Project

12105 Rendering - Introduction

Egill Magnússon - s253025

December 18, 2025

Contents

Abstract

This project implements ray intersection with an *elliptical torus* inside a WebGPU path tracer. The torus is represented as an implicit surface in a local coordinate frame; rays are transformed to local space, and intersections are found numerically in WGSL using sign-change bracketing and bisection. A bounding sphere plus a y-slab test is used for early rejection. Scenes containing one or more tori are stored in small `.tori` text files which are parsed in JavaScript and uploaded as a packed GPU buffer.

1 Introduction

This report describes my final project for the Rendering course (MSc, DTU). The goal was to extend a small WebGPU renderer with support for ray intersections against an *elliptical torus* (a torus whose tube cross section is an ellipse instead of a circle).

All work was done individually (not a group project). Scenes containing one or more tori are stored in a small custom text format (`.tori`) that is parsed in JavaScript and uploaded to the GPU.

2 Method

2.1 Torus Definition

Each torus is evaluated in its *local* coordinate system, where the major radius R lies in the xz-plane and the tube has semi-axes a (in the xz-plane) and b (along y). Let $P = (x, y, z)$ be a point in local coordinates and define

$$\rho^2 = x^2 + z^2, \tag{1}$$

$$p = \frac{a^2}{b^2}, \tag{2}$$

$$B = R^2 - a^2. \tag{3}$$

Derivation An elliptical torus can be seen as sweeping an ellipse of radii a (horizontal) and b (vertical) around the y-axis with major radius R. In 2D, the course reference [1] expresses the cross section as two ellipses centered at $x = \pm R$:

$$\left(\frac{(x-R)^2}{a^2} + \frac{y^2}{b^2} - 1\right)\left(\frac{(x+R)^2}{a^2} + \frac{y^2}{b^2} - 1\right) = 0 \tag{4}$$

Expanding and rearranging yields a compact implicit form

$$\left(x^2 + py^2 + B\right)^2 - 4R^2x^2 = 0 \tag{5}$$

where $p = \frac{a^2}{b^2}$ and $B = R^2 - a^2$. To obtain the full torus, the sweep replaces the horizontal coordinate by distance to the axis of rotation:

$$x^2 \;\mapsto\; x^2 + z^2 = \rho^2 \tag{6}$$

This gives the 3D implicit equation used in my renderer: An implicit surface function for an elliptical torus is

$$F(P) = \big(\rho^2 + py^2 + B\big)^2 - 4R^2\rho^2 = 0 \tag{7}$$

This reduces to the standard circular torus when $a = b$ (so $p = 1$).

To see why the tube cross section is elliptical, rewrite $F(P) = 0$ as

$$\rho^2 + py^2 + R^2 - a^2 = 2R\rho \tag{8}$$

which gives

$$(\rho - R)^2 + py^2 = a^2 \quad \Longleftrightarrow \quad \frac{(\rho - R)^2}{a^2} + \frac{y^2}{b^2} = 1 \tag{9}$$

Thus a controls the radial semi-axis of the tube and b controls the vertical semi-axis.

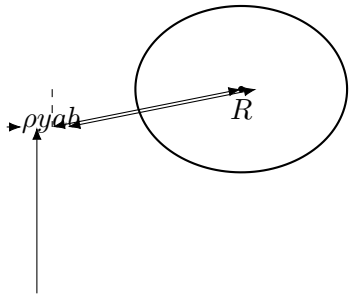

Figure 1: Tube cross section in the (ρ, y) plane: $\frac{(\rho-R)^2}{a^2} + \frac{y^2}{b^2} = 1$.

2.2 Ray Substitution

Ray-surface intersection can be expressed by substituting a ray into the implicit function. For a ray in local space

$$P(t) = O + tD, \qquad t \in [t_{\min}, t_{\max}], \tag{10}$$

the intersection condition is

$$F\big(P(t)\big) = 0 \tag{11}$$

Since $x(t), y(t), z(t)$ are linear in t, both $\rho^2(t)$ and $y^2(t)$ are quadratic in t:

$$\rho^2(t) = (O_x + tD_x)^2 + (O_z + tD_z)^2, \tag{12}$$

$$y^2(t) = (O_y + tD_y)^2 \tag{13}$$

Therefore $F\big(P(t)\big)$ becomes a quartic polynomial

$$c_4t^4 + c_3t^3 + c_2t^2 + c_1t + c_0 = 0 \tag{14}$$

In principle this quartic can be solved analytically, but in WGSL a closed-form solver is undesirable due to single-precision arithmetic and the lack of robust polynomial root-finding utilities. I instead used a numeric bracketing approach described in Section 3.2.

