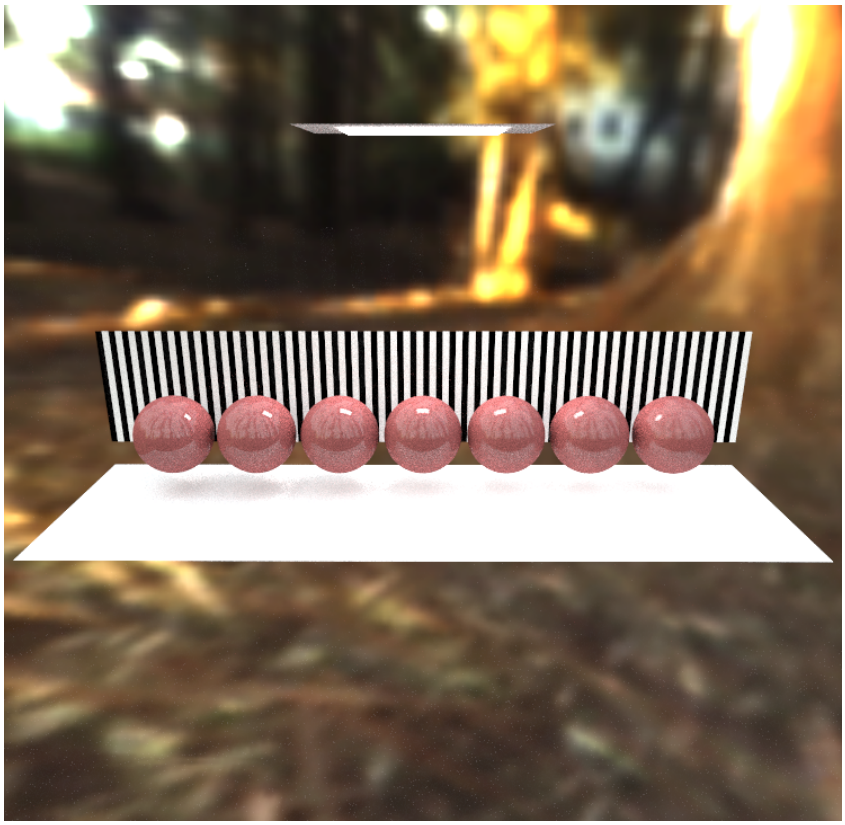
TÉCNICO LISBOA

3D Programming

Report: Assignment 2

GPU Pathtracer



Group 2:
Beatriz Neto — 103149
Catarina Costa — 112377
Rita Mota — 103509

2024/2025 – 4th Quarter

1 Introduction

2 GPU Ray Tracer

2.1 `common.glsl`

2.1.1 `getRay()`

The `getRay(Camera cam, vec2 pixel_sample)` method builds on the approach from Assignment 1, where rays were generated from the eye through a pixel on the image plane. However, in this version for the GPU Ray Tracer, the ray origin is offset using a random lens sample to simulate depth of field, and the direction is adjusted accordingly using the camera's basis vectors. Additionally, a random time value is assigned to each ray to enable motion blur. Unlike the earlier CPU-based version, this implementation integrates both effects directly and is optimized for parallel execution on the GPU.

2.1.2 `schlick()`

This method was implemented similarly to Assignment 1, taking as input the indices of refraction for the current and subsequent media, along with the cosine of the incident angle. It uses Schlick's approximation, shown in Expression 2, to compute the Fresnel reflectance.

$$K_r = R(\theta_i) = R(0) + (1 - R(0))(1 - \cos\theta_i)^5 \tag{1}$$

$$R(0) = \left(\frac{\eta_i - \eta_t}{\eta_i + \eta_t}\right)^2 \quad \text{(Fresnel reflectance at } 0^\circ\text{)} \tag{2}$$

2.1.3 `scatter()`

This function determines how an incoming ray generates secondary rays upon interacting with a surface, depending on the material type.

