Cosc363 Assignment - Ray Tracing - Max Shi (msh254)

Declaration
I declare that this assignment submission represents my own work (except for allowed material provided in the course), and that ideas or extracts from other sources are properly acknowledged in the report. I have not allowed anyone to copy my work with the intention of passing it off as their own work.
Name Max Shi
Student ID 36310991
Date 25/05/2025

Ray Tracer

The ray tracer developed for this assignment implements a comprehensive 3D rendering system, which features a ray-tracing algorithm which simulates various physical phenomena, such as reflection, refraction and shadows. Additionally, the ray tracer has many advanced rendering techniques such as stochastic sampling (soft shadows) and adaptive anti-aliasing.

Successes / Failures

A big success with the ray tracer produced is the ability to implement both advanced rendering techniques, and also complex objects, such as the torus. Specifically, a big success was implementing the tori, as it allowed for a more visually interesting scene. This was further accentuated by the introduction of the flat sphere, combining both complex objects and variations of simple objects.
Another big success was the implementation of Anti-Aliasing and Stochastic Sampling, as that allowed for a much more realistic scene. Without anti-aliasing, the scene looks very.. 'choppy' and the jagged edges were very visually unpleasant.

Regarding the failures, the most significant failure with the ray tracer was my inability to implement any Constructive Solid Geometry in the ray tracer. Although the idea was not as complex (personally I believed the tori calculations to be the most challenging), the actual implementation and details - such as changing the structure of the program (overriding SceneObject, and other project structure issues), proved to be too much..

Additional Features

1. Cylinder With Visible Cap
2. Torus
3. Refractive Sphere
4. Object-Space transform (Flat Sphere)
5. Anti-Aliasing
6. Textured Cylinder
7. Stochastic Sampling (Soft Shadows)

Cylinder + Cylinder Texturing

The ray tracer implements cylinders with clearly visible caps. The implementation handles the intersection calculations for both the cylindrical surface and the top/bottom caps with proper normal calculations for accurate lighting.

Mathematical Modelling

The cylinder is represented mathematically with the following three equations, modelling the Cylindrical Surface, Bottom Cap and Top Cap respectively.

$$(x - c_x)^2 + (z - c_z)^2 = r^2 \quad \text{where} \quad c_y \leq y \leq c_y + h$$
$$(x - c_x)^2 + (z - c_z)^2 \leq r^2 \quad \text{and} \quad y = c_y$$
$$(x - c_x)^2 + (z - c_z)^2 \leq r^2 \quad \text{and} \quad y = c_y + h$$

Ray-Cylinder Intersection is computed by solving a quadratic equation. For this, the ray is first parameterised as $\vec{p}(t) = \vec{p_0} + t\vec{d}$, then substituted in the cylinder equation results in $(p_x(t) - c_x)^2 + (p_z(t) - c_z)^2 = r^2$.
Where expanding, we obtain $(p_{0x} + t \cdot d_x - c_x)^2 + (p_{0z} + t \cdot d_z - c_z)^2 = r^2$.
And now ignoring the y-component, and rearranging in quadratic fashion, leads to $a \cdot t^2 + b \cdot t + c = 0$
Where:

$$a = d_x^2 + d_z^2 \qquad b = 2(e_x \cdot d_x + e_z \cdot d_z) \qquad c = e_x^2 + e_z^2 - r^2$$

Now given values for a, b and c, normal calculations for the discriminant can be obtained to determine intersection.
$\Delta < 0$: No intersection $\Delta = 0$: Ray is tangent to the cylinder $\Delta > 0$: Ray intersects the cylinder at two points.
For the caps, intersection is solved with the planes $y = c_y$ and $y = c_y + h$, then a subsequent check whether the intersection point is within the radius r.
Normal calculations are very straightforward, where the top and bottom caps return a vector in the corresponding direction (e.g (0,1,0), (0,-1,0)).
While the side surface normals are calculated such that the normal points outward horizontally from the central axis, resulting in the following equation:

$$\hat{n}_{\text{side}} = \frac{(x-c_x, 0, z-c_z)}{\sqrt{(x-c_x)^2 + (z-c_z)^2}}$$

Texture Mapping of Cylinder

Texture mapping the cylinder involves assigning 2D images onto 3D surfaces using **UV coordinates** (s, t) where:

$$s = horizontalTextureCoordinate \qquad t = verticalTextureCoordinate$$

To map a texture around the cylinder, we first compute the angle $theta$ around the y-axis:

$$\theta = \tan^{-1}\left(\frac{z-c_z}{x-c_x}\right)$$

Then, converting this angle into horizontal and vertical coordinates

$$s = \frac{\theta}{2\pi} \text{if } s < 0 \text{ then } s = s + 1 \qquad t = \frac{y - c_y}{h}$$

Torus

The ray tracer implements a mathematically complex torus shape with accurate intersection calculations, proper normal computations, and support for object-space transformations. The torus was one of the more challenging geometric primitives to implement due to the quartic equation that must be solved for ray-torus intersections.