2.3 Normal from Gradient

The shading normal is computed from the gradient of the implicit surface. Let

$$Q = \rho^2 + py^2 + B \tag{15}$$

Then the partial derivatives are

$$\frac{\partial F}{\partial x} = 4xQ - 8R^2x, \tag{16}$$

$$\frac{\partial F}{\partial y} = 4pyQ, \tag{17}$$

$$\frac{\partial F}{\partial z} = 4zQ - 8R^2z. \tag{18}$$

The local normal is $N_{\text{local}} = \nabla F / \|\nabla F\|$. To obtain the world-space normal, N_{local} is rotated by the torus instance rotation.

Computing ∇F analytically avoids finite differencing, which would require additional implicit evaluations and introduces a sensitive step-size parameter.

2.4 Acceleration Structure

The environment (Cornell box) is intersected using a BSP tree over triangles. For the tori, a conservative bounding volume is used in local space to reject rays quickly:

$$\text{Bounding sphere:} \quad r = R + \max(a, b). \tag{19}$$

If a ray misses this sphere, it cannot hit the torus. To further tighten the interval, a simple slab test is applied using two horizontal planes at $y = \pm b$: after computing the ray's entry/exit distances (t_{in}, t_{out}) with the bounding sphere, the corresponding y values are evaluated. The ray can be rejected if both points lie above $+b$ or both lie below $-b$. Only if these bounds pass is the more expensive implicit evaluation performed.

3 Implementation

3.1 Torus Parameters

Each torus instance stores: center position, major radius R, ellipse semi-axes (a, b), index of refraction (IOR), an extinction coefficient σ_t for colored attenuation, and an orientation (encoded as a 3×3 rotation matrix).

On the GPU, the torus data is packed using a `vec4`-based layout for alignment. In practice this is 24 floats per torus instance (6 `vec4`s): center+R, (a, b, IOR), three rotation columns, and extinction σ_t.

Rays are transformed into the torus local frame using a rigid transform (rotation + translation).

With center C and rotation matrix $\mathbf{R}$ (local axes expressed in world space), the shader computes

$$O_{\text{local}} = \mathbf{R}^{\mathsf{T}}(O_{\text{world}} - C) \tag{20}$$

$$D_{\text{local}} = \mathbf{R}^{\mathsf{T}} D_{\text{world}} \tag{21}$$

Normals are rotated back using rotation only (no scaling): $N_{\text{world}} = \mathbf{R} N_{\text{local}}$.

Scenes are stored in human-readable `.tori` files. Each file contains one or more blocks of the form:

```
torus {
 center = x, y, z
 R = ...
 a = ...
 b = ...
 ior = ...
 extinction = r, g, b
 rotation = x:deg, y:deg, z:deg
}
```

The parser reads these blocks, converts rotation angles from degrees to radians, and uploads a packed array of torus structs to a storage buffer.

3.2 Root Finding in WGSL

Instead of implementing a full closed-form quartic solver in WGSL, the intersection uses numeric bracketing on $F\big(P(t)\big)$. After restricting the ray interval using the bounding sphere (and rejecting with the y-slab test), the shader:

Step 1: Samples $F\big(P(t)\big)$ at a fixed number of points along $[t_{\min}, t_{\max}]$.

Step 2: Detects a sign change between consecutive samples ($F(t_i)F(t_{i+1}) < 0$), which brackets a root.

Step 3: Refines the bracket using a small number of bisection steps.

Step 4: Keeps the closest valid root (smallest positive t).

In my implementation the shader uses 64 coarse samples and 6 bisection steps per detected bracket. More coarse samples reduce missed intersections (especially for thin or grazing hits) at the cost of more implicit evaluations.

Since the underlying quartic can have 0, 2, or 4 real roots, a purely sampled approach can in principle miss very thin double intersections at grazing angles; using a conservative step count and a tight ray interval from the sphere intersection helps mitigate this in practice.

The implementation also uses small epsilons to improve stability: the sphere interval is padded slightly and the ray segment is biased to reduce self-intersections.

3.3 Ray-Torus Intersection Check

For each torus instance, the ray is transformed into local coordinates using the inverse rotation (transpose) and a translation by the center. If an intersection is found, the hit point is evaluated in local space, the normal is computed from the gradient, and the normal is rotated back to world space.

In the path tracer, the torus is treated as a refractive object (glass) using its IOR. The extinction σ_t is stored per-instance and is used to tint attenuation through the medium.

4 Results

I added a few demo scenes containing multiple tori.

All images in this report are rendered at 1024×1024 resolution. The tracer uses up to 15 bounces per path in the shader. The torus intersection evaluates the implicit function $F(P)$ up to 64 times along the ray segment (plus a small number of bisection evaluations when a bracket is detected), while the sphere-slab bound rejects rays that cannot hit the torus.

