ERIT – A Collection of Efficient and Reliable Intersection Tests

Martin Held*
Institut für Computerwissenschaften
Universität Salzburg
A–5020 Salzburg, Austria

Abstract

We describe ERIT, a collection of C routines for efficiently and reliably handling intersection queries between pairs of primitive objects in 3D. ERIT supports intersection queries between the following pairs of primitives: triangle/line-segment, triangle/triangle, sphere/line-segment, sphere/triangle, cylinder/line-segment, cylinder/triangle, cylinder/sphere, cone/line-segment, cone/triangle, toroid/line-segment, toroid/triangle, and sphere/sphere. All intersection routines are based on standard 'epsilon-based' floating-point arithmetic. Practical tests have proved that ERIT's routines are efficient and reliable, and we provide performance statistics for three widely-used hardware platforms. The source code for ERIT is available from the author.

1 Introduction

1.1 Motivation and Related Work

Checking whether two primitives (e.g., two triangles) intersect in three dimensions (3D) is common in graphics. An implementation should be efficient and reliable. Handling all degenerate cases – e.g., a vertex of one triangle lies on an edge of the second triangle – is somewhat tricky in 2D and gets even less straightforward in 3D. Algorithms for intersection detection also get more sophisticated when curved primitives (e.g., cylinders) are involved.

Surprisingly few algorithms and even fewer ready-to-use implementations are available. Most published results address ray-object intersections for ray-tracing. See [Wat89, FvDFH90, Gla90, Arv91, Gla92, Kir92, Hec94, Pae95]. An article on ray-triangle intersection appeared in an earlier issue of this journal [MT97].

1.2 Survey of ERIT

We present algorithms and code, a package called ERIT, for 3D intersection testing on line segments, triangles, cylinders, cones, and convex and concave toroidal[1] solids. Line segments and

*Email: held@cosy.sbg.ac.at. Part of this work was carried out while visiting SUNY Stony Brook.

[1] A toroidal solid is generated by rotating a circular arc around an axis.

triangles can be tested for intersection with any of these primitives (except for line-segment/line-segment tests). We also handle intersection tests between spheres and cylinders. In addition, ERIT offers a routine for computing the point of intersection between a triangle and a line segment, and contains a few 2D intersection tests.

ERIT is restricted to curved primitives that can be described as solids of revolution generated by rotating a line segment or a circular arc (less or equal to a half circle) around an axis of rotation. This implies that the 'bottom' and 'top' disks of our primitives lie in planes that are normal to the axis of rotation. ERIT cannot handle objects with genus one or higher, such as a torus.

ERIT is written in ANSI C, and has been compiled and tested (using GNU's `gcc`) on Sun SPARCstations, on Silicon Graphics Indigos, and on PCs running LINUX. The source code for ERIT is available upon sending an e-mail request to the author (`held@cosy.sbg.ac.at`).

ERIT operates on the C data type `double`. Comparisons to zero are carried out relative to some threshold value ϵ. We have tried to strike a balance between achieving a good robustness and a high speed[2] of the code. Still, numerical imprecisions may cause the code to miss grazing intersections, or to classify two primitives as intersecting or touching which, in reality, are separated by a tiny distance. Section 3.1 further addresses numerical robustness in ERIT.

The remainder of this paper is organized as follows: Section 2 presents algorithmic details of the intersection tests. In Section 3, we discuss implementational details and present cpu-time statistics.

2 Algorithms for Intersection Testing

This section describes our intersection routines. While there are no new insights into geometric computing, novices may find ideas for crafting routines of their own.

2.1 Notation and Basics

Notation: We denote points (in 2D or 3D) with upper-case italics letters, e.g., A; vectors (in 2D or 3D) with Euler Fraktur letters, e.g., $\mathfrak{v}$. The vector pointing from the origin to a point A is denoted by $\mathfrak{a}$. The coordinates of a vector are obtained by subscripting it, e.g., $\mathfrak{v}_x$ is the x-coordinate of $\mathfrak{v}$. For vectors pointing from one named point to another named point we often include those names into the name of the vector: e.g., $\mathfrak{ab}$ is the vector pointing from A to B, $\mathfrak{ab} = \mathfrak{b} - \mathfrak{a}$, and $\mathfrak{ab}_x$ is its x-component. The line segment between A and B is denoted by $\overline{AB}$. Upper-case calligraphic letters denote primitives, e.g., $\mathcal{T}$ stands for a triangle. Bold-face upper-case letters, e.g., $\mathbf{M}$, denote matrices. Lower-case and Greek symbols denote scalars.

A dot between two vectors, as in $\mathfrak{a} \cdot \mathfrak{b}$, denotes their dot product. A dot[3] between a scalar and a vector, as in $\lambda \cdot \mathfrak{a}$, denotes a scalar multiplication of the vector. The vector product is denoted $\mathfrak{a} \times \mathfrak{b}$. The length (norm) of a vector is denoted $|\mathfrak{a}|$. For three 3D vectors $\mathfrak{u}, \mathfrak{v}, \mathfrak{w}$, we define

$$\begin{aligned} \det(\mathfrak{u},\mathfrak{v},\mathfrak{w}) &:= \begin{vmatrix} \mathfrak{u}_x & \mathfrak{u}_y & \mathfrak{u}_z \\ \mathfrak{v}_x & \mathfrak{v}_y & \mathfrak{v}_z \\ \mathfrak{w}_x & \mathfrak{w}_y & \mathfrak{w}_z \end{vmatrix} \\ &= \mathfrak{u}_x \cdot (\mathfrak{v}_y \cdot \mathfrak{w}_z - \mathfrak{v}_z \cdot \mathfrak{w}_y) - \mathfrak{u}_y \cdot (\mathfrak{v}_x \cdot \mathfrak{w}_z - \mathfrak{v}_z \cdot \mathfrak{w}_x) + \mathfrak{u}_z \cdot (\mathfrak{v}_x \cdot \mathfrak{w}_y - \mathfrak{v}_y \cdot \mathfrak{w}_x). \end{aligned}$$

[2] Typically, robustness and speed are contradictory goals for intersection algorithms.

[3] For visual clarity, we sometimes also place dots between two scalars.

It is known that $\det(\mathfrak{u}, \mathfrak{v}, \mathfrak{w})$ is six times the oriented volume of the pyramid (tetrahedron) with apex at the origin and vertices $\mathfrak{u}, \mathfrak{v}, \mathfrak{w}$.

For a 2D vector $\mathfrak{a}$, consider the map $i : \mathbb{R}^2 \to \mathbb{R}^3$ that takes $(\mathfrak{a}_x, \mathfrak{a}_y)$ to $(\mathfrak{a}_x, \mathfrak{a}_y, 1)$. Then, for three 2D vectors $\mathfrak{a}, \mathfrak{b}, \mathfrak{p}$, we define $\det(\mathfrak{a}, \mathfrak{b}, \mathfrak{p}) := \det(i(\mathfrak{a}), i(\mathfrak{b}), i(\mathfrak{p}))$. Using simple algebraic manipulations, we obtain $\det(\mathfrak{a}, \mathfrak{b}, \mathfrak{p}) = (\mathfrak{p}_x - \mathfrak{a}_x) \cdot (\mathfrak{a}_y - \mathfrak{b}_y) + (\mathfrak{a}_y - \mathfrak{p}_y) \cdot (\mathfrak{a}_x - \mathfrak{b}_x)$. The number $\det(\mathfrak{a}, \mathfrak{b}, \mathfrak{p})$ is twice the oriented area of the triangle with vertices A, B, P. It is positive iff A, B, P are given in counter-clockwise order, zero iff A, B, P are collinear, and negative otherwise. In other words, P is strictly left of the oriented line through A, B iff $\det(\mathfrak{a}, \mathfrak{b}, \mathfrak{p}) > 0$.

Basics: When designing an intersection test for two objects, the overall goal is to check whether they intersect in such a way that

1. a witness of disjointness is quickly found in case that they do not intersect,
2. computations carried out for (1) can be re-used for confirming the existence of an intersection, and such that
3. the number of computationally expensive operations (such as the computation of square roots) is minimized.