For diffuse materials, the ray is scattered in a random direction following Lambertian reflection. Color bleeding is handled by attenuating the ray based on the surface's albedo and the cosine of the angle between the surface normal and the scattered ray direction, as shown in Expression 3.

$$\text{Attenuation} = \text{albedo} \cdot (n \cdot \mathbf{d}_{\text{scattered}}) \tag{3}$$

For metallic materials, the ray is reflected with some fuzziness proportional to the surface roughness, as in Assignment 1. The direction is calculated using the GLSL `reflect()` function, which computes the reflection of the incident direction about the surface normal. The ray's attenuation is determined by the material's specular color.

For dielectric materials, the ray can either reflect or refract based on Schlick's approximation of Fresnel reflectance. A random value is compared against the reflectance coefficient to decide between reflection or refraction. Directions are computed using the `reflect()` and `refract()` functions—the latter requires the incident direction, surface normal, and the ratio of indices of refraction. When a ray refracts inside a medium, attenuation is computed using Snell's Law.

This method returns the attenuation vector and the scattered ray.

2.1.4 `hit_triangle()` and `hit_sphere()`

Both methods were implemented in the same way as in Assignment 1: triangle intersections were handled using barycentric coordinates, while sphere intersections were optimized by analyzing the discriminant of the quadratic equation.

After presenting the assignment to the professor, an error was identified: the implementation did not account for spheres with negative radius values. This oversight caused the refraction of rays inside such spheres to behave incorrectly, resulting in unrealistic rendering of internal refraction effects.

2.1.5 `hit_movingSphere()`

This method extends the static sphere intersection logic to account for motion over time, enabling the simulation of motion blur. Unlike a regular sphere whose center is fixed, a moving sphere interpolates its center between two positions (`center0` and `center1`) based on the ray's time parameter. This makes the center of the sphere time-dependent, introducing variability in its position as rays are cast at different times.

The intersection test still solves a quadratic equation, but the sphere center used in the calculation is adjusted to reflect the sphere's location at the ray's time. This allows the method to determine whether a ray intersects the sphere not just in space, but also at the correct point in time.

The surface normal is then computed at the point of intersection using the sphere's interpolated center. If the ray hits from inside the sphere, the normal is flipped to ensure proper orientation.

2.2 `P3D_RT.glsl`

2.2.1 `directlighting()`

This function calculates the direct illumination at a surface point from a point light source, considering basic diffuse and specular reflection and hard shadows.

The direction and distance to the light source are computed first. To simulate **hard shadows**, a shadow ray is cast from the surface point toward the light. If this ray intersects any object before reaching the light, the point is considered in shadow, and the function returns no light contribution.

If the point is lit, two types of light contributions are computed:

- **Diffuse reflection** is based on Lambert's law, using the dot product between the surface's normal and light direction, scaled by the material's albedo.
- **Specular reflection** is calculated using the Blinn-Phong model. The half-vector between the light and view directions is used along with a roughness-dependent shininess value to simulate highlights.

The final color output is the sum of the diffuse and specular terms, scaled by the light's intensity and the surface-light angle.

2.2.2 `rayColor()`

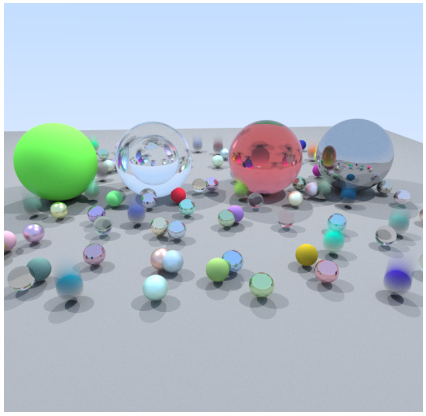
The `rayColor()` function is the core of the ray tracing algorithm. It computes the final color seen along a ray by simulating light transport through multiple surface interactions (bounces). At each bounce, the following steps are performed:

- Ray-Surface Intersection: the ray is tested against the scene geometry; if it hits an object, shading is computed at the intersection point.
- Emission Contribution: if the material at the hit point is emissive, its emission is added to the output color, scaled by the current `throughput` (which tracks the accumulated attenuation across bounces).
- Scattering : a secondary ray is computed on the basis of the scattering behavior of the material. The `throughput` vector tracks how much light is retained as the ray bounces. It begins as white (`vec3(1.0)`) and is updated at each bounce by multiplying it with the surface's attenuation (`atten`), accounting for energy loss due to reflection, absorption, or transmission. If there is no scattering, the path terminates.
- Russian Roulette Termination: to avoid wasting computation on rays that contribute little to the final image (low `throughput`), we use Russian Roulette: a probabilistic method to terminate weak rays. The probability of continuing is proportional to the remaining energy (`max(throughput.r, g, b)`); if a ray survives, the `throughput` is scaled by the inverse of the survival probability to compensate for terminated paths, ensuring energy conservation and unbiased results.

