Intersection of a Line and a Box

David Eberly, Geometric Tools, Redmond WA 98052
https://www.geometrictools.com/

Created: November 28, 2018
Last Modified: September 11, 2020

Contents

1 Introduction

Let the oriented box have center $\mathbf{C}$, orthonormal axis directions $\mathbf{U}_i$, and extents e_i for $0 \leq i \leq 2$. Points in the oriented box are $\mathbf{X} = \mathbf{C} + \sum_{i=0}^{2} y_i \mathbf{U}_i$ where $|y_i| \leq e_i$. The extents can be stored as a 3-tuple $\mathbf{e} = (e_0, e_1, e_2)$. The coefficients of the $\mathbf{U}_i$ can be written as a 3-tuple $\mathbf{Y} = (y_0, y_1, y_2)$. The oriented box points are then $\mathbf{X} = \mathbf{C} + R\mathbf{Y}$ with $-\mathbf{e} \leq \mathbf{Y} \leq \mathbf{e}$. The inequality comparisons are componentwise.

An aligned box is defined by a minimum point $\mathbf{a}$ and a maximum point $\mathbf{b}$. A point $\mathbf{X}$ in the box satisfies $\mathbf{a} \leq \mathbf{X} \leq \mathbf{b}$, where the inequality comparisons are componentwise.

An aligned box can be converted to an oriented box by choosing $\mathbf{C} = (\mathbf{b} + \mathbf{a})/2$ and $\mathbf{e} = (\mathbf{b} - \mathbf{a})/2$. An oriented box is an aligned box in its own coordinate system, as described next. Let $R = [\mathbf{U}_0 \mathbf{U}_1 \mathbf{U}_2]$, a rotation matrix with the specified columns. In a geometric query you can transform points by $\mathbf{Y} = R^{\mathsf{T}}(\mathbf{X} - \mathbf{C})$. The minimum point is $-\mathbf{e}$ and the maximum point is $\mathbf{e}$. The box axis directions are $(1, 0, 0)$, $(0, 1, 0)$ and $(0, 0, 1)$.

The linear component is $\mathbf{P} + t\mathbf{D}$ where $\mathbf{D}$ is a unit-length vector. The component is a line when $t \in (-\infty, +\infty)$, a ray when $t \in [0, +\infty)$ or a segment when $t \in [-\varepsilon, \varepsilon]$ for some $\varepsilon > 0$.

This document discusses test-intersection queries between the box and a linear component. In this approach we care only whether the objects intersect but not where they intersect. The algorithm is based on the method of separating axes which in turn is based on the Minkowski differences of sets.

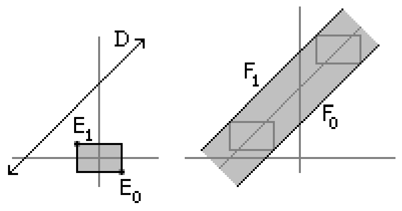
The document also discusses find-intersection queries where we do want to the points of intersection. The Liang–Barsky clipping algorithm is used.

2 Test-Intersection Queries

2.1 Lines and Boxes

The application of the method of separating axes to a line and an oriented box is described here. The Minkowski difference of the oriented box and the line is an infinite convex polyhedron obtained by extruding the oriented box along the line and placing it appropriately in space. Figure 1 illustrates this process in two dimensions.

Figure 1. An oriented box and a line. The oriented box is extruded along the line in the direction $\mathbf{D}$. The illustration is based on the oriented box having its center at the origin. If the center were not at the origin, the extruded object would be translated from its current location.



Four of the oriented box edges are perpendicular to the plane of the page. Two of those edges are highlighted with points and labeled as E_0 and E_1. The edge E_0 is extruded along the line direction $\mathbf{D}$. The resulting face, labeled F_0, is an infinite planar strip with a normal vector $\mathbf{U}_0 \times \mathbf{D}$, where $\mathbf{U}_0$ is the unit-length normal of the face of the oriented box that is coplanar with the page. The edge E_1 is extruded along the line direction to produce face F_1. Because edges E_0 and E_1 are parallel, face F_1 also has a normal vector $\mathbf{U}_0$. The maximum number of faces that the infinite polyhedron can have is six (project the oriented box onto a plane with normal $\mathbf{D}$ and obtain a hexagon), one for each of the independent oriented box edge directions. These directions are the same as the oriented box face normal directions, so the six faces are partitioned into three pairs of parallel faces with normal vectors $\mathbf{U}_i \times \mathbf{D}$.

Now that we have the potential separating axis directions, the separation tests are

$$\begin{aligned} |\mathbf{D} \times \mathbf{\Delta} \cdot \mathbf{U}_0| &> e_1|\mathbf{D} \cdot \mathbf{U}_2| + e_2|\mathbf{D} \cdot \mathbf{U}_1| \\ |\mathbf{D} \times \mathbf{\Delta} \cdot \mathbf{U}_1| &> e_0|\mathbf{D} \cdot \mathbf{U}_2| + e_2|\mathbf{D} \cdot \mathbf{U}_0| \\ |\mathbf{D} \times \mathbf{\Delta} \cdot \mathbf{U}_2| &> e_0|\mathbf{D} \cdot \mathbf{U}_1| + e_1|\mathbf{D} \cdot \mathbf{U}_0| \end{aligned} \tag{1}$$

where $\mathbf{\Delta} = \mathbf{P} - \mathbf{C}$. The term $\mathbf{D} \times \mathbf{\Delta} \cdot \mathbf{U}_i$ is used instead of the mathematically equivalent $\mathbf{U}_i \times \mathbf{D} \cdot \mathbf{\Delta}$ in order for the implementation to compute $\mathbf{D} \times \mathbf{\Delta}$ once, leading to a reduced operation count for all three separation tests. Pseudocode for the test-intersection query of a line and an oriented box in 3 dimensions is shown in Listing 1.