We note that a bounded convex primitive $\mathcal{P}_1$ intersects another bounded and strictly convex primitive $\mathcal{P}_2$ iff either some point on the boundary of $\mathcal{P}_1$ lies inside $\mathcal{P}_2$, or if the point of $\mathcal{P}_1$ closest to $\mathcal{P}_2$ (or to some reference point of $\mathcal{P}_2$) is inside $\mathcal{P}_2$. In particular, a line segment does not intersect a convex solid if one end point is in the exterior of the solid and if the ray from this end point to the other end point is directed away from the solid. Also, two primitives cannot intersect if they are on different sides of a separating plane.

For testing a triangle $\mathcal{T}$ for intersection with one of our 3D solid primitives $\mathcal{S}$ we will use one of the following two generic algorithms:

Algorithm TRI-I: We first test whether any of the three edges of $\mathcal{T}$ intersects $\mathcal{S}$. This includes checking whether an edge of $\mathcal{T}$ is completely contained in $\mathcal{S}$, which can be carried out by checking whether one of the end points of the edge is contained in $\mathcal{S}$. If the edges of $\mathcal{T}$ do not intersect $\mathcal{S}$, then the intersection of the supporting plane of $\mathcal{T}$ with $\mathcal{S}$ must be completely in the interior or the exterior of $\mathcal{T}$. Whether it is in the interior is checked by intersecting $\mathcal{T}$ with a set of line segments. (What constitutes a proper set of line segments depends on the solid.)

Algorithm TRI-II: We first select two half-spaces that contain $\mathcal{S}$ and trim $\mathcal{T}$ to them. This yields a polygon $\mathcal{P} \subset \mathcal{T}$ with up to five vertices. (Typically, we will use planes through the bottom and top disks of our solids as bounding planes of the half-spaces.) Then we proceed similar to Algorithm TRI-I: We test the edges of $\mathcal{P}$ for intersection with $\mathcal{S}$, and if no intersection is found, we test $\mathcal{T}$ for intersection with a proper set of line segments.

For both algorithms the bulk of the work lies in the corresponding solid/line-segment tests, which we will describe in detail for each of our solids. Whether TRI-I or TRI-II is faster largely depends on how efficiently $\mathcal{T}$ can be trimmed to the half-spaces, and how simple the individual solid/line-segment tests are. We will report which algorithm turned out to be faster for each intersection test studied.

Whenever possible, we apply dimension reduction to perform computations in 2D rather than in 3D. Dimension reduction is possible if two primitives intersect iff their projections onto a plane (or intersections with a plane) intersect. For instance, a line intersects a sphere iff it intersects the circle given by the intersection of the sphere with a plane through the line and the center of the circle.

When applying dimension reduction the actual computations are carried out in one of the coordinate planes[4]. To minimize numerical problems, we project the entities in the 3D plane onto a coordinate plane by dropping the coordinate of the normal vector of the 3D plane whose absolute value is largest.

We stress that the intersection tests in this paper should be used in conjunction with a computationally inexpensive pre-test to weed out pairs of primitives that cannot intersect. A typical pre-test is to check whether the (axis-aligned) bounding boxes of the two primitives overlap. The primitives cannot intersect if their bounding boxes do not. If available, one may also want to use other bounding volumes such as spheres [Hub96] or so-called k-dops [KK86, KHM+98].

2.2 Triangle/Line-Segment and Triangle/Triangle Tests

Triangle/Line-Segment Test: In 3D consider a triangle $\mathcal{T}$ with vertices A, B, C, and a line segment $\mathcal{L}$ with vertices P, Q, cf. Fig. 1. As a first step, we check whether the end points of $\mathcal{L}$ lie on the same side of the supporting plane $\mathcal{P}$ of $\mathcal{T}$. Let $\mathfrak{n} := \mathfrak{ab} \times \mathfrak{ac}$ be the normal vector[5] of $\mathcal{P}$. Then P and Q lie on the same side of $\mathcal{P}$ iff the signs of $\mathfrak{n} \cdot \mathfrak{ap}$ and $\mathfrak{n} \cdot \mathfrak{aq}$ are identical. The point P lies on $\mathcal{P}$ if $\mathfrak{n} \cdot \mathfrak{ap} = 0$. In this degenerate case we apply dimension reduction and do a 2D point-in-triangle test if Q does not lie on $\mathcal{P}$, i.e., if $\mathfrak{n} \cdot \mathfrak{aq} \neq 0$. If $\mathfrak{n} \cdot \mathfrak{ap} = 0$ and $\mathfrak{n} \cdot \mathfrak{aq} = 0$ then both P and Q are coplanar with $\mathcal{T}$, and we do a 2D triangle/line-segment test.

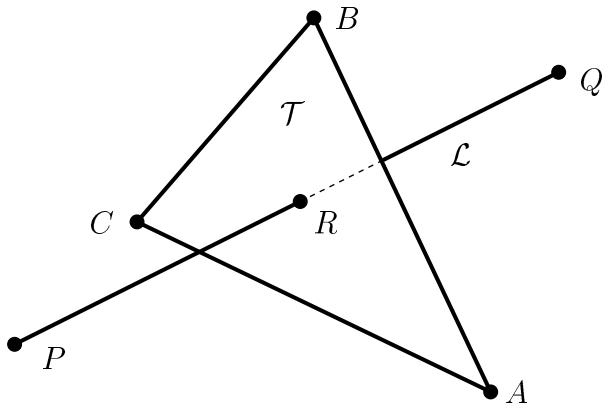


Figure 1: Triangle/Line-Segment Intersection.

If P and Q lie on different sides of $\mathcal{P}$ then we compute the intersection of $\mathcal{L}$ with $\mathcal{P}$. Consider the parameterization $\mathfrak{p} + t \cdot \mathfrak{pq}$, with $0 \leq t \leq 1$. The t-value of the point of intersection is given

[4] That is, in one of the following three planes: $x = 0$, $y = 0$, or $z = 0$.

[5] A user of ERIT can either let the code compute a normal vector, or give it as input data to ERIT's subroutines. For instance, in order to increase the robustness of the normal computation for long skinny triangles, a user might want to compute cross products on the basis of the largest vertex angle.

by $t_r = (\mathfrak{n} \cdot \mathfrak{a} - \mathfrak{n} \cdot \mathfrak{p})/\mathfrak{n} \cdot \mathfrak{pq}$. Since $\mathfrak{n} \cdot \mathfrak{ap}$ and $\mathfrak{n} \cdot \mathfrak{aq}$ have already been computed, we may rewrite[6] this as $t_r = \mathfrak{n} \cdot \mathfrak{ap}/(\mathfrak{n} \cdot \mathfrak{ap} - \mathfrak{n} \cdot \mathfrak{aq})$. The approach outlined has been encoded in the routine TriEdge3D[7].

Let $\mathfrak{r} := \mathfrak{p} + t_r \cdot \mathfrak{pq}$. It remains to check whether R lies within $\mathcal{T}$. We use dimension reduction and project both $\mathcal{T}$ and R onto a coordinate plane. Let A', B', C', R' be the projected points, and assume that A', B', C' are arranged in counter-clockwise order. The resulting 2D point-in-triangle test can be carried out by checking the signs of the determinants $\det(\mathfrak{a}', \mathfrak{b}', \mathfrak{p}')$, $\det(\mathfrak{b}', \mathfrak{c}', \mathfrak{p}')$, and $\det(\mathfrak{c}', \mathfrak{a}', \mathfrak{p}')$. This approach has been encoded in TriPnt2D. Alternatively, one can express R' in barycentric coordinates with respect to A', B', C': the triangle contains R' iff there exist $0 \leq \lambda, \mu \leq 1$ such that $\mathfrak{r}' = \lambda \cdot \mathfrak{a}' + \mu \cdot \mathfrak{b}' + (1 - \lambda - \mu) \cdot \mathfrak{c}'$, and $1 - \lambda - \mu \geq 0$. However, computing barycentric coordinates turned out to be slower than doing the determinant-based test.

