RTG1IL Real Time Graphics – WS 2024/25
Lab Report 1

Jan Eberwein, s2410629003
Johannes Eder, s2410629004

Abstract

This project showcases a custom raytraced scene in OpenGL/GLSL featuring a new procedural floor pattern (a black–white line grid) and multiple new torus primitives, in accordance with the assignment requirements. The floor pattern is generated by a simple analytic function, while the torus geometry is implemented via a signed-distance function (SDF) and sphere tracing.

1.1 Assignment Overview

The primary objectives were:

- **Implement a new surface pattern** (not a simple checkerboard or solid color) for the floor.
- **Add a new primitive** that was not covered in the basic lab session; in this case, three stacked toruses using an SDF approach.
- **Demonstrate reflections and shading** with at least one bounce, and capture screenshots for the final submission.

We chose a simple black–white line grid floor to fulfill the "new surface pattern" requirement and used an SDF-based torus to fulfill the new primitive requirement.

1.2 Implementation Details

1.2.1 Scene Overview

Our updated code uses a main C++ program (`raytracing.cpp`) that sets up an OpenGL context and a fullscreen quad. A vertex shader (`raytracing.vs.glsl`) and a fragment shader (`raytracing.fs.glsl`) implement the actual ray tracing logic in GLSL. The user can control camera movements and reflection depth; the result is drawn each frame to the screen.

1.2.2 Floor: Black–White Grid Pattern

We removed the original checkerboard/honeycomb patterns and implemented a floor with crisp black grid lines on a white background:

Listing 1.1: Snippet for the black–white grid floor (in (raytracing.fs.glsl)

```
float rayGridFloorIntersection(vec3 ro, vec3 rd, out vec3 floorColor) {
    if (abs(rd.y) < 1e-6) return 1e9; // No hit if ray parallel
    float t = -ro.y / rd.y;
    if(t < 1e-6) return 1e9;         // Hit must be in front of the ray

    vec3 hitPos = ro + rd * t;
    // Each cell is size=1.0; lines have width=0.05
    float cellSize = 1.0;
    float lineWidth = 0.05;

    vec2 modPos = mod(hitPos.xz, cellSize);
    float distToLine = min(modPos.x, cellSize - modPos.x);
    distToLine      = min(distToLine, min(modPos.y, cellSize - modPos.y));

    // If close to a grid line, color is black; otherwise white.
    if(distToLine < lineWidth)
        floorColor = vec3(0.0);
    else
        floorColor = vec3(1.0);

    return t;
}
```

When the ray intersects the plane at `y=0`, we compute its x, z coordinates modulo the cell size. The distance to the closest grid line determines whether we output white or black for each cell.

1.2.3 Adding Torus Primitives

We replaced simpler primitives (spheres, cubes) with three toruses stacked vertically. A torus is defined by two radii (major R and minor r) and can be represented by the following signed-distance function (SDF):

$$\mathrm{sdTorus}(\vec{p}, (R, r)) = \sqrt{\left(\sqrt{p_x^2 + p_z^2} - R\right)^2 \; + \; p_y^2} \; - r.$$

We “sphere trace” the torus by iteratively stepping along the ray until the SDF is below some small ϵ. The core intersection code:

Listing 1.2: Torus intersection via sphere tracing (snippet in `raytracing.fs.glsl`)

```
float intersectTorus(vec3 ro, vec3 rd, vec3 torusCenter, float R, float r) {
    const int MAX_STEPS = 64;
    const float tMax = 100.0;
    float t = 0.0;
    for(int i=0; i<MAX_STEPS; i++){
        vec3 pos = ro + rd * t - torusCenter;
        float dist = sdTorus(pos, vec2(R, r));
        if(dist < EPSILON)
```

```
            return t;   // we have an approximate hit
        t += dist;
        if(t > tMax) break;
    }
    return INFINITY; // no intersection
}
```

For normal computation, we approximate partial derivatives with finite differences around the hit position:

Listing 1.3: Approximating the torus surface normal (snippet)

```
vec3 getTorusNormal(vec3 pos, vec3 torusCenter, float R, float r) {
    float e = 0.0005;
    vec3 p = pos - torusCenter;
    float d0 = sdTorus(p + vec3(e,0,0), vec2(R,r)) - sdTorus(p - vec3(e,0,0), vec2(R
    ,r));
    float d1 = sdTorus(p + vec3(0,e,0), vec2(R,r)) - sdTorus(p - vec3(0,e,0), vec2(R
    ,r));
    float d2 = sdTorus(p + vec3(0,0,e), vec2(R,r)) - sdTorus(p - vec3(0,0,e), vec2(R
    ,r));
    return normalize(vec3(d0, d1, d2));
}
```

Finally, three toruses are placed in the scene at different y coordinates:

Listing 1.4: Stacking three toruses (snippet from `rayTraceScene()`)

```
addTorus(ro, rd, vec3(2.0, 0.5, 2.0), 0.7, 0.2, vec3(0.0, 0.0, 1.0),
         hitDist, hitColor, hitNormal);
addTorus(ro, rd, vec3(2.0, 1.5, 2.0), 0.7, 0.2, vec3(0.0, 0.0, 1.0),
         hitDist, hitColor, hitNormal);
addTorus(ro, rd, vec3(2.0, 2.5, 2.0), 0.7, 0.2, vec3(0.0, 0.0, 1.0),
         hitDist, hitColor, hitNormal);
```

1.2.4 Reflection and Final Image

To achieve reflections, each ray is traced for up to `maxDepth` bounces. After each hit, we compute the reflection direction and continue tracing. A simple Blinn–Phong model and shadow checks provide shading. We keep track of the *fresnel* factor to blend reflections. The final rendered result is shown in Figure 1.1.

1.3 Conclusion

In this exercise, we introduced:

1. A **new floor pattern** using a black–white line grid function.
2. Three **stacked torus primitives** for geometry, implemented via a signed-distance function and sphere tracing.
3. Full reflection logic with multi-bounce support, demonstrating shading, shadows, and basic Fresnel reflection blending.

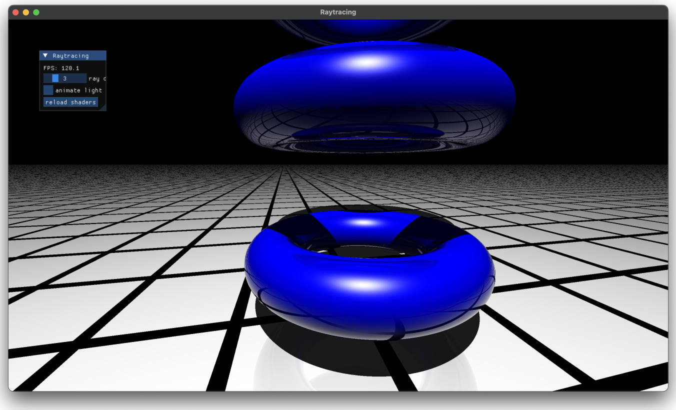

Figure 1.1: Scene with a black–white line grid floor and three stacked blue toruses.

Future improvements could include reducing artifact “banding” on reflections (by raising sphere-tracing steps or implementing an analytic solution), but the scene already demonstrates the major features well.