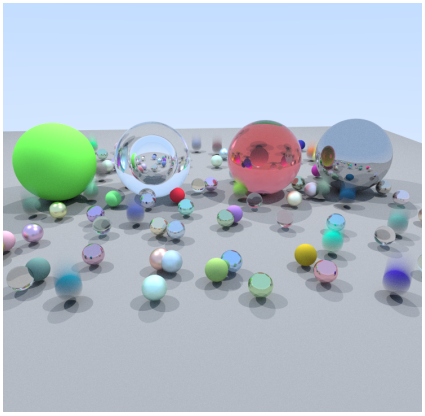
- Background Color: if the ray does not hit any object, a background color is returned. This may be a gradient or an environment map, depending on the setup of the scene.

The function loops up to a maximum number of bounces (`MAX_BOUNCES`), tracing light as it reflects, refracts, or is absorbed.

The implementation of these methods allowed for the rendering of Scene 0, the Shirley Weekend scene, as shown in Figure 1 .



(a) Scene 0 – No Negative Radius



(b) Scene 0 with Professor's Correction

Figure 1: Renderings of Scene 0

2.3 Soft Shadows

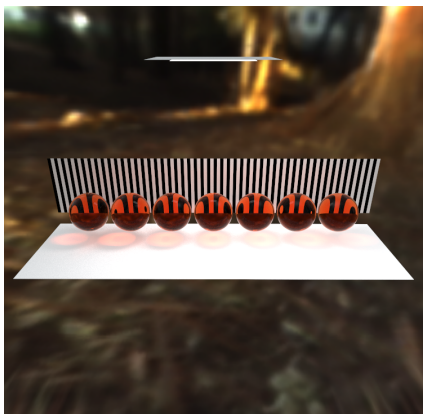
To create soft shadows, the quad light was implemented in scenes 1, 2, 3 and 4, using the new structure defined in `rayColor`, `quadLight`, which contained the coordinates of the three necessary points to define the rectangle, the color, the area and the normal. With this information, the function `directlighting` received the structure, calculated the coordinates of the jittered sample in the world coordinates and proceeded with the calculation of the color.

2.4 Fuzzy Refractions and Caustics

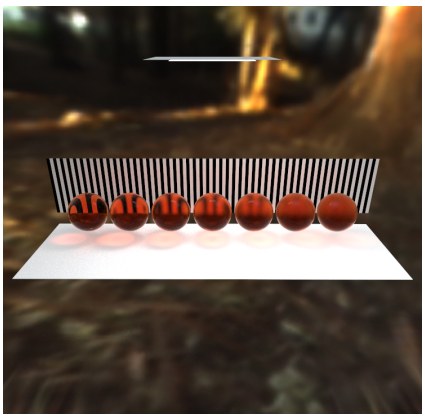
To correctly render Scene 1, fuzzy refractions were implemented by perturbing the refracted ray direction with a random vector inside a unit sphere, scaled by the material's roughness, similar to fuzzy reflections.

Additionally, the visual effect of caustics was approximated by the way refracted rays bent through the spheres and projected onto the floor, even without explicit caustic simulation.

Figure 2 shows Scene 1 rendered with and without fuzzy refractions.



(a) Scene 1 without Fuzzy Refractions



(b) Scene 1 with Fuzzy Refractions

Figure 2: Renderings of Scene 1

2.5 Camera with Orbit and Zoom Movement

To enable detailed exploration and analysis of the rendered scenes, a camera system with orbit and zoom functionality was implemented, controlled primarily through mouse input.

The core of this system is the method `void GetCameraVectors(out vec3 cameraPos, out vec3 cameraFwd, out vec3 cameraUp, out vec3 cameraRight)`, which calculates the camera's position and orientation vectors based on user input and scene-specific parameters.