Note that the orientation of A', B', C' can be deduced from $\mathfrak{n}$ if the normal vector is computed as $\mathfrak{n} := \mathfrak{ab} \times \mathfrak{ac}$. If one drops the z-coordinate, and $\mathfrak{n}_z > 0$, then A', B', C' are arranged in counter-clockwise order in the x, y-system. Similar for $\mathfrak{n}_y > 0$ and the z, x-system, and $\mathfrak{n}_x > 0$ and the y, z-system.

After checking whether P and Q lie on opposite sides of $\mathcal{P}$, an alternative approach is to check whether Q lies within the infinite triangular pyramid whose apex is P and whose cross-section with $\mathcal{P}$ is $\mathcal{T}$. This test can be carried out by computing determinants. However, this approach turned out to be slower than checking whether R lies in $\mathcal{T}$, as done by TriEdge3D.

Triangle/Triangle Test: Consider $\mathcal{T}$, with vertices A, B, C, and a second triangle with vertices P, Q, R, cf. Fig. 2. TriTri3D first checks whether P, Q, R lie on different sides of the supporting plane $\mathcal{P}$ of $\mathcal{T}$. Then, the intersection of the second triangle with $\mathcal{P}$ is computed. This yields a line segment $\mathcal{L}$ which is coplanar with $\mathcal{T}$. Clearly, $\mathcal{T}$ intersects the second triangle iff $\mathcal{T}$ intersects $\mathcal{L}$.

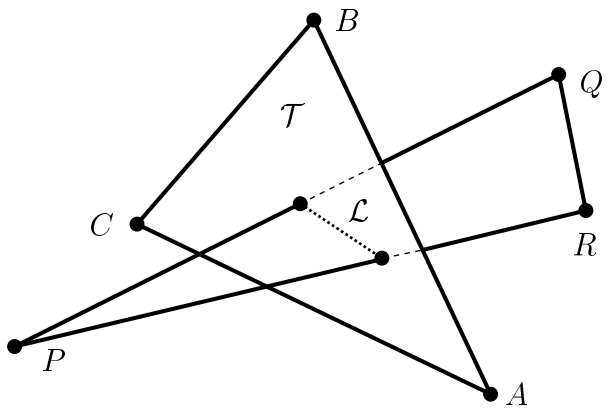


Figure 2: Triangle/Triangle Intersection.

After applying dimension reduction, the resulting 2D triangle/line-segment test is solved by TriEdge2D as follows: we first determine on which sides of the triangle's edges the end points of the line segment lie. If both lie on the exterior side of an edge then no intersection exists. If

[6] Here we face a typical trade-off between robustness and speed: the former expression for t_r is more robust but takes slightly longer to evaluate.

[7] Within ERIT, we use 'edge' as a short-hand for 'line segment'.

either end point lies on the interior sides of all edges then an intersection exists. If neither of these two simple cases applies, then an intersection exists iff any two of the three vertices of the triangle lie on different sides of the line segment. All tests for sidedness are carried out by computing the signs of determinants.

2.3 Sphere/Line-Segment and Sphere/Triangle Tests

Sphere/Line-Segment Test: Consider a sphere $\mathcal{S}$ with center P and radius ρ, and a line segment $\mathcal{L}$ with end points A and B. Let C be the normal projection of P onto the supporting line of $\mathcal{L}$, cf. Fig. 3. Evidently, $\mathcal{S}$ intersects $\mathcal{L}$ iff (a) A or B lie within $\mathcal{S}$ or (b) the distance δ between P and C is less than ρ and C lies between A and B. Consider the situation where $\delta_1 := |\mathfrak{ap}| > \rho$, and let $\delta_2 := |\mathfrak{ac}|$. Note that $\mathcal{S}$ and $\mathcal{L}$ cannot intersect if $\mathfrak{ab} \cdot \mathfrak{ap} < 0$ because the oriented line segment 'points away' from the sphere in this case. Otherwise, we compute δ_2 as $\delta_2^2 = (\mathfrak{ab} \cdot \mathfrak{ap})^2/|\mathfrak{ab}|^2$. Furthermore, observe that C lies between A and B iff $\delta_2^2 < |\mathfrak{ab}|^2$. Thus, if C lies between A and B then $\mathcal{S}$ intersects $\mathcal{L}$ iff $\delta^2 = \delta_1^2 - \delta_2^2 \leq \rho^2$. If C does not lie between A and B then $\mathcal{S}$ intersects $\mathcal{L}$ iff $|\mathfrak{bp}|^2 < \rho^2$. This approach has been encoded in SphEdge3D. Attempts to re-arrange the decision steps of this algorithm did not seem to yield a faster test.

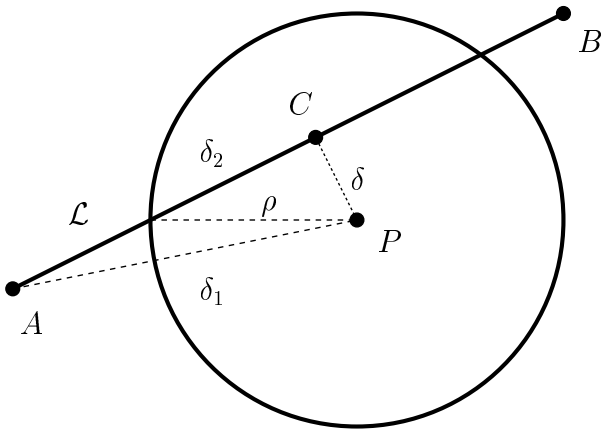


Figure 3: Sphere/Line-Segment Intersection.

Sphere/Triangle Test: For checking whether $\mathcal{S}$ intersects a triangle $\mathcal{T}$ with vertices A, B, C, the routine SphTri3D first checks whether any edge of $\mathcal{T}$ intersects $\mathcal{S}$. If no such boundary intersection exists then it checks whether the distance between P and its normal projection Q onto the supporting plane $\mathcal{P}$ of $\mathcal{T}$ is less than ρ, and whether Q lies within $\mathcal{T}$. Obviously, there exists t_q such that $\mathfrak{q} = \mathfrak{p} + t_q \cdot \mathfrak{n}$, where $\mathfrak{n}$ is the normal vector of $\mathcal{P}$. We get $t_q = \mathfrak{n} \cdot \mathfrak{pa}/|\mathfrak{n}|^2$, which yields $|\mathfrak{pq}|$ as $|\mathfrak{pq}|^2 = (\mathfrak{n} \cdot \mathfrak{pa})^2/|\mathfrak{n}|^2$. After dimension reduction, a call to TriPnt2D suffices to finish this intersection test.

2.4 Cylinder/Line-Segment and Cylinder/Triangle Tests

Cylinder/Line-Segment Test: We start by defining the type of cylinder ERIT can handle: A right (circular) cylinder is the set of points between two parallel planes, $\mathcal{P}', \mathcal{P}''$, whose distances

from a line $\mathcal{L}_c$ normal to $\mathcal{P}', \mathcal{P}''$ are no greater than some given positive radius ρ. Thus, we can conveniently characterize a cylinder by points P, Q which correspond to the intersections of $\mathcal{L}_c$ with $\mathcal{P}', \mathcal{P}''$, and by a radius ρ. If $\mathcal{L}_c$ is parallel to the z-axis and if $\mathfrak{p}_z \leq \mathfrak{q}_z$ then we call the cylinder an 'upright' circular cylinder.

It is not surprising that testing an upright circular cylinder for intersection with another primitive can be done more efficiently than testing a general circular cylinder. Thus, one may be tempted (1) to transform a general cylinder into an upright cylinder, (2) to transform the other primitive accordingly, and (3) to perform the more efficient intersection test for the upright cylinder. In the sequel we focus on general cylinders; see Section 3.2 for a discussion of the cpu-time consumption of several intersection tests (with and without transformation).

