Physics of Diamonds - Ray Tracing and Optical Physics

LYLIA MESA, Télécom Paris - IGR Track - Fundamentals of Computer Graphics

This project explores the capabilities of ray tracing by simulating the optical and kinematic physics of gemstones. While my initial ideas involved complex scene geometry, the focus shifted to the rendering of diamonds, specifically their refractive and reflective properties. To demonstrate these effects dynamically, a physics engine was implemented to simulate collisions and free fall, allowing the diamond to spin and interact with light in real-time. The core contribution is a study of chromatic dispersion, total internal reflection (TIR), and Fresnel equations within a custom OpenGL ray tracer.

1 INTRODUCTION

When approached with the requirement to use Ray Tracing for the Final Project, my immediate goal was to leverage the technique's strength in photorealistic light computations. Instead of relying on high-polygon geometry or complex scene layouts, I focused on the material properties of gemstones, and specifically diamonds.

While the term "gemstone" is very vast and regroups stones with a huge range of properties [1], diamonds exhibit unique optical behaviors, specifically high refraction and dispersion, that are difficult to approximate with standard rasterization. To showcase these effects, the project also incorporates a rigid body physics simulation. By allowing the diamond to collide with the ground, spin, and tumble, the renderer can display how light paths shift dynamically as the object's orientation changes relative to the light sources.

In this way, the rendering pipeline is the most critical component, designed to bridge optical physics with practical computer graphics implementation.

2 SYSTEM OVERVIEW

The application presents a scene with a diamond mesh and two light sources. The user has full control over the camera and the simulation state:

- **Camera Control:** Activate Control with C, rotate via Left Click + Drag ; up, down, left, right via arrow keys, Zoom-In and Out via I and O ; Reset position with F
- **Physics:** Toggle free fall with P ; Reset position with R.
- **Environment:** Switch floor and wall material between checkered matte (for shadows) and mirror (for reflections) using M ; switch between dark and light backgrounds using L ; activate position control for red light (left) with 1, blue light (right) with 2 and control with the same keys as the camera
- **Geometry:** Toggle background walls with W .
- **Export:** Save the current frame as a `.tga` file with S .

** Note: All keys are given with a qwerty convention. For instance, using azerty, M is ?. ** Note 2 : Visibility is pretty much compromised in checkered mode with walls, because of the high IOR values and the pattern's reflection within the diamond. However it is faster to render and to see the perspective of the scene than in mirror mode, which is supposed to be the "main" mode.*

Author's address: Lylia Mesa, Télécom Paris - IGR Track - Fundamentals of Computer Graphics .

3 GEOMETRY AND DATA

The scene consists of a procedural infinite plane (the ground), optional walls, and a diamond mesh.

Though my first goal was to render multiple (rather) high poly gem meshes, it turns out the rendering was computationally very demanding, making the animations or moving in the scene extremely difficult - so I only kept one diamond mesh, with around 70 faces.

While geometry is not the primary focus, the import process required specific handling. The standard `.off` format used in previous labs proved insufficient for the diamond's sharp facets, as online `.obj -> .off` converters distorted the mesh topology.

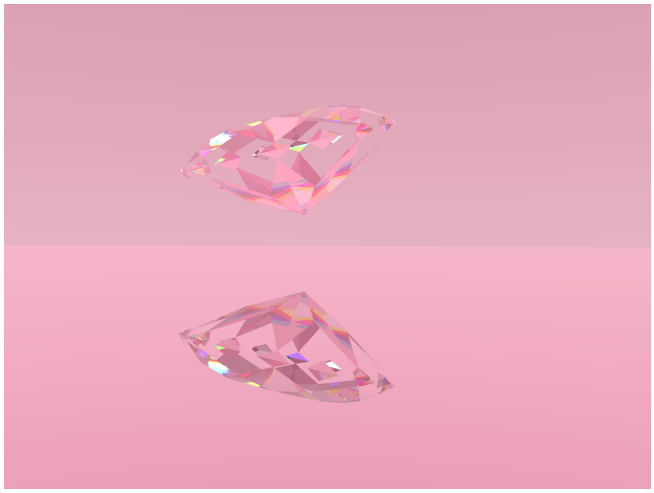


Fig. 1. A falling diamond and its reflection, obtained during a simulation.

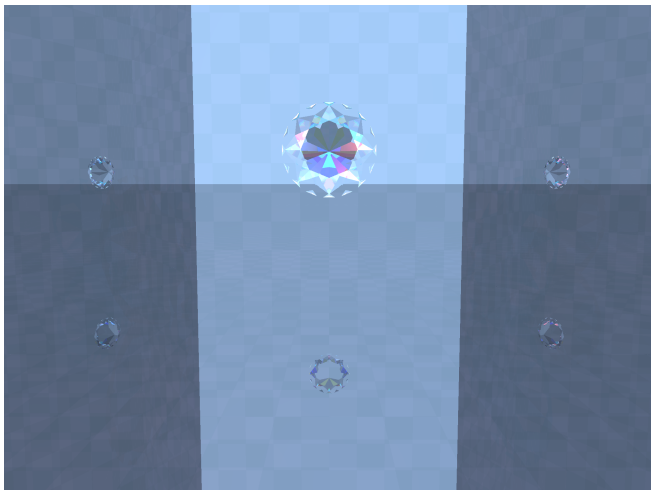


Fig. 2. System interface in light mode, with checkered mirror material, and walls. The diamond is suspended before the physics simulation is toggled.