Mathematical Modelling

A torus is defined by two parameters: the major radius R (distance from the center of the tube to the center of the torus) and the minor radius r (radius of the tube itself).

In standard position, the torus is centered at the origin with its axis aligned with the y-axis.

The implicit equation of a torus in standard position is: $(\sqrt{x^2 + z^2} - R)^2 + y^2 = r^2$

Which can be rewritten as: $(x^2 + y^2 + z^2 + R^2 - r^2)^2 - 4R^2(x^2 + z^2) = 0$

Ray-Torus Intersection

For a ray defined by origin $\vec{p_0}$ and direction $\vec{d}$, the intersection with the torus involves substituting the ray equation $p(t) = \vec{p_0} + t\vec{d}$ into the torus equation. This results in a quartic equation in t: $At^4 + Bt^3 + Ct^2 + Dt + E = 0$

Now, to derive the coefficients for the quartic equation, we must first use these following equations

$\vec{E} = (E_x, E_y, E_z) = \vec{d}$ (ray direction)

$\vec{D} = (D_x, D_y, D_z) = \vec{p_0}$ (ray origin)

A = major radius (R)

B = minor radius (r)

$$H = 8A^2(D_xE_x + D_zE_z)$$
$$I = 4A^2(D_x^2 + D_z^2)$$
$$J = E_x^2 + E_y^2 + E_z^2$$
$$K = 2(D_xE_x + D_yE_y + D_zE_z)$$
$$L = D_x^2 + D_y^2 + D_z^2 + A^2 - B^2$$

$$G = 4A^2(E_x^2 + E_z^2)$$

Resulting in our Coefficients,

$$A = J^2 \qquad B = 2JK \qquad C = 2JL + K^2 - G \qquad D = 2KL - H \qquad E = L^2 - I$$

Note: For more detail, see this and also the appendix 2.1 will have a more detailed computation of how these intermediate equations were obtained, and their relevance.

Regardless, now that we have our quartic equation, we can now solve the Quartic to find intersections.

Durand-Kerner Method for Quartic Equation Solving

The Durand-Kerner method is an iterative numerical technique for finding all roots of a polynomial simultaneously. Firstly, we initialize 4 complex roots with distinct values:

$$z_1^{(0)} = 0.5 + 0.5i \qquad z_2^{(0)} = -0.5 + 0.5i \qquad z_3^{(0)} = -0.5 - 0.5i \qquad z_4^{(0)} = 0.5 - 0.5i$$

Then, update each root using the formula:

$$z_i^{(k+1)} = z_i^{(k)} - \frac{P(z_i^{(k)})}{\prod_{j \neq i}(z_i^{(k)} - z_j^{(k)})} \text{ Where } P(z) \text{ is the polynomial evaluated at } z.$$ (see : this and that)

We then continue until convergence (this is when the difference between iterations is below a set tolerance)

Then, we can extract the **real** roots, as the intersection points.

Performance Calculations (Bounding Box)

Due to the intensive quartic equation solution, there is a bounding box check before the calculations,

$\|\vec{p_0} - \vec{c}\|^2 \leq (R + r)^2$ Where $\vec{c}$ is the center of the torus and $(R + r)$ is the radius of the bounding sphere.

Normal Calculations

Luckily, normal calculations are not so intensive as finding the intersection for the torus..

The normal at a point $\vec{p}$ on the torus surface is calculated as follows:

1. Find the closest point $\vec{Q}$ on the circular axis of the torus:

$$\alpha = \frac{R}{\sqrt{p_x^2 + p_z^2}} \qquad \vec{Q} = (\alpha p_x, 0, \alpha p_z)$$

2. The normal vector is the direction from $\vec{Q}$ to $\vec{p}$:

$$\vec{N} = \vec{p} - \vec{Q} \qquad \hat{N} = \frac{\vec{N}}{\|\vec{N}\|}$$

Refractive Sphere

The ray tracer implements refraction of light through an object. A green sphere on the right side of the scene demonstrates this feature with a refractive index of 1.02.