Let A, B be the end points of a line segment $\mathcal{L}$ which is to be tested for intersection with $\mathcal{C}$, cf. Fig. 4. The following approach, which forms the basis of CylEdge3D, follows the ideas presented in [She94, CW94]. If $\mathcal{L}$ and $\mathcal{L}_c$ are not parallel then the point C of the supporting line of $\mathcal{L}$ which is closest to $\mathcal{L}_c$ fulfills the following equation, for suitable λ, μ, ν: $\mathfrak{c} = \mathfrak{p} + \lambda \cdot \mathfrak{pq} + \mu \cdot \mathfrak{n} = \mathfrak{a} + \nu \cdot \mathfrak{ab}$, where $\mathfrak{n} := \mathfrak{ab} \times \mathfrak{pq}$. By taking the dot product with $\mathfrak{n}$, we obtain $\mu = -\mathfrak{ap} \cdot \mathfrak{n}/|\mathfrak{n}|^2$. Furthermore, the distance δ between the supporting line of $\mathcal{L}$ and $\mathcal{L}_c$ is given by $\delta^2 = (\mathfrak{ap} \cdot \mathfrak{n})^2/|\mathfrak{n}|^2$. If $\mathcal{L}$ and $\mathcal{L}_c$ are parallel then δ^2 is computed as the (squared) distance from A to $\mathcal{L}$. Comparing δ^2 with ρ^2 gives a first test for rejecting lines which are too far away from the cylinder.

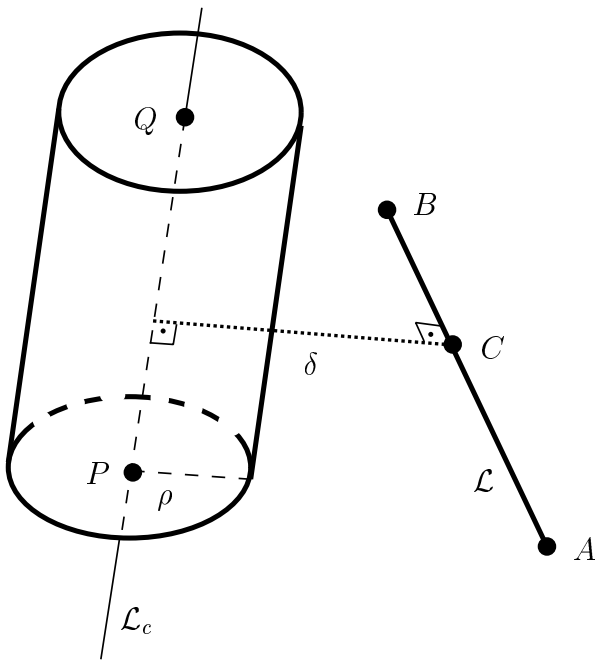


Figure 4: Cylinder/Line-Segment Intersection.

Now assume that the supporting line of $\mathcal{L}$ intersects the infinite cylinder. We will compute a parameter interval $[s_1, s_2]$ such that $\mathfrak{a} + s \cdot \mathfrak{ab}$ lies within the infinite cylinder iff $s_1 \leq s \leq s_2$. Let $[s_3, s_4]$ be the maximum parameter interval of the supporting line of $\mathcal{L}$ such that $\mathfrak{a} + s \cdot \mathfrak{ab}$ lies between $\mathcal{P}'$ and $\mathcal{P}''$ for $s_3 \leq s \leq s_4$. Obviously, $\mathcal{L}$ does not intersect $\mathcal{C}$ if $[s_3, s_4] \cap [0, 1] = \emptyset$, which yields another quick test for rejecting line segments which do not intersect the cylinder. The parameters s_3, s_4 are obtained by taking the minimum and the maximum of the following

two values: $\mathfrak{pq} \cdot \mathfrak{ap}/\mathfrak{pq} \cdot \mathfrak{ab}$ and $\mathfrak{pq} \cdot \mathfrak{aq}/\mathfrak{pq} \cdot \mathfrak{ab}$. (If $\mathcal{L}$ is normal to $\mathcal{L}_c$ then $[s_3, s_4] := [0, 1]$ if $\mathcal{L}$ lies between $\mathcal{P}', \mathcal{P}''$, or no intersection exists.) Finally, $\mathcal{L}$ intersects $\mathcal{C}$ iff $[s_1, s_2] \cap [s_3, s_4] \neq \emptyset$.

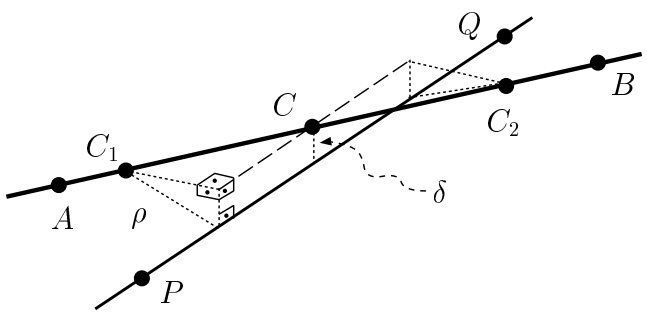


Figure 5: Computing Δs.

It remains to explain how $[s_1, s_2]$ can be determined efficiently. Let C_1, C_2 be the points where the line enters (respectively exits) the infinite cylinder, cf. Fig. 5. Note that $|\mathfrak{c}_1\mathfrak{c}| = |\mathfrak{c}_2\mathfrak{c}|$. We will determine Δs such that $\mathfrak{c}_1 = \mathfrak{c} - \Delta s \cdot \mathfrak{ab}$ and $\mathfrak{c}_2 = \mathfrak{c} + \Delta s \cdot \mathfrak{ab}$. By crossing the equation for λ, μ, ν with $\mathfrak{pq}$ and afterwards taking the dot product with $\mathfrak{n}$, we obtain $\nu = (\mathfrak{ap} \times \mathfrak{pq}) \cdot \mathfrak{n}/|\mathfrak{n}|^2$, which yields C. Now let $\mathfrak{n}' := \mathfrak{pq} \times \mathfrak{n}$. In Fig. 5, $\mathfrak{n}'$ is the direction vector of the dotted line segment through C_1 and the vertex with the three orthogonality signs depicted around it. The length of the projection of $\mathfrak{c}_1\mathfrak{c}$ onto $\mathfrak{n}'/|\mathfrak{n}'|$ is given by $\mathfrak{c}_1\mathfrak{c} \cdot \mathfrak{n}'/|\mathfrak{n}'|$. Since $\mathfrak{c}_1\mathfrak{c} = \Delta s \cdot \mathfrak{ab}$, by applying the Pythagorean theorem we obtain that $(\Delta s \cdot \mathfrak{ab} \cdot \mathfrak{n}'/|\mathfrak{n}'|)^2 = \rho^2 - \delta^2$, i.e., $\Delta s = \sqrt{\rho^2 - \delta^2}|\mathfrak{n}'|/|\mathfrak{ab} \cdot \mathfrak{n}'|$. The parameters s_1, s_2 are readily obtained as $s_1 := \nu - \Delta s$ and $s_2 := \nu + \Delta s$.

For an upright cylinder, the intersection test can be simplified. UpCylEdge3D first determines the parameter interval $[s_3, s_4]$ of that portion of $\mathcal{L}$ which lies between $\mathcal{P}'$ and $\mathcal{P}''$. Since $\mathcal{C}$ is upright, $\mathcal{P}', \mathcal{P}''$ are parallel to the xy-plane. Thus, the parameter of the intersection of $\mathcal{L}$ with $\mathcal{P}'$ is given by $\mathfrak{ap}_z/\mathfrak{ab}_z$. We observe that the distance between $\mathcal{L}_c$, which coincides with the z-axis, and the supporting line of $\mathcal{L}$ can be obtained by applying dimension reduction: Let A', B', P' be the projections of A, B, P onto the xy-plane. The parameter t_c of the point of $\mathcal{L}$ which is closest to $\mathcal{L}_c$ is given by $t_c := \mathfrak{p}'\mathfrak{a}' \cdot \mathfrak{a}'\mathfrak{b}'/|\mathfrak{a}'\mathfrak{b}'|^2$. Let $t := \min(\max(s_3, t_c), s_4)$, and $\mathfrak{c}' := \mathfrak{a}' + t \cdot \mathfrak{a}'\mathfrak{b}'$. Then, $\mathcal{L}$ intersects $\mathcal{C}$ iff $|\mathfrak{p}'\mathfrak{c}'|^2 \leq \rho^2$.