After loading the `.obj` file into the project, an important matter was its initialization. Since I downloaded the mesh from a web page, I had no idea about its default size, rotation and position. And indeed, loading it directly without any pre-processing was inefficient; the mesh was difficult to position and rotate as wanted within the scene. To fix this, I decided to normalize and center the diamond mesh, through the following code.

```
//main.cpp
void initMeshToTexture(std::shared_ptr<Mesh> mesh) {

    const auto& vertices = mesh->vertexPositions();
    const auto& triangles = mesh->triangleIndices();

    // Calculate the mesh's center
    glm::vec3 center(0.0f);
    for (const auto& v : vertices) center += v.position;

    if (!vertices.empty()) center /= (float)vertices.size
    ();

    // Calculate radius (Max distance between vertices)
    float maxDistSq = 0.0f;

    for (const auto& v : vertices) {
        glm::vec3 diff = v.position - center;
        float dSq = dot(diff, diff);

        if(dSq > maxDistSq) maxDistSq = dSq;
        }

    float radius = sqrt(maxDistSq);
    if(radius < 1e-6) radius = 1.0f;

    std::vector<float> bufferData;
    bufferData.reserve(triangles.size() * 3 * 3);

    // Centering + Normalizing (Scale to 1)
    for (const auto& tri : triangles){

        glm::vec3 v0 = (vertices[tri.x].position - center
    ) / radius;
        glm::vec3 v1 = (vertices[tri.y].position - center
    ) / radius;
        glm::vec3 v2 = (vertices[tri.z].position - center
    ) / radius;

    // push back the new coordinates of each vertex
    }

    // Store the updated Data
    // Update the global scale
}
```

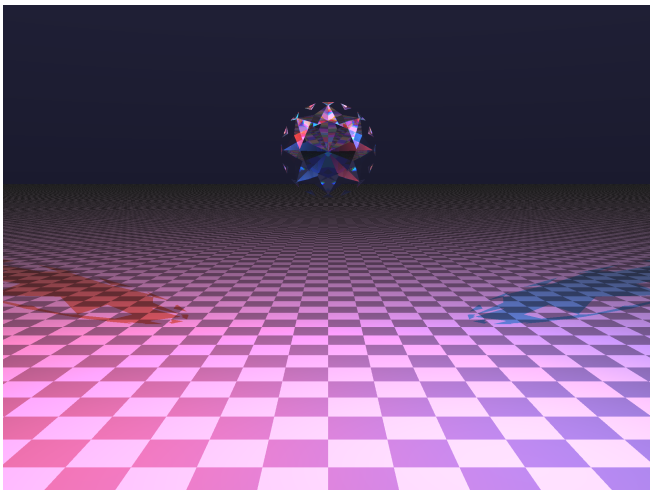


Fig. 3. System interface in dark mode, with checkered material, and no walls. The diamond is suspended before the physics simulation is toggled.

4 PHYSICS SIMULATION

The simulation is based on a free-fall model with gravity, which I worked on during a previous lab session. However, interactions with a surface (hence collisions) seemed to be an interesting extension, showcasing optical properties of the diamond even better, through bouncing or spinning.

4.1 Inertia Approximation

While velocity or position don't rely on the mesh's geometry, angular momentum does, through inertia. Considering the diamond mesh as a cube or sphere would be inaccurate, because it neither has the same symmetry properties nor mass distribution. So, I approximated the diamond's inertia tensor using a cone model :

$$I_{cone} = \begin{bmatrix} \frac{3}{20}M(R^2 + 4h^2) & 0 & 0 \\ 0 & \frac{3}{10}MR^2 & 0 \\ 0 & 0 & \frac{3}{20}M(R^2 + 4h^2) \end{bmatrix} \quad (1)$$

This approximation yielded rotational behaviors that felt more physically grounded for the gem's shape.

4.2 Collision handling

Instead of treating the floor as a mesh with a bounding box, it is defined as a spatial boundary at $y = -1$. This simplifies the collision logic to a plane-check, avoiding expensive mesh-mesh intersection tests.

- **Bounding volume:** However, the biggest challenge of this part is the diamond shape itself, and how to bound it for collisions with that spatial boundary. Since Collision Handling is not the main part of this project, and that the main artifact was the diamond's free fall movement, to display optical phenomena - I considered the diamond's collision bounding box as a sphere. Now, while this truly simplifies the problem and offers fairly satisfying free fall movements and bouncing (which are guided by the intertia given above), the diamond stabilizes after a few seconds in an unrealistic angle. Even though the top half of the diamond is actually close to a sphere (which explains why I picked a sphere bounding box and not AABB), its bottom isn't. This results in the diamond balancing on an imaginary semi-sphere (My bounding box supposition lets the program consider that the diamond is a closed sphere, even though it has the inertia of a cone). Since this part could definitely be improved further, I did try implementing the collision handling using the mesh's vertex positions (by looping on them and comparing their y-value to the floor's), but probably due to the mesh's complex geometry, the diamond would tend to stabilize half inside the ground.
- **Position Projection:** The main goal of collision detection is to stop the bounding box from going behind boundaries