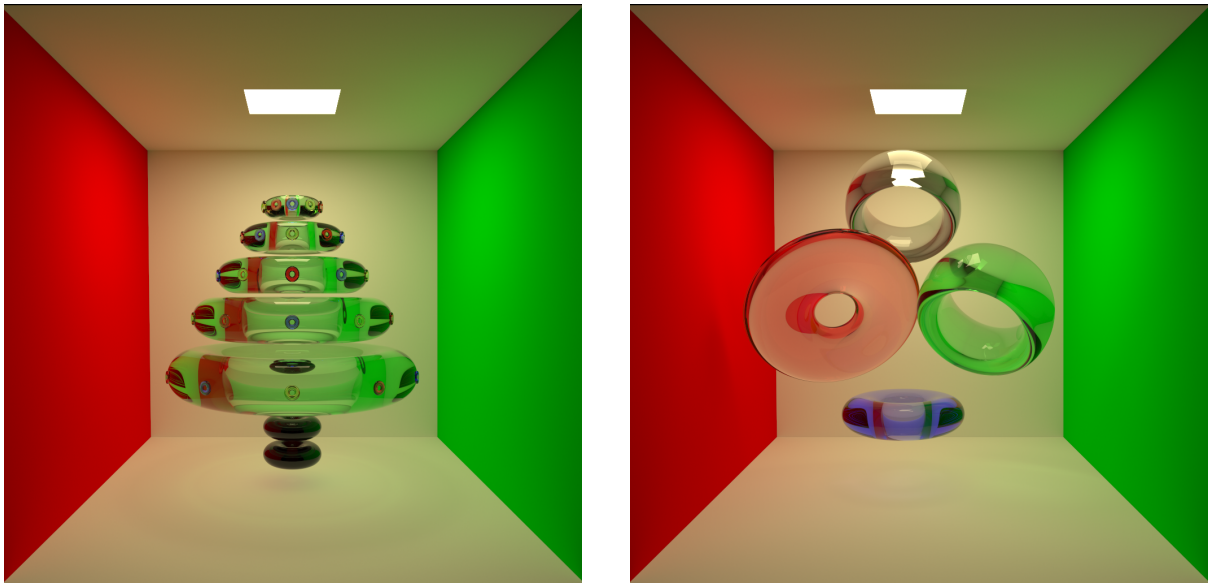

Figure 2: Rendered results. Left: torus Christmas tree scene. Right: elliptical torus scene.

5 Discussion

The elliptical torus intersection integrates cleanly with the WebGPU path tracer: the implicit surface provides both an intersection test and a consistent normal via the gradient. The numeric bracketing approach in WGSL is compact and robust, but it introduces a resolution trade-off: if too few steps are used, very thin features or grazing intersections can be missed. The sphere-slab bound reduces wasted work and keeps the additional cost manageable.

Geometrically, a ray can intersect a torus multiple times (the quartic may have up to four real

roots). For rendering, the closest valid hit is typically sufficient; the implementation therefore searches for the smallest t in the current ray interval.

6 Relation to Project Material

The project handout included a chapter by Cychosz [1], which derives the implicit equation of an elliptical torus, the corresponding quartic polynomial for ray-torus intersection, and a recommended bounding method based on a sphere followed by a vertical slab test. My implementation follows this material closely in its geometric formulation and in the use of the sphere-slab bounding volume.

Bounding Method

The project material proposes a two-stage bounding technique: a bounding sphere of radius $R + \max(a, b)$ to eliminate distant rays, followed by a vertical slab test using the planes $y = \pm b$ to reject rays that pass entirely above or below the torus. My implementation follows this method directly, as it is both simple to implement and highly effective at reducing the number of rays that reach the expensive implicit evaluation.

Quartic Root Finding

While the reference derives the full quartic polynomial and discusses analytic solution strategies, WGSL lacks double precision and robust polynomial solvers. Analytic quartic methods are therefore impractical in this environment. Instead, my implementation uses a sampled sign-change search followed by bisection. This numerical approach is stable under single precision and interacts well with the restricted ray interval provided by the sphere-slab test.

Normal Computation

The reference provides a specialized expression for computing the torus normal. In my implementation, normals are computed directly from the gradient ∇F of the implicit surface. This avoids additional algebraic handling, eliminates finite-difference approximations, and is consistent with how other implicit surfaces in the renderer are shaded.

Overall, the implementation follows the geometric principles of the project material while employing numerical methods that are better suited to the constraints of WGSL and a real-time path tracing context.

7 Conclusion

This project adds elliptical torus geometry to a WebGPU renderer by representing the surface implicitly, transforming rays into a local frame, and solving intersections numerically in WGSL. The implementation supports rotated instances, stable normals from the implicit gradient, and practical performance through conservative sphere-slab culling and a bounded bisection-based root finder.

References

[1] E. Cychosz. *Intersecting a Ray with an Elliptical Torus*. In *Ray Tracing Gems* (Chapter V.2). Apress.