Refraction is dictated via Snell's Law, which relates θ_1 angle of incidence, θ_2 angle of refraction, as well as n1 and n2 (refractive index of medium 1 and medium 2)

$\frac{\sin\theta_1}{\sin\theta_2} = \frac{n_2}{n_1}$. In vector format, refracted direction is calculated via: $\vec{t} = \frac{n_1}{n_2}\vec{i} + \left(\frac{n_1}{n_2}\cos\theta_1 - \cos\theta_2\right)\vec{n}$

where: $\vec{t}$ is the refracted direction $\vec{i}$ is the incident direction $\vec{n}$ is the surface normal $\cos\theta_1 = -\vec{i}\cdot\vec{n}$ and $\cos\theta_2 = \sqrt{1 - \left(\frac{n_1}{n_2}\right)^2 (1 - \cos^2\theta_1)}$

Object-Space Transformations

Object-space transformations are represented using 4×4 matrices:

Translation by vector (t_x, t_y, t_z) :

$$T = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling by factors (s_x, s_y, s_z) :

$$S = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation around the X-axis by angle theta

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation around the Y-axis by angle theta

$$R_y = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation around the Z-axis by angle theta

$$R_z = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Multiple transformations can be applied by combining matrices (multiplying matrices). The order of multiplication is important, as matrix multiplication is not commutative.
An example of this is best by looking in the code directly..

```
void scale(const glm::vec3& s) {
    glm::mat4 S = glm::scale(glm::mat4(1.0f), s);
    transform_ = S * transform_;
    invTransform_ = glm::inverse(transform_);
    normalTransform_ = glm::transpose(invTransform_); }
```

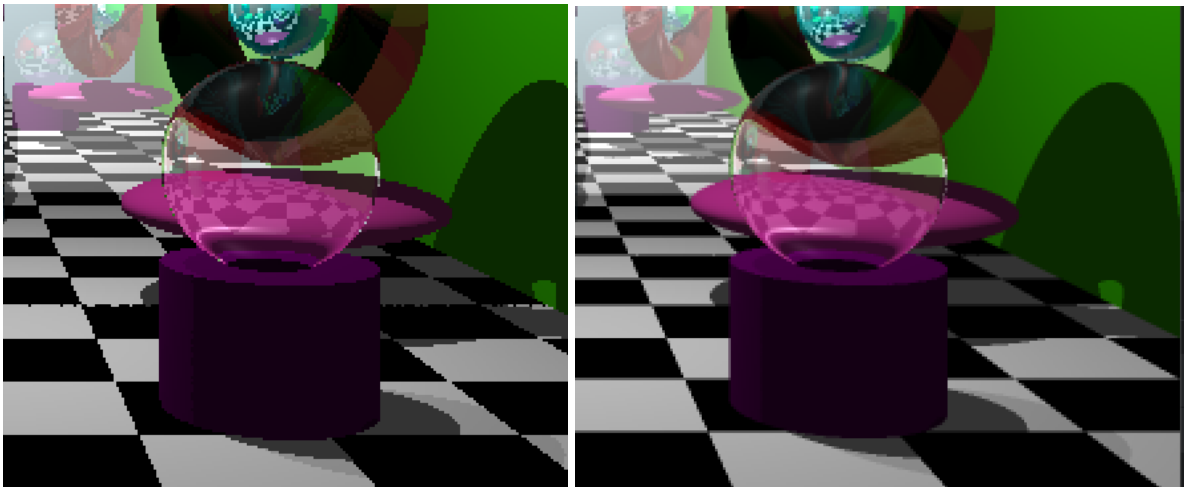
Key things to note: scale() creates a scaling matrix and applies it to the current transformation. Additionally, the inverse (for ray transformation) and normal transformation (for lighting) are also updated.