(here, the ground). So, in case it happens, the mesh has to be "projected" back into its allowed zone.

```
// Rigidsolver.hpp
//Positional projection, where X.y designates the y coordinate of the position, floorY the floor's position
body->X.y = floorY + radius;
```

- **Tumbling simulation**: Since a sphere is perfectly round, the bounding box doesn't naturally "tumble" when hitting a flat surface (it just rolls). So, to fake the tumbling of a sharp diamond bouncing on its corners, I injected "Random Torque" into the angular momentum whenever a hard impact occurs.

```
if (std::abs(body->V.y) > 2.0f) {
    Vec3f randomTorque((float(rand())/RAND_MAX - 0.5f) * 10.0f, 0.0f, ...);
    body->L += randomTorque;
}
```

- **Energy restitution**: The diamond bounces on the ground, depending on how much energy it conserves after entering in contact with the floor. This phenomenon can be expressed as so, with e defined as the ratio of final to initial relative velocity:

  $v_{final} = -e.vinitial$

  - If e=1.0: Perfectly elastic collision (like a superball), the diamond will make big bounces for a long time (and even forever if there is no air friction or condition to slow it down)
  - If e=0.0: Perfectly inelastic collision (as wet clay)

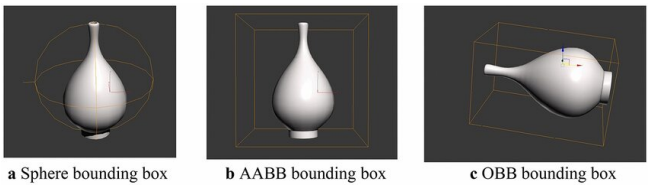
```
if (body->V.y < 0) { // bounce if the diamond is falling down
    // ...
    body->P.y *= -restitution; //restitution
    body->V.y = body->P.y / body->M;
}
```

[7]

- **Velocity Thresholding:** Realistically, the previous formula iteratively reduces the speed to a close value to zero, but it cannot stop the diamond from making "small bounces". So, a solution would be to approximate the speed to zero once it reaches a certain threshold. When an object is "resting" on the ground, gravity pulls it down, and the floor pushes it up. In a discrete stepper, this looks like a series of tiny bounces.

```
if (body->V.y > -0.1f) {
    body->V.y = 0.0f; // stop vertical movement
} else {
    body->P.y *= -restitution;
}
```

This stops the endless "jittering" artifact.



a Sphere bounding box **b** AABB bounding box **c** OBB bounding box

Fig. 4. Bounding Boxes © Baiqiang Gan

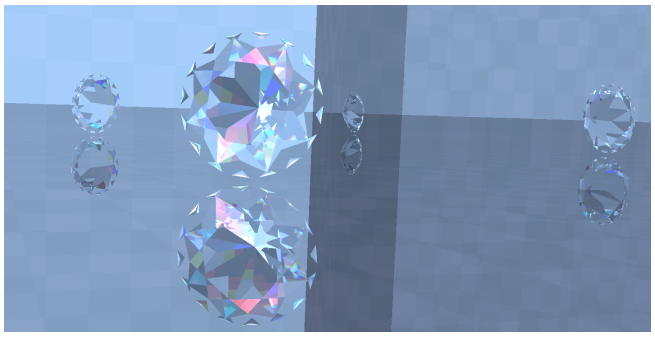


Fig. 5. Screenshot taken during simulation.

- **Friction and damping:** By default, only gravitational forces are applied onto the diamond. However, if the diamond stops bouncing (and has a null vertical speed because of velocity thresholding), it can still slide across the floor and/or spin in place. In practice, Coulomb friction -which gives a relationship between the tangential and normal reaction of the floor surface- would apply. However, an actual implementation of it would be specificially costly, so I chose to apply a simple damping factor.

```
if (isTouchingGround){
// While touching the ground, apply friction to stop the spin & slide (no bounce case)
body->P.x *= friction;
body->P.z *= friction;
body->V.x *= friction;
body->V.z *= friction;
body->L *= 0.75;
body->omega *= 0.75;
```

5 RENDERING IMPLEMENTATION

5.1 Recursive Whitted Ray Tracing

My core renderer is a Recursive Whitted Ray Tracer [2]. This method allows rays to spawn "child" rays upon hitting a surface. When a ray intersects a transparent or reflective surface, it generates secondary rays (Reflection or Refraction) and accumulates the resulting color.

Since GLSL (OpenGL Shading Language) does not support true recursion, I implemented an iterative loop with a fixed depth to simulate the bounces (I fixed it at 12 iterations, which seemed to be a good compromise between visual quality and computation time). The general reasoning behind this the ray tracer goes as follows :

```
for (int depth = 0; depth < MAX_DEPTH; depth++) {
    // 1. Find closest intersection
    // 2. Compute surface normal
    // 3. Update ray origin and direction for next bounce
    // 4. Accumulate color
}
```

The color computation takes physical phenomena into account (refraction, reflection, etc.), and these are tackled in the optical physics section.