To be able to apply UpCylEdge3D or UpCylTri3D (to be described below) to a general cylinder, we need to transform the cylinder into an upright cylinder. We briefly explain how the 4×4 matrix $\mathbf{M}$ of a rigid transformation is derived which maps P to the origin and Q to a point on the non-negative z-axis, that is $\mathfrak{p} \cdot \mathbf{M} = 0$ and $\mathfrak{q} \cdot \mathbf{M} = (0, 0, h)$, with $h := |\mathfrak{pq}|$. (Here, $\mathfrak{p}$ and $\mathfrak{q}$ have to be regarded to be in homogeneous coordinates.) Assume that $\mathfrak{pq}$ is not parallel to the z-axis. (Otherwise, a simple translation suffices to achieve the desired mapping.) Let $\mathfrak{n}'' := \mathfrak{pq}/|\mathfrak{pq}|$, and select a coordinate of $\mathfrak{n}''$ that is greater than $\frac{1}{2}$ in absolute value.[8] Then pick any of the remaining coordinates of $\mathfrak{n}''$. Without loss of generality we assume that the x, y-coordinates of $\mathfrak{n}''$ got selected. Then, ${\mathfrak{n}'}_x := {\mathfrak{n}''}_y/\Delta$, ${\mathfrak{n}'}_y := -{\mathfrak{n}''}_x/\Delta$, ${\mathfrak{n}'}_z := 0$, where $\Delta := \sqrt{{\mathfrak{n}''}_x^2 + {\mathfrak{n}''}_y^2}$. (Note that the division by Δ is safe as $\Delta > \frac{1}{2}$.) Finally, $\mathfrak{n} := \mathfrak{n}' \times \mathfrak{n}''/|\mathfrak{n}' \times \mathfrak{n}''|$. This yields the three axis vectors of an orthonormal coordinate system, with $\mathfrak{n}''$ being aligned with $\mathfrak{pq}$. Now regard these vectors as the rotational part (rows) of a 4×4 transformation matrix, and use $\mathfrak{p}$ as the translational part (i.e.,

[8] Since $|\mathfrak{n}''| = 1$, one such coordinate must exist.

as the fourth row). Inverting this matrix by transposing its rotational part (and adjusting the translational part) yields the desired matrix $\mathbf{M}$:

$$\mathbf{M} := \begin{pmatrix} \mathfrak{n}_x & \mathfrak{n}'_x & \mathfrak{n}''_x & 0 \\ \mathfrak{n}_y & \mathfrak{n}'_y & \mathfrak{n}''_y & 0 \\ \mathfrak{n}_z & \mathfrak{n}'_z & \mathfrak{n}''_z & 0 \\ -\mathfrak{n}\cdot\mathfrak{p} & -\mathfrak{n}'\cdot\mathfrak{p} & -\mathfrak{n}''\cdot\mathfrak{p} & 1 \end{pmatrix}.$$

It is easy to see that $\mathfrak{p}\cdot\mathbf{M} = 0$ and $\mathfrak{q}\cdot\mathbf{M} = (0,0,h)$. The end points A, B of a line segment are correspondingly transformed as $\mathfrak{a}\cdot\mathbf{M}$ and $\mathfrak{b}\cdot\mathbf{M}$.

Cylinder/Triangle Test: For checking whether $\mathcal{C}$ intersects a triangle $\mathcal{T}$ with vertices A, B, C, the routine CylTri3D uses the generic Algorithm Tri-I. Assume that no boundary intersection exists. If $\mathcal{L}_c$ is normal onto the supporting plane of $\mathcal{T}$ then an interior intersection exists iff $\overline{PQ}$ intersects $\mathcal{T}$. Otherwise, let $\mathfrak{n}' := \mathfrak{pq} \times \mathfrak{n}$, where $\mathfrak{n}$ is the normal vector of $\mathcal{T}$, and consider the plane $\mathcal{P}$ through $\mathcal{L}_c$ with normal vector $\mathfrak{n}'$. Let s be the sign of $\mathfrak{pq}\cdot\mathfrak{n}$, and let $\mathfrak{n}'' := s\cdot\rho\cdot(\mathfrak{n}'\times\mathfrak{pq})/|\mathfrak{n}'\times\mathfrak{pq}|$. The intersection of $\mathcal{P}$ with $\mathcal{C}$ forms a rectangle with vertices P^+, P^- and Q^+, Q^-, where $\mathfrak{p}^+ := \mathfrak{p}+\mathfrak{n}''$, $\mathfrak{p}^- := \mathfrak{p}-\mathfrak{n}''$, $\mathfrak{q}^+ := \mathfrak{q}+\mathfrak{n}''$, and $\mathfrak{q}^- := \mathfrak{q}-\mathfrak{n}''$. Note that P^+ and P^- are those points of the disk containing P that are closest/farthest from the supporting plane of $\mathcal{T}$. (Analogously for Q^+ and Q^-.) A plane normal to $\mathfrak{n}$ that sweeps through space will either enter this rectangle at $\mathfrak{q}^+$ and exit it at $\mathfrak{p}^-$, or enter at $\mathfrak{p}^-$ and exit at $\mathfrak{q}^+$. (Recall that $\mathfrak{n}''$ depends on s.) Thus, the interior of $\mathcal{T}$ intersects $\mathcal{C}$ iff $\mathcal{T}$ intersects the line segment $\overline{P^-Q^+}$.

In the case of an upright cylinder, UpCylTri3D uses the generic Algorithm Tri-II, with $\mathcal{P}', \mathcal{P}''$ as bounding planes. Note that 2D circle/line-segment tests suffice for performing the boundary intersection checks. If no boundary intersection exists then TriEdge3D is called to check $\overline{PQ}$ for intersection with $\mathcal{T}$. (Algorithm Tri-I turned out to be slower than Tri-II for upright cylinders.)

2.5 Cylinder/Sphere Intersection

Consider a cylinder $\mathcal{C}$ defined by P, Q and radius ρ_c, and a sphere $\mathcal{S}$ with center A and radius ρ_s, cf. Fig. 6. Let B be the normal projection of A onto the rotation axis of $\mathcal{C}$, and let $\Delta := |\mathfrak{pb}|$ and $\delta := |\mathfrak{ab}|$. Also, let $h := |\mathfrak{pq}|$ be the height of the cylinder. We claim that $\mathcal{C}$ intersects $\mathcal{S}$ iff $\delta \le (\rho_c + \rho'_s)$, for a suitably chosen ρ'_s.

We observe that $\Delta = |\mathfrak{pq}\cdot\mathfrak{pa}|/h$. The value of ρ'_s depends on the relative order of P, Q, B along the rotation axis of $\mathcal{C}$. The order is $P \to Q \to B$ if $\mathfrak{pq}\cdot\mathfrak{pa} > h^2$, as depicted in Fig. 6. Similarly, the order is $B \to P \to Q$ if $\mathfrak{pq}\cdot\mathfrak{pa} < 0$; otherwise, B is between P and Q. If B is between P and Q then $\rho'_s := \rho_s$. Now assume that the order is $P \to Q \to B$. (The other remaining case, $B \to P \to Q$, is dealt with in a similar manner.) We get $d := \Delta - h = (\mathfrak{pq}\cdot\mathfrak{pa} - h^2)/h$. By applying the Pythagorean theorem twice and substituting for d, we get $\delta^2 = |\mathfrak{pa}|^2 - \Delta^2$ and $\rho'_s := \sqrt{\rho_s^2 - (\mathfrak{pq}\cdot\mathfrak{pa} - h^2)^2/h^2}$. (No intersection exists if $d > \rho_s$, i.e., if the discriminant is negative.)