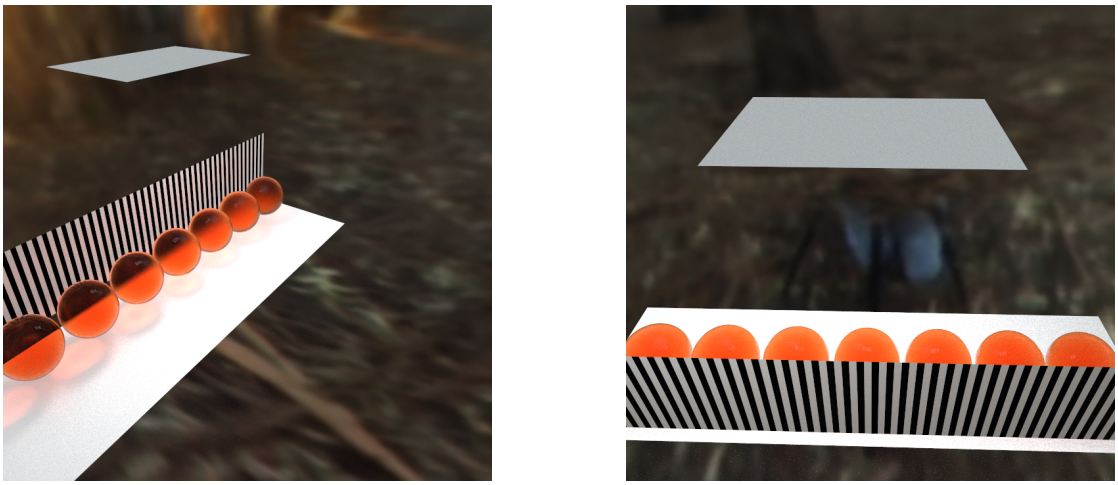
Zooming is activated when the mouse button is pressed, causing the camera to move closer to the scene's focal point. While zoomed in, the user can orbit the camera by moving the mouse: horizontal movement controls the azimuthal rotation, and vertical movement adjusts the elevation angle. The elevation angle is clamped between predefined minimum and maximum limits to prevent unnatural flipping. These angles are converted from spherical coordinates relative to a fixed target point (`c_cameraAt`) into Cartesian coordinates, allowing smooth and intuitive orbiting around the scene's center.

The zoom distance is clamped between scene-specific minimum and maximum values (`minZoom` and `maxZoom`) to maintain proper framing and avoid clipping. This zoom level is incorporated into the spherical coordinate calculations to ensure consistent orbit behavior regardless of zoom.

Upon releasing the mouse button, the camera smoothly zooms back to its original distance while preserving the current viewing angles.

Finally, the method computes the camera's forward (`cameraFwd`), right (`cameraRight`), and up (`cameraUp`) vectors, which define its orientation in 3D space. These vectors are subsequently used to build the view matrix and generate rays for the path tracer.

Figure 3 shows Scene 1 viewed from different angles.



(a) Scene 1 from a Different Angle

(b) Scene 1 from Behind

Figure 3: Renderings of Scene 1 from Different Angles

3 Microfacets BRDF

PBR, or Physically Based Rendering, aims to mimic light behavior in a physically plausible way, resulting in more realistic visuals compared to traditional shading models.

PBR relies on microfacet theory, which describes a surface as a collection of tiny, perfectly reflective mirrors (microfacets). The rougher the surface, the more randomly these microfacets are oriented, causing incoming light to scatter in more directions, which visually softens reflections. At zero roughness, all microfacets are aligned, and the surface behaves like a perfect mirror.

To implement the Microfacets BRDF, we used the metal material and a new one that we created, plastic. This plastic material has an albedo, a roughness, and a specular color (F_0) of 0.04. The metal material contains a specular color and roughness.