Adaptive Anti-Aliasing

Adaptive anti-aliasing works by recursively subdividing the pixel into four sub-regions and comparing the colours. If the color difference between samples exceeds a threshold, then we further subdivide the quadrants. We then average the colors of all samples to determine the final pixel colour
The color difference between two samples is calculated as the Euclidean distance in RGB space:

$$\text{diff} = \sqrt{(r_1 - r_2)^2 + (g_1 - g_2)^2 + (b_1 - b_2)^2}$$

Above (left) : No Anti-Aliasing, Above(right) : Adaptive Anti-Aliasing, Adaptive Depth 2 | both 500 divs



Stochastic Sampling (Soft Shadows) -> PCSS method

Soft shadows are generated by defining an area light source with a radius, sampling random points on the light source, for each sample point, cast a shadow ray to determine visibility, and then average the results to determine shadow intensity.
Area Light Source

An area light is modelled as a disk with center $\vec{L_c}$ and radius r_L. Points on the disk may be parameterized as: $\vec{L}(s,t) = \vec{L_c} + s\vec{u} + t\vec{v}$ Where: $\vec{u}$ and $\vec{v}$ are orthogonal unit vectors in the plane of the disk and s and t are parameters such that $s^2 + t^2 \leq r_L^2$.
Now that we have the light source modelled, we construct an orthonormal basis $\vec{u}, \vec{v}$ around the light direction to sample random positions across the light surface.

Random Sampling

Now for each SAMPLES_PER_PIXEL, we generate a point on the light source

```
float rx = (dist(rng) * 2.0f - 1.0f) * LIGHT_RADIUS;
float ry = (dist(rng) * 2.0f - 1.0f) * LIGHT_RADIUS;
glm::vec3 randomLightPos = lightPos + rx * u + ry * v;
```

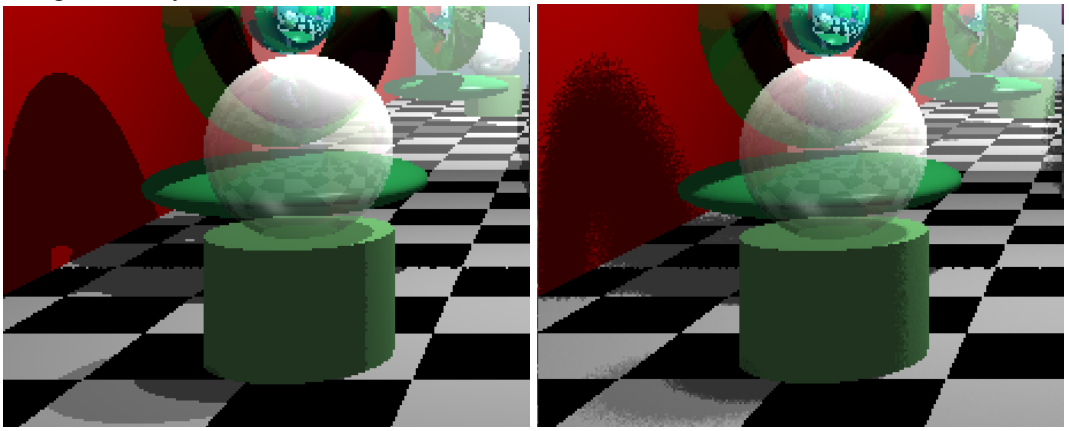
Which produces distributed rays aimed towards random parts of the light source.

For each sample point $\vec{L_i}$ on the light source, we cast a shadow ray from the intersection point $\vec{P}$ to $\vec{L_i}$. The shadow factor is calculated as

$SF = 1 - \frac{N_{shadow}}{N_{total}}$ Where: N_{shadow} is the number of samples that are occluded & N_{total} is the total number of samples.

Hence we obtain the final color with the soft shadows:

$C = \frac{\sum_{i=1}^{N_{visible}} C_i}{N_{total}} \cdot A \cdot I \cdot SF$ where Ci is the color contribution from visible light samples, A is the attenuation factor, I is the light intensity, and SF is the shadow factor.



Above (left) : No softer shadows, Above(right) : Soft Shadows 8 Samples per pixel, light rad = 2.

Estimated Time to load Ray Tracer (Tested on Windows: see below for specs))

Number Of Divisions	No Anti-Aliasing, No Softer Shadows	Anti-Aliasing (Depth 2), No Softer Shadows	No Anti-Aliasing, Softer Shadows (4 Samples per pixel)	Anti-Aliasing **(Depth 1)**, Softer Shadows (4 samples per pixel)
250	5.57s	50s	10s	25s
500	12s	3m 20s	27s	2m
1000	50s	14m	1m 55s	8m

The highlighted value is the default ray tracer settings.

Build Process / Environment

This ray tracer was developed in Windows/Linux (Debian) using the CLion IDE. System specs for Windows were 64GB DDR4 Ram (@3600 MHz), AMD Ryzen 7 5800X 8-Core Processor (@3.80 GHz).
This project can be compiled using the CSSE lab machines via VSCodium.