It remains to check whether $\delta \le (\rho_c + \rho'_s)$, thus ending the description of the routine CylSph3D. The routine UpCylSph3D mimics this approach for the case of an upright cylinder.

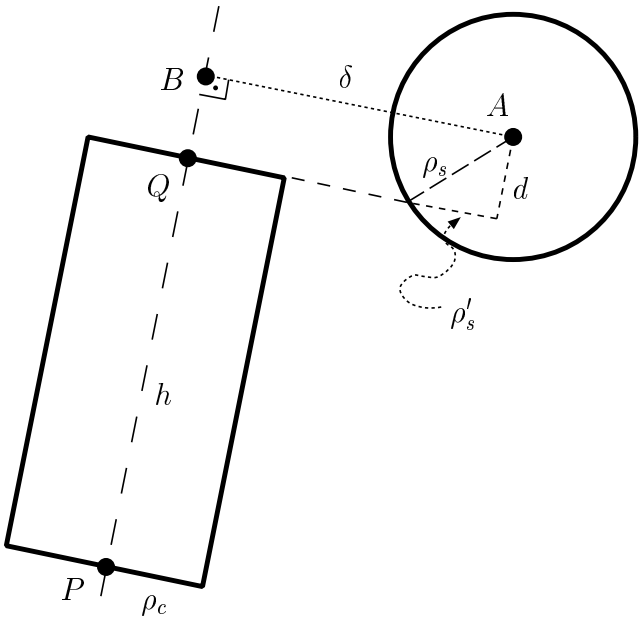


Figure 6: Cylinder/Sphere Intersection Test.

2.6 Cone/Line-Segment and Cone/Triangle Tests

Based on the experience gained with the cylinder/line-segment and cylinder/triangle tests, we decided to implement cone/line-segment and cone/triangle tests only for upright cones. (A general cone can always be transformed into an upright position; see Section 3.2 for a discussion of the cpu-time consumption.) An upright (frustum of a) cone $\mathcal{C}$ is described by a line $\mathcal{L}_c$ parallel to the z-axis, by points P and Q on $\mathcal{L}_c$, and by two radii ρ_b and ρ_t which define the bottom and the top disks of the cone, cf. Fig. 7. (Possibly, either ρ_b or ρ_t may be zero.) Without loss of generality, P is at the origin, and Q is on the positive z-axis, i.e., $\mathfrak{q} = (0, 0, h)$, where h is the height of the cone. Let $\mathcal{P}', \mathcal{P}''$ be the supporting planes of the bottom and top disks.

Cone/Line-Segment Test: Consider a line segment $\mathcal{L}$ that is to be checked by UpConEdge3D for intersection with the upright cone $\mathcal{C}$. Let A, B denote the end points of $\mathcal{L}$ trimmed to the space between $\mathcal{P}'$ and $\mathcal{P}''$. For a z-coordinate $\bar{z}$, with $0 \leq \bar{z} \leq h$, the radius $\rho(\bar{z})$ of the disk obtained by slicing $\mathcal{C}$ with the plane $z = \bar{z}$ is given by $\rho(\bar{z}) := \rho_b + (\rho_t - \rho_b)\bar{z}/h$. Obviously, A is within the cone $\mathcal{C}$ iff the distance δ between A and $\mathcal{L}_c$ is not greater than $\rho(\mathfrak{a}_z)$. (In our implementation we compare $\delta^2 = \mathfrak{a}_x^2 + \mathfrak{a}_y^2$ to $\rho(\mathfrak{a}_z)^2$.)

Assume that neither A nor B lies within $\mathcal{C}$. If $\mathcal{L}$ is parallel to the x, y-plane then a simple 2D circle/line-segment test suffices. Otherwise, assume that $\rho_b \neq \rho_t$, and, without loss of generality, that $\rho_b > \rho_t$, and $\rho_b > 0$. (If $\rho_b = \rho_t$ then $\mathcal{C}$ is a cylinder and we know how to proceed, and if $\rho_b = 0$ then the cone degenerates to a line segment, and we return "no intersection".)

If ρ_t is not very small compared to ρ_b, say $\rho_b > \frac{1}{100}\rho_t$, then we apply a projective transformation to $\mathcal{L}$ and $\mathcal{C}$ that maps $\mathcal{C}$ into an upright cylinder with radius ρ_b. Let A' and B' be the images of A and B under this transformation. We observe that $\mathfrak{a}'_x = \mathfrak{a}_x\rho_b/\rho(\mathfrak{a}_z)$, and $\mathfrak{a}'_y = \mathfrak{a}_y\rho_b/\rho(\mathfrak{a}_z)$; similarly for $\mathfrak{b}'$. It is easy to see that the line segment $\overline{AB}$ intersects $\mathcal{C}$ iff the minimum distance

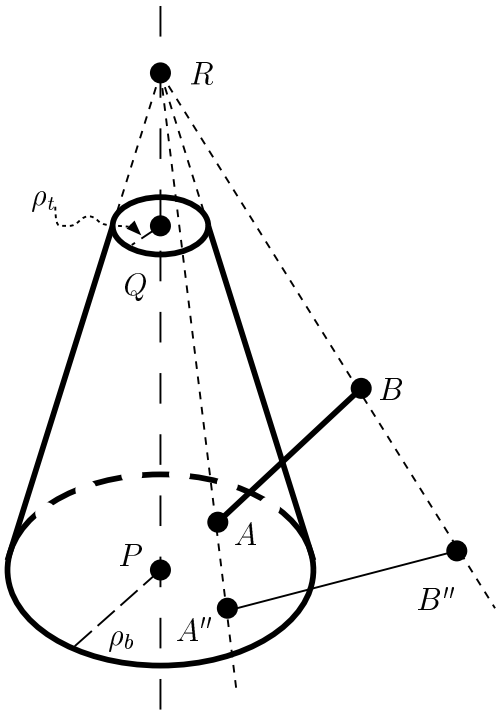


Figure 7: Cone/Line-Segment Intersection.

between $\overline{A'B'}$ and $\mathcal{L}_c$ is at most ρ_b, which can be decided by means of a 2D circle/line-segment test.

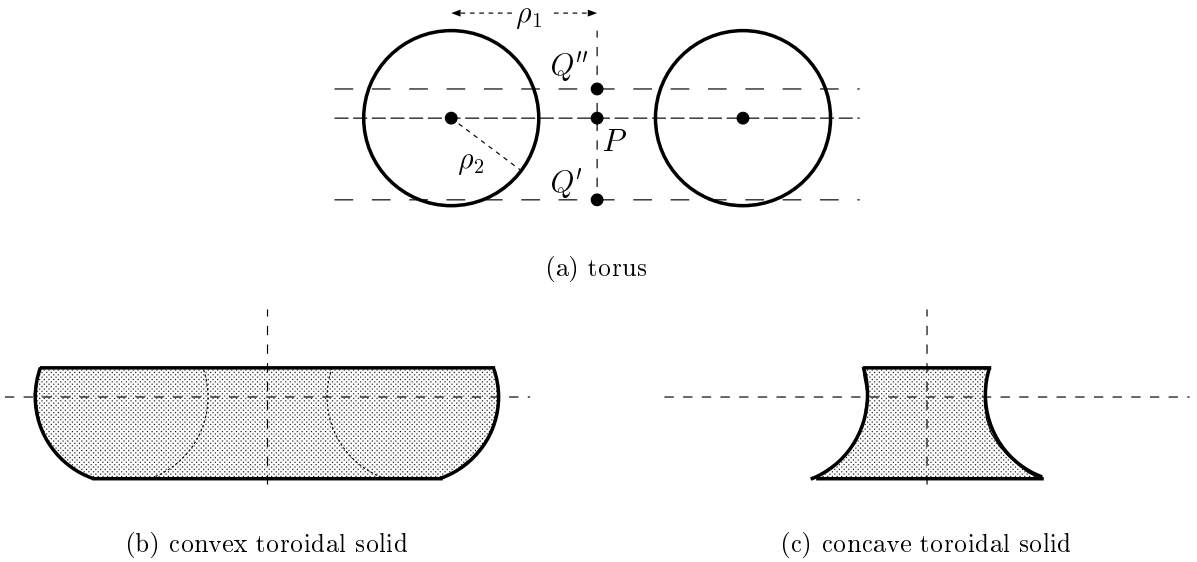
If $\rho_t \ll \rho_b$ then the division by $\rho(\mathfrak{a}_z)$ or $\rho(\mathfrak{b}_z)$ may be unreliable. In this case we compute the point R on $\mathcal{L}_c$ for which $\rho(\mathfrak{r}_z) = 0$. Due to our assumption on ρ_b, ρ_t we have that R is not identical to P. (It may be identical to Q, though; in any case it is not 'very far' away from Q.) Using R as the center of projection, we project A and B onto the x, y-plane, thus obtaining the points A'', B''. Again, $\overline{AB}$ intersects $\mathcal{C}$ iff $\overline{A''B''}$ intersects the bottom disk of $\mathcal{C}$.