5.2 Ray Generation

In the physical world, light hits an object and bounces *into* a camera. In ray tracing, that is simulated in reverse for efficiency. Rays are

shot *from* the camera (p̃in hole camera), through a virtual pixel grid (which is the image plane), and into the scene. So, for every pixel (x,y) the program constructs a vector D :

$$D = F + (U.\alpha.tan(2\theta))R + (v.tan(2\theta))U \tag{2}$$

Where:

- F, R, U are Forward, Right, and Up camera vectors.
- α is the Aspect Ratio.
- θ is the Field of View (FOV)

My implementation of the ray generation goes as follows :

```
void main() {
    // Normalized Device Coordinates (NDC) : converts
     pixel coordinates (0 to width, 0 to height) to a
     range of -1 to +1
    vec2 uv = (gl_FragCoord.xy / resolution) * 2.0 - 1.0;

    // Aspect Ratio Correction : stretches the X-axis so
     the image doesn't look squished on wide screens
    float imageAspectRatio = resolution.x / resolution.y;

    // Field of View (FOV) : we calculate the size of the
      "sensor" based on the camera's zoom (fov)
    float tanFov = tan(fov / 2.0);

    // Ray Direction Construction
    // We use the formula above (2), which combines the
     Camera's basis vectors (Right R, Up U, Forward F) to
     point the ray (cameraDir is the Forward vector)
    vec3 rayDir = normalize(cameraDir
                  + (uv.x * imageAspectRatio * tanFov) *
     cameraRight
                  + (uv.y * tanFov) * cameraUp);

    // all primary rays start at the camera's position.
    Ray r;
    r.origin = cameraPos;
    r.dir = rayDir;

    // ... trace function
}
```

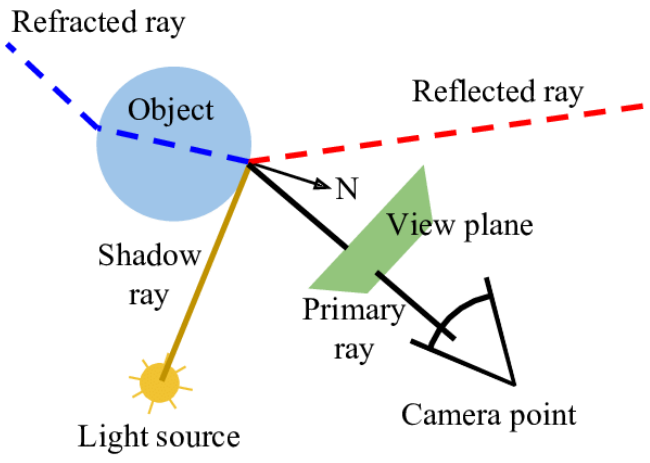


Fig. 6. Key Elements in Ray Tracing. © José Pedro Aguerre

5.3 Intersection (Möller–Trumbore)

To check if a ray hits the diamond (or more specifically, one of its faces), I used the Möller–Trumbore algorithm [3]. Instead of calculating the plane equation and then checking if the point is inside the triangle (which is slower), this algorithm solves the equation (in t,u,v, but only t is actually useful later on to compute the intersection point):

$$P = O + tD = (1 - u - v)V_0 + uV_1 + vV_2 \tag{3}$$

where O is the origin point, D the direction vector, and u, v act like weights for barycentric coordinates.

More specifically : if $u \geq 0$, $v \geq 0$, and $u + v \leq 1$, the point lies inside the triangle.

If any condition fails, the ray missed the tri.

This way, the intersection point P is also expressed as a weighted average of the triangle's three vertices (V0,V1,V2).

My code implements this theory :

```
// raytracer.frag - traceScene loop
for (int i = 0; i < numTriangles; i++) {
    // ... (vertex fetching code)
    vec3 edge1 = v1 - v0;
    vec3 edge2 = v2 - v0;

    // Calculate Determinant (Cramer's rule setup)
    vec3 h = cross(r.dir, edge2);
    float a = dot(edge1, h);

    // Parallel Check : If 'a' is near 0, the ray is
     parallel to the triangle plane (miss)
    if (a > -0.00001 && a < 0.00001) continue;

    float f = 1.0 / a;
    vec3 s = r.origin - v0;

    // Barycentric Coordinate U : check if the hit is
     within the triangle's edges along one axis
    float u = f * dot(s, h);
    if (u < 0.0 || u > 1.0) continue; //miss

    // Barycentric Coordinate V
    vec3 q = cross(s, edge1);
    float v = f * dot(r.dir, q);
    if (v < 0.0 || u + v > 1.0) continue; //miss (Hit
     outside triangle bounds)

    // Distance (t)
    // If we survived the checks above, we have a hit
    float t = f * dot(edge2, q);

    if (t > EPSILON && t < closest.t) {
        closest.t = t;
        closest.hit = true;
        // ... normal calculation ...
    }
}
```

6 OPTICAL PHYSICS: THE DIAMOND LOOK

The visual appearance of the diamond is entirely procedural, derived entirely from physical laws governing light interaction.

It was very exciting to work on this part, since I was taught most of these physical laws (Snell-Descartes, chromatic dispersion...) during classes in the past few years, and the implementation allowed me to see their logic even better.