- In VSCode install the CMake and CMake Tools extension
- Create a directory and move all project files including CMakeLists.txt into the directory
- Open the directory in VSCodium
- In the CMake tab, go to Project Status -> Configure, and press Select A Kit button (pencil icon)
- Select linux gcc as the compiler, doing so should create necessary files in a build directory
- Go to Project Outline, and set the build and launch targets if necessary
- Go to Project Status -> Launch, and press the run button (play icon). The project should then compile and run the executable

References, AI & More.

GenAI was used in multiple ways. These include; Breaking down hard topics and explaining relevance to ray tracing or openGL more generally, source finding. GenAI used : Claude Sonnet 3.7 (thinking), and OpenAI ChatGPT 4.5 (Research Preview). GenAI was *NOT* used for code generation, although some explanations surrounding topics did contain code. It is to be noted GenAI was *NOT* the biggest helper for this project, but rather good quality sources.
A full list of references is below:

Torus:
Line-Torus Intersection for Ray Tracing: Alternative Formulations
Efficient Ray-Torus Intersection for Ray Tracing
Durand–Kerner method - Wikipedia
Ray-tracing quartic surfaces - Stack Overflow

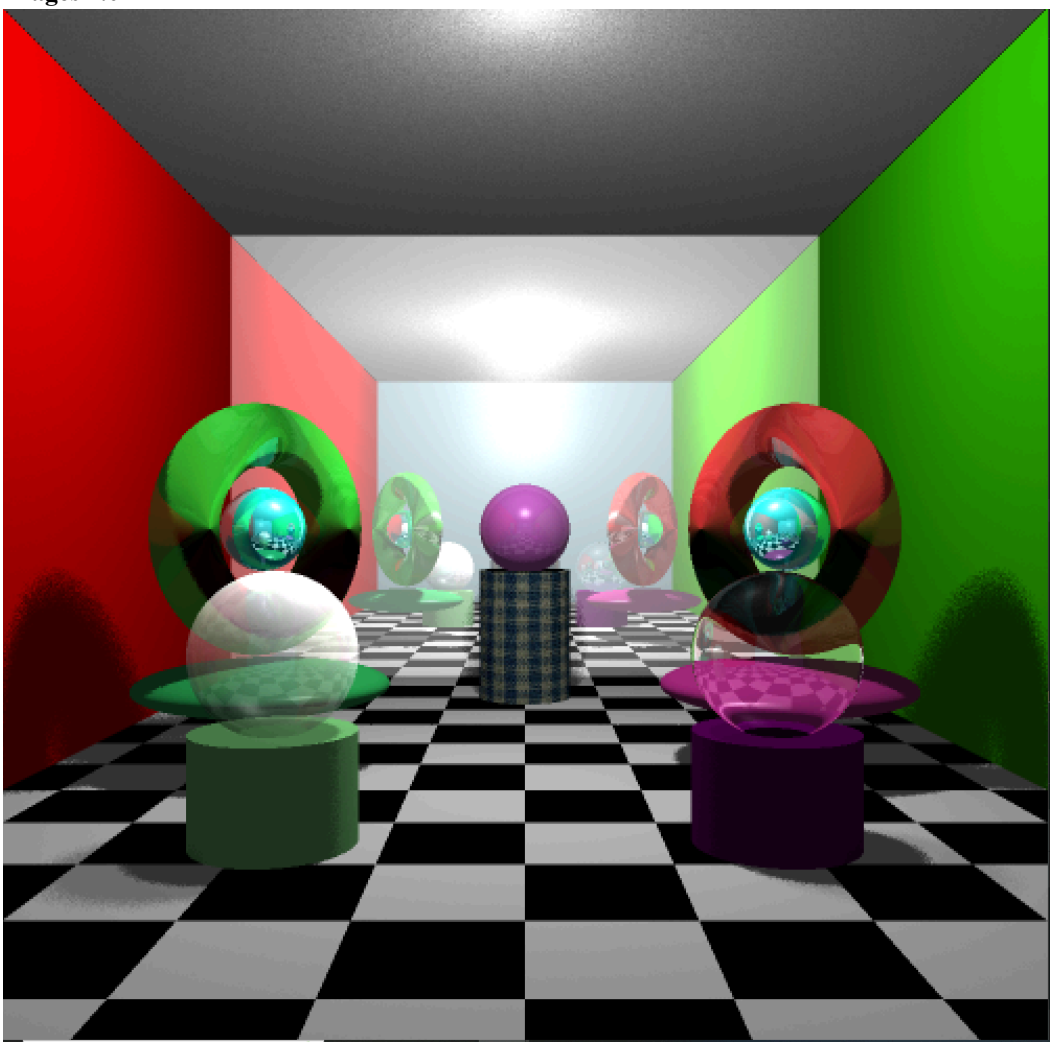
Soft Shadows:
Percentage-Closer Soft Shadows | NVIDIA
Variance Soft Shadow Mapping
Stochastic Soft Shadow Mapping
Soft Shadows - PCF & Random Sampling // OpenGL Tutorial #41