Cone/Triangle Test: UpConTri3D uses the generic Algorithm Tri-II, with $\mathcal{P}', \mathcal{P}''$ as bounding planes. If no boundary intersection exists then it tests whether $\overline{PQ}$ intersects the triangle. (Algorithm Tri-I turned out to be slower than Tri-II for upright cones.)

2.7 Toroid/Line-Segment and Toroid/Triangle Intersection

A torus is a solid of revolution that is generated by sweeping a disk of radius ρ_2 around an axis $\mathcal{L}$. (The disk and the axis of rotation always stay coplanar.) Fig. 8(a) shows the cross section of a torus with radius ρ_1 and disk radius ρ_2. Its axis of rotation passes through the point P. In the sequel, we will assume that $\mathcal{L}$ is the z-axis and that P equals the origin; a general torus can easily be transformed into this position. Now pick two points Q', Q'' on $\mathcal{L}$ such that their distance from P is at most ρ_2, and such that $\mathfrak{q}'_z \leq \mathfrak{q}''_z$. We subtract from the torus all points whose z-coordinate is less than $\mathfrak{q}'_z$ or greater than $\mathfrak{q}''_z$. By taking the convex hull of the resulting solid, a convex toroidal solid is obtained, cf. Fig. 8(b). Subtracting the torus from this solid yields a concave toroidal solid,

as depicted in Fig. 8(c).



(a) torus

(b) convex toroidal solid

(c) concave toroidal solid

Figure 8: Cross sections of toroidal solids of revolution.

Toroid/Line-Segment Test: UpCvxTorEdge3D, which handles a convex toroid, and UpCveTorEdge3D, which handles a concave toroid, both start with clipping the line segment to the z-interval spanned by the toroid. Let A, B be the end points of the clipped line segment $\mathcal{L}$. For a z-coordinate $\bar{z}$, with $\mathfrak{q}'_z \leq \bar{z} \leq \mathfrak{q}''_z$, the radius $\rho(\bar{z})$ of the disk obtained by slicing the toroid with the plane $z = \bar{z}$ is given as $\rho(\bar{z}) := \rho_1 + \sqrt{\rho_2^2 - \bar{z}^2}$, in the case of a convex toroid, and as $\rho(z') := \rho_1 - \sqrt{\rho_2^2 - \bar{z}^2}$ in the case of a concave toroid. (Recall that P is at the origin.) This allows to decide quickly whether A or B lie in the interior of the toroid.

If both A and B lie in the exterior of the toroid then we determine the z-coordinate $\bar{z}$, with $\mathfrak{q}'_z \leq \bar{z} \leq \mathfrak{q}''_z$, for which $\rho(\bar{z})$ achieves the maximum. If the normal projection of $\mathcal{L}$ onto the xy-plane does not intersect a disk with radius $\rho(\bar{z})$ centered at P then no intersection between $\mathcal{L}$ and the toroid exists. Otherwise, a quartic equation is obtained by expressing $\mathcal{L}$ in parameterized form $\mathfrak{a} + t \cdot \mathfrak{ab}$, and by substituting for x, y, z in the following equation of the torus: $(x^2 + y^2 + z^2 + \rho_1^2 - \rho_2^2)^2 - 4\rho_1^2(x^2 + z^2) = 0$. The resulting quartic equation is then solved by standard algebraic methods.[9] Let R be a candidate intersection, given as $\mathfrak{r} = \mathfrak{a} + t \cdot \mathfrak{ab}$, for some real root t of the quartic equation. In the case of a convex toroid, R is a valid intersection iff $0 \leq t \leq 1$. (Recall that we have clipped the line segment appropriately.) In the case of a concave toroid, we also need to check that $\mathfrak{r}_x^2 + \mathfrak{r}_y^2 \leq \rho_1^2$, i.e., that R is at distance at most ρ_1 from the toroid's axis of rotation.

Toroid/Triangle Test: UpCvxTorTri3D and UpCveTorTri3D both apply the generic Algorithm Tri-II. Similar to the cone/triangle tests, additional triangle/line-segment tests have

[9] Our quartic solver is a thorough re-work of the one presented in [Sch90]; the original code of [Sch90] does not seem to work correctly.

to be performed if no boundary intersection exists. While one additional test suffices for convex toroidal solids, up to three tests are needed for concave toroidal solids.

3 Implementation and Results

3.1 Implementational Details of ERIT

ERIT has been implemented in ANSI C. Except for standard arithmetic tricks used for achieving an increased efficiency, the intersection tests have been implemented as described above. (For instance, for the sake of saving two divisions, we would compute the scalar product of two vectors and afterwards divide by the norm of one of the vectors, rather than normalizing this vector prior to computing the scalar product. Also, we compare squared distances to minimize the number of square roots that need to be computed.)

ERIT operates on the data type double. No (data) structures are used other than a C structure called vertex, which contains the three coordinates of a point. All low-level routines, both for handling vector arithmetic as well as for other basic computations (such as determinants), have been implemented as C macros: ERIT is entirely macro-based, except for arithmetic operations performed on scalar data. This makes the code efficient, easy to read, and easy to modify.

As with any other code that operates on standard floating-point arithmetic, handling numerical degeneracies requires some care. We have chosen the conventional ϵ-based approach to dealing with this problem; most comparisons are carried out with respect to two user-defined precision thresholds: ZERO and EPS. (Some low-level functions and macros may scale these thresholds somewhat to adjust them to the size of the input coordinates.)

Both thresholds, ZERO and EPS, are define'd in every file, and (by default) initialized by means of two global variables. The first threshold, ZERO, is used whenever a division needs to be carried out (or a square root needs to be computed). Clearly, it should be set to a small number but must not be set to zero to avoid arithmetic exceptions. (In our own applications, we have set ZERO to 10^{-12}.) The other precision threshold, EPS, is used for detecting grazing contacts, etc.

Since we anticipate that a user may want to change our approach to carrying out floating-point comparisons for some particular application, we have cast all comparisons into C macros. That is, checking whether some value x is less than zero is done by means of 'lt(x,eps)', where 'eps' is one of the two precision thresholds; the C macro 'lt' translates to '$x < -eps$'. Similarly, checking whether x is less than or equal to zero is done by means of 'le(x,eps)', which translates to '$x \leq eps$'. Complementary comparisons have been implemented through negation; e.g., 'gt(x,eps)' is defined as '!le(x,eps)'. If speed is of utmost importance then the user may want to set EPS to zero. In this case all comparison macros that are not vital for preventing a crash of the code will function as their conventional counterparts would function.

To test the practical reliability of ERIT in an automated way, we subjected all intersection routines to tests with hundreds of thousands of pairs of (random) primitives. The random primitives where defined by vertices randomly chosen within the unit cube. (We have algorithmically different implementations for most of the intersection tests, which gave us a simple means for automatically checking whether our intersection routines agree.) Since ERIT forms the core of our code for collision detection and NC verification [KHM+98], it also gets tested on real world-data on a regular basis.