6.1 Refraction (Snell's Law)

When light enters the denser diamond medium, it slows down and bends. This is governed by Snell-Descartes' Law:

$$\frac{\sin\theta_2}{\sin\theta_1} = \frac{n_1}{n_2} = \eta \tag{4}$$

where n_1 is the Index of Refraction (IOR) of the first material (air), n_2 the IOR of the second material (diamond) θ_1 the incident angle of the light ray (in the air) θ_2 the reflection angle in the second material (in the diamond)

In GLSL, this is handled by the built-in `refract` function, which takes as argument the ray vector, the surface normal vector and the ratio of the two IOR, expressed by the law. It returns the refracted light ray's vector. Since we have n_1=1 (air) and the law, if the light ray enters the diamond (facing the front face) we use $\eta = \frac{1}{n_2}$ Otherwise (back face), we use $\eta = n_2$.

```
float eta = rec.frontFace ? 1.0 / ior : ior;
vec3 refracted = refract(r.dir, N, eta);
```

6.2 Chromatic Dispersion

In reality, the Index of Refraction (IOR) varies with wavelength. This causes white light to split into its constituent colors (red, green, blue), through a phenomenon known as dispersion.

To simulate this "rainbow" effect, I trace the scene three times per pixel, once for each color channel, and with distinct IOR values:

- **Red:** $IOR = 1.70$ (Low refraction)
- **Green:** $IOR = 1.80$
- **Blue:** $IOR = 1.90$ (High refraction)

Although real diamonds have an average IOR closer to 2.4, I picked these values to exaggerate the effect and make the spectral separation clearly visible, even with a fixed depth in ray tracing.

```
//main.cpp

vec3 colR = traceWhitted(r, 1.70);
vec3 colG = traceWhitted(r, 1.80);
vec3 colB = traceWhitted(r, 1.90);
```

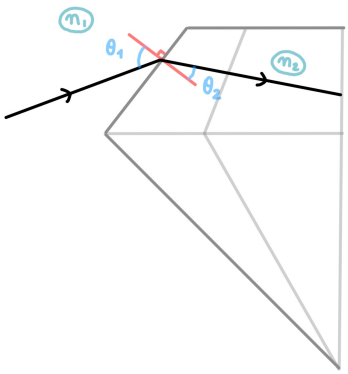


Fig. 7. Snell-Descartes Law : When hitting interface between two different materials, incident light ray results in a refractive ray and a reflective ray

```

vec3 finalColor = vec3(colR.r, colG.g, colB.b);
```

6.3 Total Internal Reflection (TIR)

A critical aspect of a diamond's brilliance is Total Internal Reflection, which is a specific case of the Snell-Descartes law (2). If a light ray tries to exit the diamond at a shallow angle (beyond the critical angle), it cannot escape and is reflected 100% back inside the diamond.

This traps the light, allowing it to bounce internally until it strikes a facet at a steep enough angle to exit. This is what makes the gem "glow" rather than just look transparent like glass. Mathematically, if the refraction vector has a length of 0, TIR occurs:

```
//raytracer.frag
```

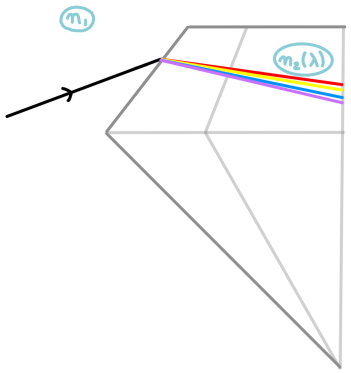


Fig. 8. Chromatic dispersion illustration

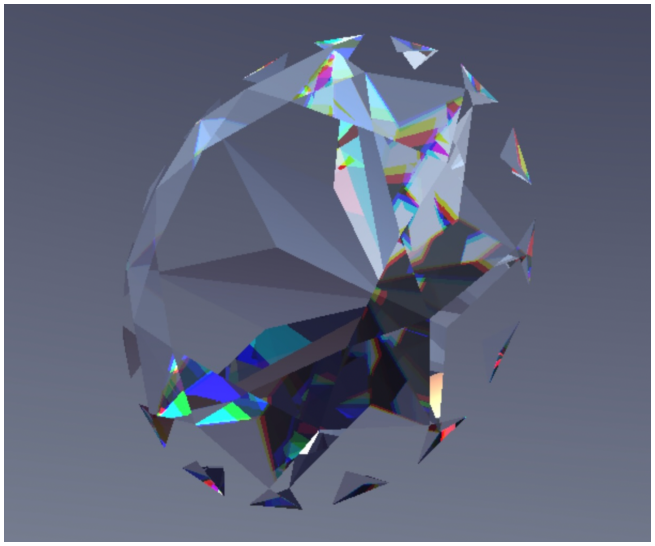


Fig. 9. Chromatic dispersion seen in the diamond (screenshot taken during development, so the scene looks different from the final result)

```
vec3 refracted = refract(r.dir, N, eta);

if (length(refracted) == 0.0) {
    r.dir = reflectDir;
    r.origin = hitPoint + N * EPSILON;
}
if (length(refracted) == 0.0) {
    r.dir = reflect(r.dir, N); // Trapped inside
}
```

Though not necessary to compute explicitely on its own, θ_c can be derived from the Snell-Descartes law, with $\theta_2 = 90°$.