Anti-Aliasing
Anti-aliasing - Wikipedia
Supersampling - Wikipedia
Ray Tracing With Adaptive Supersampling in Object Space
OpenGL Tutorial 22 - Anti-Aliasing (MSAA)

Misc
9.7 - Light Attenuation — LearnWebGL
std::mt19937 Class in C++ | GeeksforGeeks
What is the proper way of seeding std::mt19937 with std::chrono::high_resolution_clock inside a class? - Stack Overflow

Fabric Pattern: Polyhaven
Fabric Pattern 07 Texture • Poly Haven

Appendix
Images 1.0



Above: Ray Tracer Output with Anti-Aliasing 1, numdiv 500, and softer shadows.

Code 2.0

2.1: Computation of Intermediate Values for Torus Calculations

So, how are the intermediate values calculated for the torus?

A torus centered at the origin and lying in the XY-plane has this implicit equation:

$(x^2 + y^2 + z^2 + R^2 - r^2)^2 - 4R^2(x^2 + y^2) = 0$

- R: major radius (distance from origin to tube center)
- r: minor radius (tube radius)

Any point (x, y, z) that satisfies this equation lies on the surface of the torus.

Next, we define the ray as:

$\vec{p}(t) = \vec{D} + t\vec{E}$

Where:

- $\vec{D} = (D_x, D_y, D_z)$: ray origin
- $\vec{E} = (E_x, E_y, E_z)$: ray direction

So we can define the components of the ray as

$x(t) = D_x + tE_x, \quad y(t) = D_y + tE_y, \quad z(t) = D_z + tE_z$

Lets substitute now. We know

$Q(t) = (x(t)^2 + y(t)^2 + z(t)^2 + R^2 - r^2)^2 - 4R^2(x(t)^2 + y(t)^2)$ (from above)

Now we expand:

$x(t)^2 + y(t)^2 + z(t)^2 = (\vec{D} + t\vec{E}) \cdot (\vec{D} + t\vec{E}) = D^2 + 2t(\vec{D} \cdot \vec{E}) + t^2E^2$

Where:

$D^2 = D_x^2 + D_y^2 + D_z^2$

$E^2 = E_x^2 + E_y^2 + E_z^2$

$\vec{D} \cdot \vec{E} = D_xE_x + D_yE_y + D_zE_z$

So:

$S(t) = D^2 + 2t(\vec{D} \cdot \vec{E}) + t^2E^2$

Then:

$Q(t) = (S(t) + R^2 - r^2)^2 - 4R^2(x(t)^2 + y(t)^2)$

Giving a full result of:

$x(t)^2 + y(t)^2 = D_x^2 + 2tD_xE_x + t^2E_x^2 + D_y^2 + 2tD_yE_y + t^2E_y^2$

$= D_x^2 + D_y^2 + 2t(D_xE_x + D_yE_y) + t^2(E_x^2 + E_y^2)$

We can now format this into a quartic form:

$(a_2t^2 + a_1t + a_0)^2$ subtracted by

$4R^2(b_2t^2 + b_1t + b_0)$

leads to : $Q(t) = At^4 + Bt^3 + Ct^2 + Dt + E = 0$

With this quartic's coefficients dependent on:

$D^2,\ E^2,\ \vec{D} \cdot \vec{E}$

$D_x^2 + D_y^2,\ D_xE_x + D_yE_y,\ E_x^2 + E_y^2$

$R,\ r$

And so:

$G = 4R^2(E_x^2 + E_y^2)$

$H = 8R^2(D_xE_x + D_yE_y)$

$I = 4R^2(D_x^2 + D_y^2)$

$J = E^2 = E_x^2 + E_y^2 + E_z^2$

$K = 2(\vec{D} \cdot \vec{E})$

$L = D^2 + R^2 - r^2$

Therefore:

$J^2t^4 + 2JKt^3 + (2JL + K^2 - G)t^2 + (2KL - H)t + (L^2 - I) = 0$