For each frame we also use a random offset for the current pixel (which inherently creates an Anti Aliasing effect). The number of samples to take per pixel is also capped at 10000 in order to avoid floating point issues (i.e the running average growing too large). As such, the main function is the starting point of the program: It sets the camera and all its variables needed for itself but also for other techniques that will be applied in the future (Motion Blur, for example), deals with the movement of the mouse to update the position of the camera, asks for the color of a given pixel while also performing gamma correction. The way we store all previous frames results is by saving it as a texture in *channel0* and feeding it as input to the current frame. Additionally we use the 4th channel of the aforementioned texture to store the frame counter in order to reset it when the scene is altered (i.e, when we move the mouse), so that we can discard past frames and start from scratch. Finally, our scene is represented by “being built on each invocation of the hit world function”, i.e, it's not actually stored in memory.

At the top of the *P3D_RT.glsl* file there are some control variables which can be set in order to enable or disable certain features:

- **USE_RUSSIAN_ROULETTE** - Enables/Disables Russian Roulette Optimization
- **ORBIT_CAMERA** - Enables/Disables Orbit Camera
- **SHOWCASE_DOF** - Swaps the aperture and focal plane distance’s value to better showcase DOF
- **SHOWCASE_FUZZYREFL** - Swaps the Reflection Roughness value of the right sphere to better showcase fuzzy refractions
- **SHOWCASE_FUZZYREFR** - Swaps the Refraction Roughness value of the middle sphere to better showcase fuzzy refractions
- **NO_NEGATIVE_SPHERE** - Removes the inner “negative radius” sphere

2 Ray Colour

2.1 Functioning

The ray color function is responsible for dealing and assigning a color for each pixel, taking into consideration refractions, reflexions, light sources and even the background that is supposed to be shown when a given ray does not hit any object. Due to lack of recursion, the way RayColour works is using a for cycle (iteratively rather than recursively). We start the pixel color at black (0,0,0) and a throughput color at white. We shoot the ray we got from our camera (which computes the primary ray direction taking into account the aperture of a lens and the focal plane in order to create the Depth of Field effect) and, when that ray hits an object we add it's color to our throughput (direct lighting). Afterwards we get a secondary ray which corresponds to the current ray's bounce off the hit object, swap our current ray with that one before the next cycle, and update the throughput with the attenuation (which most of the times corresponds to the object's albedo) received from the scatter function.

2.2 Direct Lighting

The direct lighting method (which is currently called 3 times, due to having 3 point lights in the scene), first checks whether the intersection computed in the hit_world before is in shadow or not (by calling hit world again but with a ray going from the intersection point to the light, a shadow feeler). If not in shadow we sum to our current colour variable the colour computed using the Blinn-Phong model (noting that we hard set values for the hit object's specular, diffuse, shininess and respective specular and diffuse color depending on the type of object).

2.3 Scatter

Our scatter function is used to get the ray resulting from our current ray bouncing off the material. This function works by checking the type of material the hit object is: For **diffuse** (lambertian) objects, we achieve color bleeding by randomly "*bouncing in a cosine weighted hemisphere direction of the surface normal*" - since in these materials light sort of bounces in all directions, what we do is generate a unit sphere on top of the hit point, and pick a point in its surface, done by adding the hit normal to the hit point (to get the sphere center) to a normalized random unity sphere, returning the scatter direction as a ray going from the intersection point to this computed one. For **metallic** objects we simply compute the reflected ray (as seen in mirror reflections) and perform fuzzy reflection by deviating it in a random direction using a sphere of radius equal to the reflection roughness (further explained in section 4.1). For **dielectric** materials, we choose whether to return a reflection or refraction ray by computing Schlick's approximation of the Fresnel Equation to get the reflection probability, and then use it to randomly pick between doing reflection or refraction. It should be noted that this function also returns the attenuation value to which we will multiply our current throughput with (i.e the impact that this object's albedo will have on our ray's current color).

3 Intersections

3.1 Sphere

For the sphere intersection, we decided to use the same method we used for the last assignment, by using implicit geometry to represent both concepts (sphere and ray) by their mathematical equations. Then, by mixing them together until we find a single equation in order to t (the multiplier of the ray's direction), we end up with a quadratic equation that, by the use of its determinant, can help us distinguish t0 and t1 (that represent two points of intersection with the sphere) and understand which one represents the "entry" point an the "exit" point. Important to say that this solution also helps us to know if the ray actually starts inside the sphere.

3.2 Moving Sphere

The moving sphere intersection works very much like the sphere one, with the difference being that we have to compute the sphere's center by taking into account the sphere's center at t0, the sphere's center at t1, and the time at which the ray was sent.

3.3 Triangle

For the triangle intersection, and for the sake of efficiency, we used the **Tomas Moller** technique and defined our triangles by barycentric coordinates when testing a possible intersection. This approach is far better than the traditional geometric one because it only needs to store the vertices of each triangle, and not the whole pnae where it fits. As for the first assignment, since the explanation of this technique is pretty extensive, we will go to details. Insted, we advise consulting the original document that we used when implementing such technique: https://cadxfem.org/inf/Fast%20MinimumStorage%20RayTriangle%20Intersection.pdf

4 Extras

4.1 Fuzzy Reflections & Fuzzy Refractions

Although Fuzzy Reflections weren't an extra we chose to include them in this section since both fuzzy methods were implemented in very similar fashions. To simulate the surface of our refractive materials having roughness we implemented Fuzzy Refractions. We started by adding a new ***roughnessRefr*** parameter to our materials. Then we accomplish the fuzziness (both for reflection and refraction) by deviating our scatter ray by a random direction within a unit sphere with radius equal to the corresponding roughness parameters. As such, this deviation is done after computing the reflection or refraction direction.

4.2 Camera Zoom + Orbital Camera

For our **zoom** we created a uniform float value (meaning that this value persists through each call of the shader) to replace our static ***fovy***. We did this because, by looking at the Shadertoy VS Code plugin, we discovered it allows for the inclusion of a slider that allows us to play with the uniform's values during execution. The downside to this is that we have no way of announcing to our fragment shader that we should reset the sample counter and reset the scene.

For the **orbital camera**, all we did was define a minimum and max angle, a sensitivity (i.e how much our mouse movement impacts the camera movement) and a camera distance variable (how far away from the target the camera is). We can get our x angle by multiplying our mouse's x coordinate by this sensitivity and, as for our y angle we can get it by performing a linear interpolation between the minimum and maximum angle and the mouse's y coordinate (making it so the camera is clamped between looking straight up and straight down). With these 2 angles we can then compute the camera's eye position.

4.3 Russian Roulette

By default each ray will bounce up to a max of 10 times (or until it hits nothing) before "disappearing" but after a few bounces the color increment may start becoming negligible enough that the ray should be able to terminate early without us losing quality. By enabling the ***USE_RUSSIAN_ROULETTE*** variable, after getting the new scatter ray (in the RayColor function) we check what the maximum current throughput color channel is and use it as the percentage of our ray NOT terminating early - the closer the 1, the lower the probability of russian roulette terminating our bounces. To further compensate for the fact that we're terminating our raytracing function early, we divide our current throughput by the maximum current throughput color channel (which is between 0 and 1), meaning each individual ray will be brighter, to compensate for the fact that some

rays may end up being killed to soon. With this performance optimization we're able to more than double our FPS (from ~20 to around ~50) without losing any noticeable image quality.