6.4 Fresnel Equations

As for most materials, diamond's surfaces are more reflective at grazing angles. Fresnel equations compute reflective and refractive light components' intensity (while Snell-Descartes laws return their angle).

$R = \frac{I_r}{I_i}$ where I_i and I_r are the incident and reflection's intensity
Specifically :

$$R = \frac{1}{2}\left[\left(\frac{n_1\cos\theta_i - n_2\cos\theta_t}{n_1\cos\theta_i + n_2\cos\theta_t}\right)^2 + \left(\frac{n_2\cos\theta_i - n_1\cos\theta_t}{n_2\cos\theta_i + n_1\cos\theta_t}\right)^2\right]$$

where

$$\cos\theta_t = \sqrt{1 - \left(\frac{n_1}{n_2}\right)^2 (1 - \cos^2\theta_i)}$$

and θ_i is the incidence angle of the incoming light ray.

Nonetheless, implementing this straight-forwardly is computationally expensive. So, instead, I used Schlick's approximation [4] , which calculates a term F that blends the reflective and refractive components:

$$F(\theta) = F_0 + (1 - F_0)(1 - \cos\theta)^5 \quad (5)$$

where $F_0 = (\frac{n_{air} - n_{diamond}}{n_{air} + n_{diamond}})^2$, and $n_{air} = 1$. This ensures the diamond has realistic glare at the edges. The implementation is rather straightforward :

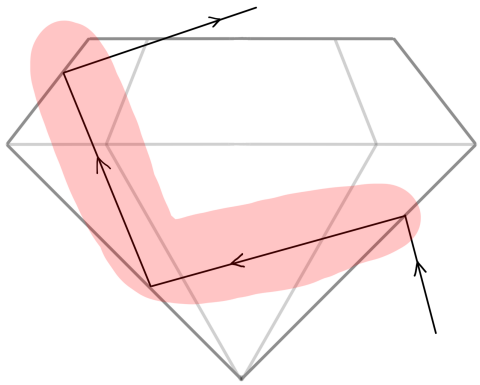


Fig. 10. TIR concept : the diamond shines due to the trapped rays

```
float f0 = pow((1.0 - ior) / (1.0 + ior), 2.0);
float fresnelTerm = f0 + (1.0 - f0) * pow(1.0 - abs(dot(-
    r.dir, N)), 5.0);

vec3 reflectDir = reflect(r.dir, N);
vec3 envReflection = getEnvironment(reflectDir);
finalColor += throughput * envReflection * fresnelTerm;
```

6.5 Specular Highlights (Blinn-Phong)

This is not implemented anymore in the final version of the code, but I still found it interesting to mention. Even with refraction and reflection, the raw mesh initially appeared somewhat hollow. To resolve this, I added a Blinn-Phong specular term to simulate the direct reflection of the light source on the surface facets [5]:

$$H = \frac{L + V}{||L + V||}, \quad I_{spec} = (\mathbf{N} \cdot \mathbf{H})^{shininess} \quad (6)$$

This adds the sharp white highlights characteristic of polished gems, which is specifically noticeable when the diamond rotates.

Reflective materials To showcase the diamond's material, I textured the ground as a mirror. To implement this, I

```
// raytracer.frag

// Surface shine (direct specular highlight)
vec3 sunDir = normalize(vec3(0.5, 1.0, 0.5));
vec3 viewDir = normalize(cameraPos - hitPoint);
vec3 halfVector = normalize(sunDir + viewDir);

// Blinn-Phong
float NdotH = max(dot(N, halfVector), 0.0);
float specular = pow(NdotH, 100.0);

// Add the shine to the final color
finalColor += vec3(1.0) * specular * 0.8; //the 0.8 is an
    intensity term, this looked more realistic
```

6.6 Light Absorption (Beer-Lambert)

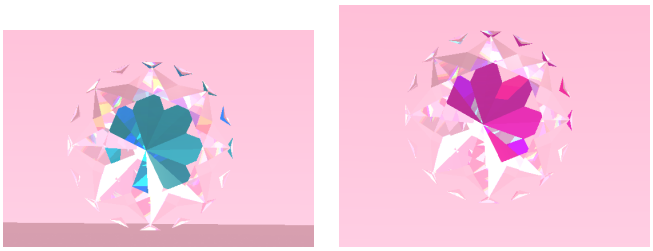
In opposite to regular glass, gemstones and diamonds absorb light. The Beer-Lambert law states that light intensity decays exponentially with distance and density:

$I = I_0 e^{-\sigma d}$

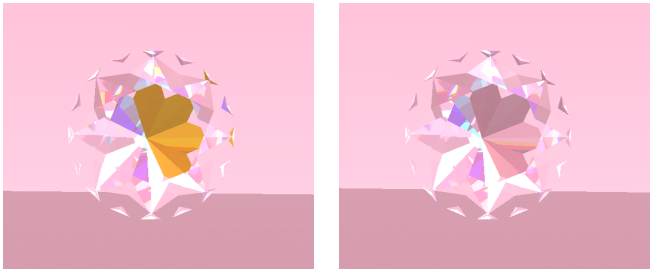
where σ is the absorption coefficient (color) and d is the distance traveled inside the crystal. Since light is composed of R,G,B components, σ is a vector of three absorption coefficients in this case.

```
// raytracer.frag

// Distance traveled within the diamond
// 'rec.t' is the distance from the last hit to the
    current one
float dist = rec.t;

// Absorption Coefficient
vec3 absorbance = vec3(2.0, 1.0, 0.1);

    // Beer-Lambert Law : only applied if the light ray
    is hitting the back face (exiting), since that would
    mean it just traveled through the volume
if (!rec.frontFace) {
    vec3 transmission = exp(-absorbance * dist);
    throughput *= transmission;
}
```

By playing with the absorption coefficients' values, the diamond takes different "tints" that follow additive color theory. This offers the possibility to create other gemstones, like sapphire, ruby, etc.