3.2 Test Results

ERIT was embedded into a test bed for determining cpu use. We generated random instances of primitive objects, and ran the intersection tests on all those pairs of primitives for which the bounding boxes overlapped. Table 1 lists the cpu-time consumptions (in milliseconds) of our intersection tests, for 100,000 tests each. The entries of the table were computed by normalizing the timings recorded from running several hundreds of thousands of intersection tests. Every entry includes the overhead caused by stepping through a 'for'-loop repeatedly, accessing the precomputed data (vertices, etc.) for the primitives, and the costs of bounding-box tests. (Of course, all bounding-box tests were unnecessary as we had already weeded out all pairs of primitives that could be rejected by a bounding box test.)

The tests were run on three different hardware platforms: on a Sun Ultra 30 running Solaris 2.6, a PentiumPro PC with a 200MHz processor running Red Hat LINUX 4.1, and an SGI Indigo 2 with an R10000 processor running IRIX 6.2. The code was compiled using GNU's `gcc`, with the '`-O2`' flag. All timings were obtained by using the C system function 'getrusage()'. (We report both the system and the user time.)

Intersection Test	Ultra 30	PentiumPro	Indigo-2
TriEdge3D	229	495	250
TriTri3D	230	495	252
Möller	234	581	274
TriPnt2D	83	163	99
SphEdge3D	107	180	110
SphTri3D	206	471	204
SphSph3D	67	118	82
UpCylEdge3D	292	701	349
CylEdge3D	287	794	310
UpCylTri3D	381	842	423
CylTri3D	502	1,285	533
CylTriPrepro	208	466	245
UpCylSph3D	233	551	281
CylSph3D	227	533	235
UpConEdge3D	333	857	416
UpConTri3D	495	1,096	580
UpCvxTorEdge3D	412	955	515
UpCvxTorTri3D	625	1,790	704
UpCveTorEdge3D	472	1,056	597
UpCveTorTri3D	653	1,844	766

Table 1: CPU times (in milliseconds) for 100,000 intersection tests.

We explicitly caution a user of ERIT not to expect to get exactly the same performance figures when running the code on a machine 'identical' to one of the test platforms. Several minor details of the configuration can influence the speed of the code.

We started with comparing TriTri3D to Möller's triangle/triangle test [Möl97]. (We ftp'ed his code from the site given in [Möl97], and integrated it into our test bed.) While the two triangle/triangle tests were of the same speed on the Ultra 30, TriTri3D was significantly faster than Möller's code on the PentiumPro, and somewhat faster on the Indigo-2. It is also interesting to note that a triangle/line-segment test takes nearly as long as a triangle/triangle test, on all three platforms.

TriPnt2D, SphEdge3D, and SphSph3D have been implemented as C macros rather than C functions. Typically, for these tests it does not seem to pay off to perform a bounding-box pre-test. (The timings reported include the overhead of a bounding-box pre-test, though.)

The timings get more interesting for the intersection tests involving a cylinder. We tested general cylinders, which were transformed into an upright position before calling an intersection test restricted to the handling of upright cylinders. Note that, for comparison purposes, the entries for the 'Up'-tests list the time consumed by the actual intersection test plus the time consumed by this transformation. Does it pay off to transform a cylinder into an upright position? Apparently, for cylinder/line-segment tests this depends on the platform used, see Table 1. For cylinder/triangle tests the transformation always seems to pay off, and UpCylTri3D always was clearly faster than CylTri3D. It is natural to conclude that the transformation pays off the more the more complicated an intersection test is. For instance, for the simple cylinder/sphere test, the transformation does not pay off on any of the three platforms.

We also measured the cpu-time consumption of CylTriPrepro, i.e., of the transformation of the cylinders and triangles. Thus, the actual cost of a test for an upright cylinder (that does not need to be transformed) equals roughly the appropriate entry listed in the table minus the entry listed for CylTriPrepro. Note that cache sizes may have influenced these timings. (The transformation of the primitives was implemented with C macros, thus increasing the number of local variables to be kept within the test loop.)

Summarizing, for more complicated intersection tests it seems to pay off to transform a cylinder into an upright position before invoking an intersection test. For this reason, the intersection tests for a cone or a toroid versus a line segment or a triangle have only been implemented for the upright case. Timings for those tests are included in Table 1.

Acknowledgments

The C macros for transforming a cylinder or cone into its upright position are based on macros for handling 4×4 transformation matrices, which were derived from C code written by J. Cychosz [Cyc90]. I have benefited from discussions with J. Klosowski and J. Mitchell. Detailed comments of J. Hughes helped to improve this paper.

This work was supported by grants from Boeing Computer Services, Bridgeport Machines, Sun Microsystems, and by NSF Grant CCR-9504192. The Sun Ultra 30 used in the tests was donated by Sun Microsystems.

Availability of ERIT

The source code for ERIT is available upon sending a request to the author (`held@cosy.sbg.ac.at`).

References

[Arv91] J. Arvo, editor. *Graphics Gems II.* Academic Press, 1991. ISBN 0-12-064480-9.

[CW94] J.M. Cychosz and W.N. Waggenspack. Intersecting a Ray with a Cylinder. In P.S. Heckbert, editor, *Graphics Gems IV*, pages 356–365. Academic Press, 1994. ISBN 0-12-336155-9.

[Cyc90] J.M. Cychosz. Efficient Post-Concatenation of Transformation Matrices. In A.S. Glassner, editor, *Graphics Gems*, pages 476–481. Academic Press, 1990. ISBN 0-12-286166-3.

[FvDFH90] J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes. *Computer Graphics: Principles and Practice.* The Systems Programming Series. Addison-Wesley, 1990. ISBN 0-301-12110-7.

[Gla90] A.S. Glassner, editor. *Graphics Gems.* Academic Press, 1990. ISBN 0-12-286166-3.

[Gla92] A.S. Glassner. *Ray Tracing: Theory and Practice.* Morgan Kaufmann, 1992.

[Hec94] P.S. Heckbert, editor. *Graphics Gems IV.* Academic Press, 1994. ISBN 0-12-336155-9.

[Hub96] P.M. Hubbard. Approximating Polyhedra with Spheres for Time-Critical Collision Detection. *ACM Trans. Graph.*, 15(3):179–210, July 1996.

[KHM+98] J.T. Klosowski, M. Held, J.S.B. Mitchell, H. Sowizral, and K. Zikan. Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs. *IEEE Trans. Visualizat. Comput. Graph.*, 4(1):21–36, Jan 1998.

[Kir92] D. Kirk, editor. *Graphics Gems III.* Academic Press, 1992. ISBN 0-12-409670-0.

[KK86] J.T. Kajiya and T.L. Kay. Ray Tracing Complex Scenes. In *Comput. Graphics (SIGGRAPH '86 Proc.)*, volume 20, pages 269–278, Dallas, TX, USA, Aug 1986.

[Möl97] T. Möller. A Fast Triangle-Triangle Intersection Test. *J. Graphics Tools*, 2(2):25–30, 1997.

[MT97] T. Möller and B. Trumbore. Fast, Minimum Storage Ray-Triangle Intersection. *J. Graphics Tools*, 2(1):21–28, 1997.

[Pae95] A.W. Paeth, editor. *Graphics Gems V.* Academic Press, 1995. ISBN 0-12-543455-3.

[Sch90] J. Schwarze. Cubic and Quartic Roots. In A.S. Glassner, editor, *Graphics Gems*, pages 404–407. Academic Press, 1990. ISBN 0-12-286166-3.

[She94] C.-K. Shene. Computing the Intersection of a Line and a Cylinder. In P.S. Heckbert, editor, *Graphics Gems IV*, pages 353–355. Academic Press, 1994. ISBN 0-12-336155-9.

[Wat89] A. Watt. *Fundamentals of Three-Dimensional Computer Graphics.* Addison-Wesley, 1989. ISBN 0-201-15442-0.