The microfacet BRDF is used in the context of the rendering equation. The BRDF describes how light is reflected at a surface and consists of two parts: A diffuse term, and a specular term. The

specular BRDF includes: F, the Fresnel reflectance; D, the normal distribution function; and G, the geometry term.

$$f_r(v,l) = \frac{\rho}{\pi} + \frac{F(v,h)D(h)G(l,v)}{4(n\cdot l)(n\cdot v)} \tag{4}$$

- Fresnel Reflectance: For this we used the Schlick approximation of the Fresnel Reflectance

$$F_{Schlick}(v,h) = F_0 + (1.0 - F_0)(1.0 - (v\cdot h))^5 \tag{5}$$

- Normal Distribution Function: Describes the distribution of the microfacet normals depending on the roughness. For this, we used the GGX distribution

$$D_{GGX}(h) = \frac{\alpha^2}{\pi\left((n\cdot h)^2(\alpha^2-1)+1\right)^2} \qquad \text{where } \alpha = r^2 \tag{6}$$

- Geometry Term: Describes the masking and shadowing effect that occurs in the microfacts depending on the direction of incidence of the light and the viewing direction of the observer. For this we used the G1 GGX Schlick geometry term

$$G_{Smith}(l,v) = G_1(l)G_1(v) \tag{7}$$

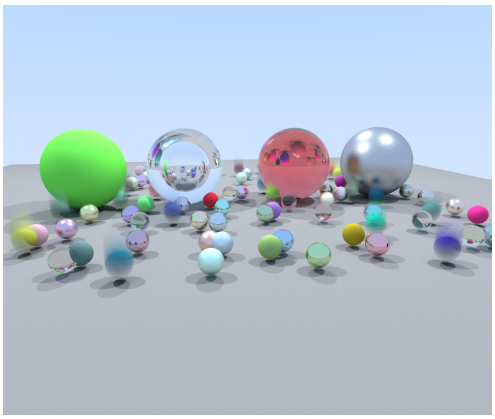
$$G_{1Schlick-GGX}(v) = \frac{(n\cdot v)}{(n\cdot v)(1-k)+k} \qquad \text{where } k = \frac{\alpha}{2} \tag{8}$$

In indirect lighting for the plastic material, we use importance sampling to choose between diffuse scattering and specular reflection, based on Schlick approximation of Fresnel reflectance. Specifically, we compute the Fresnel term, average it to get a scalar probability of reflecting, and use that to decide.

If specular is chosen, we reflect the incoming ray and apply roughness-based jittering to simulate microfacet spread.

If diffuse is chosen, we use a random direction based on the surface normal.

To maintain energy conservation, we divide each contribution by its sampling probability (prob for specular, 1 - prob for diffuse).



(a) Metal material with roughness=0.4

(b) Plastic material

4 Cubemap Background

To enhance the visual appeal and realism of the rendered scenes, different cubemap backgrounds were employed. Cubemaps consist of six square images mapped onto the faces of an imaginary cube that surrounds the scene. This setup creates a seamless 360-degree environment, providing realistic distant backgrounds and reflections. Figure 5 illustrates the layout of a typical cubemap texture.

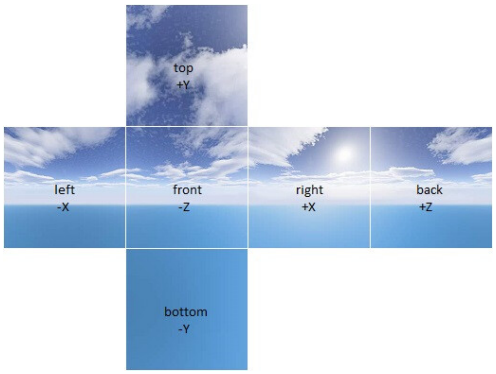


Figure 5: Example Layout of a Cubemap

Two cubemaps were used in this work: one depicting a forest environment, and another showing a cityscape. The cubemap textures were loaded via `iChannel1` using a wildcard pattern, for example:

```
#iChannel1 "file://cubemaps/yokohama_{}.jpg" // Use a single wildcard for CubeMap
```

To correctly display the colors of the cubemap backgrounds, the sampled sRGB textures were converted to linear color space before being used in lighting calculations. This ensures physically accurate lighting and shading when combined with scene materials.

The forest cubemap is shown in Figures 2 and 3, while Figure 6 shows Scene 1 rendered with the city cubemap background.

Figure 6: Scene 1 Rendered with a City Cubemap Background