(a) σ=(0.5,0.1,0.1) (b) σ=(0.1,0.5,0.1)



(c) σ=(0.1,0.1,0.5) (d) σ=(0.2,0.2,0.2)

Fig. 11. Different values of the absorption vector that showcase additive color theory. Bottom right is using the final values set.

For example, if a gemstone looks deep blue, it means it has absorbs a lot of Red and Green, and less blue.

6.7 Limits of optical simulations and solutions

As mentioned previously, I implemented my ray tracer using a loop of a fixed depth. So, while the key physical phenomena are implemented, the optical simulation would first return certain "black spots" within the diamond or its reflection. This is due to the ray tracer's loop's fixed depth; in case a light ray is refracted inside the mesh, but does not get to exit the diamond before the loop terminates (so if needs to bounce inside the diamond more times than the loop's iterations), no light is exits the mesh. The renderer then returns black for that ray.

As a solution, I suppose that (even though it is actually still trapped in the diamond), the light ray *would* eventually hit the environment. I sample the environment map using the current reflection vector (in the active iteration). While this is physically inaccurate (for example, the ray could still be inside my diamond, but the program samples the sky), it provides enough information to fill the black pixel, making it look like a sparkle. (The difference is surprisingly barely noticeable).

```
//raytracer.frag - inside traceWhitted function loop

for (int depth = 0; depth < MAX_DEPTH; depth++) {
    // ...
    // If we reached the last bounce (depth 11) and the ray is still bouncing, force a sample from the environment
    if (depth == MAX_DEPTH - 1) {
        finalColor += throughput * getEnvironment(reflect(r.dir, rec.normal));
        break;
    }
    // ...
}
```

A 1990 SIGGRAPH Paper by James Arvo and David Kirk provides the theoritical proof of the "black spots" (through a truncation of the Neumann series of the rendering equation), and explain it as a violation of energy conservation. While my fix is deterministic, Arvo and Kirk presented an alternative probabilistic method. [6] Yet complexity wise, using a deterministic solution seems to be the most efficient.

7 OPTICAL PHYSICS: THE MIRROR LOOK

Though the diamond's material rendering was a bigger part of the project, I also implemented a mirror material to be able to witness the diamond's reflection (or, different angles of the diamond at the same time), for the walls and floor.

I implemented the mirror material using specular reflection.

```
//raytracer.frag - inside traceWhitted function loop

if (u_mirrorMode == 1) {
    // Base Material : darker albedo for the mirror surface itself
    vec3 albedo = pattern > 0.5 ? vec3(0.05) : vec3(0.1);
     //Checkered pattern (less visible in mirror mode, but helps visualising the perspective better)

    // Local Color
    finalColor += throughput * (albedo * 0.2 + diff * albedo * shadow) * 0.2;

    // Absorption : The mirror reflects 80 % of the light coming in
    throughput *= vec3(0.8);

    // Reflection
    // update the ray's origin to the hit point (pushed slightly by epsilon to avoid self-intersection)
    r.origin = hitPoint + N * EPSILON;

    // Calculate the perfect reflection vector (GLSL built-in)
    r.dir = reflect(r.dir, N);

    // Recursion : 'continue' forces the loop to skip the rest of this iteration and start
    // the next bounce (depth+1) with the new ray
    continue;
}
```

Very similarly to the diamond material, I implemented a deterministic solution to the "black spots" that appear on the mirror (in the diamond's reflection).

8 CONCLUSION

My project implements a renderer that captures the main physical properties of diamonds (motion-wise and optics-wise). By integrating rigid body dynamics with recursive ray tracing, the system demonstrates interactions between geometry, motion, and light. Though my current implementation does not allow it (because the rays are traced from the camera), photon mapping could be an interesting extension to my current work, because it can compute the effects of the light rays that went through and around the diamond (or any other body), on the ground. (See Figure 14)

During my research for this project, I also read about Path Tracing, which seems to offer even higher precision and realism, especially for complex light scenes. It seems to be the latest "trend" in rendering,

which I'm still far away from, but the results seem truly impressive and the theory interesting!

REFERENCES

[1] Shiyun Jin, Nathan D. Renfro, Aaron C. Palke, and James E. Shigley STRUCTURES BEHIND THE SPECTACLE: A REVIEW OF OPTICAL EFFECTS IN PHENOMEN AL GEMSTONES AND THEIR UNDERLYING NANOTEXTURES 2025

[2] Turner Whitted. 1980. An improved illumination model for shaded display. *Communications of the ACM* 23, 6 (1980), 343–349.

[3] Tomas Möller and Ben Trumbore. 1997. Fast, Minimum Storage Ray-Triangle Intersection. *Journal of Graphics Tools* 2, 1 (1997), 21–28.

[4] Christophe Schlick. 1994. An Inexpensive BRDF Model for Synthetic Image Generation. *Computer Graphics Forum* 13, 3 (1994), 233–246.

[5] James F. Blinn. 1977. Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph.* 11, 2 (1977), 192–198.

[6] James Arvo and David Kirk, "Particle Transport and Image Synthesis", *SIGGRAPH*, 1990. Particle Transport and Image Synthesis. SIGGRAPH 1990.

[7] Guendelman, E., Bridson, R., & Fedkiw, R. Nonconvex rigid bodies with stacking. SIGGRAPH, 2003

[8] Nvidia What is Path Tracing ? link : https://blogs.nvidia.com/blog/what-is-path-tracing/

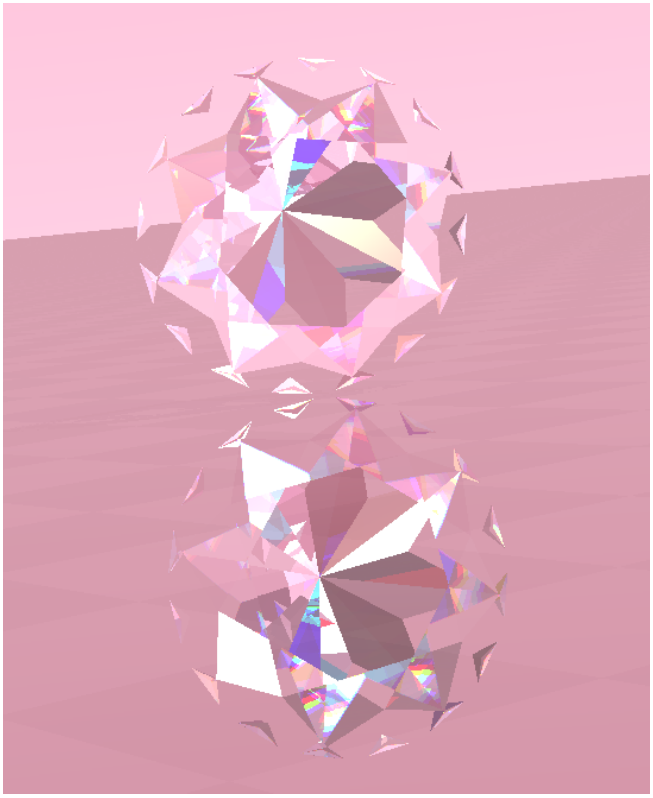


Fig. 12. The diamond with all optical phenomena implemented, mid simulation, with its reflection on the ground. (Background was originally pink in light mode, hence this result

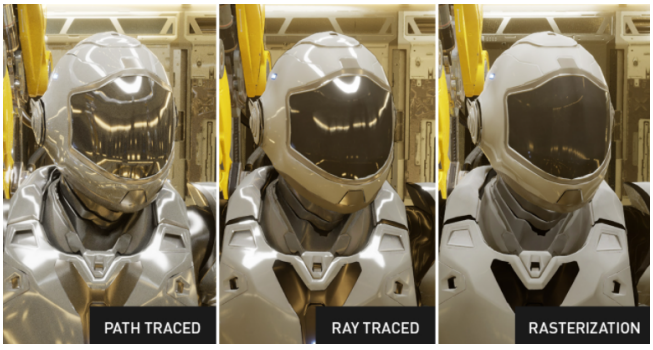


Fig. 13. Comparison of path tracing, ray tracing, rasterization, ©Nvidia

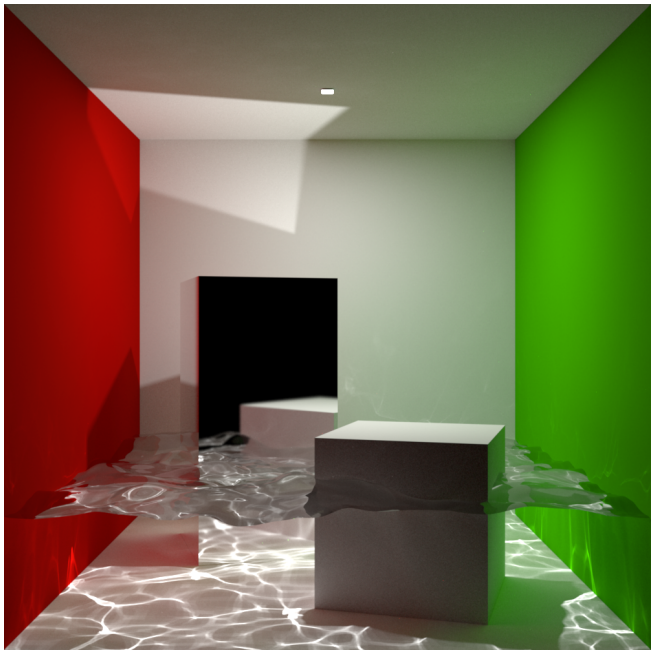


Fig. 14. Underwater reflections using Photon Mapping, ©Appleseed (unknown author)