Listing 1. Pseudocode for the test-intersection query between a line and an oriented box in 3 dimensions. The function abs returns the absolute value of its scalar argument.

```
bool TestIntersection(Line3 line, OrientedBox3 box)
{
    Vector3 PmC = line.P - box.C;
    Vector3 DxPmC = Cross(line.D, PmC);
    Real ADdU[3], ADxPmCdU[3];

    ADdU[1] = abs(Dot(line.D, box.U[1]));
    ADdU[2] = abs(Dot(line.D, box.U[2]));
    ADxPmCdU[0] = abs(Dot(DxPmC, box.U[0]));
    if (ADxPmCdU[0] > box.e[1] * ADdU[2] + box.e[2] * ADdU[1])
    {
        return false;
    }

    ADdU[0] = abs(Dot(line.D, box.U[0]));
    ADxPmCdU[1] = abs(Dot(DxPmC, box.U[1]));
    if (ADxPmCdU[1] > box.e[0] * ADdU[2] + box.e[2] * ADdU[0])
    {
        return false;
    }

    ADxPmCdU[2] = abs(Dot(DxPmC, box.U[2]));
    if (ADxPmCdU[2] > box.e[0] * ADdU[1] + box.e[1] * ADdU[0])
    {
        return false;
    }

    return true;
}
```

We can formulate the test-intersection for an aligned box by using the center-extent form of the box. Listing 2 contains pseudocode for the query.

Listing 2. Pseudocode for the test-intersection query between a line and an aligned box in 3 dimensions.

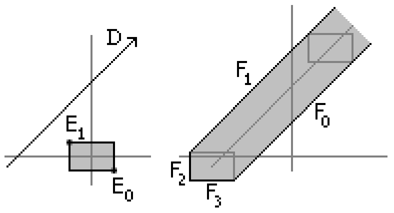
```
// P is the line origin, D is the unit-length line direction, and e contains
// the box extents.  The box center has been translated to the origin and
// the line has been translated accordingly.
bool DoLineQuery(Vector3 P, Vector3 D, Vector3 e)
{
    Vector3 DxP = Cross(D, P);

    if (abs(DxP[0]) > e[1] * abs(D[2]) + e[2] * abs(D[1]))
    {
        return false;
    }

    if (abs(DxP[1]) > e[0] * abs(D[2]) + e[2] * abs(D[0]))
    {
        return false;
    }

    if (abs(DxP[2]) > e[0] * abs(D[1]) + e[1] * abs(D[0]))
    {
        return false;
    }

    return true;
}

bool TestIntersection(Line3 line, AlignedBox3 box)
{
    // Get the centered form of the aligned box.
    Vector3 C = (box.max + box.min) / 2;
    Vector3 e = (box.max - box.min) / 2;

    // Transform the line to the aligned-box coordinate system.
    Vector3 P = line.P - C;
    Vector3 D = line.D;

    return DoLineQuery(P, D, e);
}

// The query for the line and oriented box but reformulated to use DoLineQuery.
bool TestIntersection(Line3 line, OrientedBox3 box)
{
    // Transform the line to the oriented-box coordinate system.
    Vector3 delta = line.P - box.C;
    Vector3 P = { Dot(delta, box.U[0]), Dot(delta, box.U[1]), Dot(delta, box.U[2]);
    Vector3 D = { Dot(line.D, box.U[0]), Dot(line.D, box.U[1]), Dot(line.D, box.U[2]);

    return DoLineQuery(P, D, box.e);
}
```

The GTE files IntrLine3AlignedBox3.h and IntrLine3OrientedBox3.h implement this design.

2.2 Rays and Boxes

The infinite convex polyhedron that corresponds to the Minkowski difference of a line and an oriented box becomes a semi-infinite object in the case of a ray and an oriented box. Figure 2 illustrates this process in two dimensions.

Figure 2. An oriented box and a ray. The oriented box is extruded along the ray in the direction $\mathbf{D}$. The faces F_0 and F_1 are generated by oriented box edges and $\mathbf{D}$. The faces F_2 and F_3 are contributed from the oriented box.

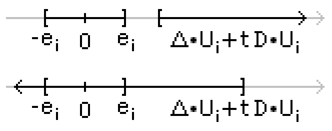


The semi-infinite convex polyhedron has the same three pairs of parallel faces as for the line, but the polyhedron has the oriented box as an end cap. The oriented box contributes three additional faces and corresponding normal vectors. Thus, we have six potential separating axes. The separation tests are

$$\begin{aligned}
&|\mathbf{D}\times\mathbf{\Delta}\cdot\mathbf{U}_0| > e_1|\mathbf{D}\cdot\mathbf{U}_2| + e_2|\mathbf{D}\cdot\mathbf{U}_1| \\
&|\mathbf{D}\times\mathbf{\Delta}\cdot\mathbf{U}_1| > e_0|\mathbf{D}\cdot\mathbf{U}_2| + e_2|\mathbf{D}\cdot\mathbf{U}_0| \\
&|\mathbf{D}\times\mathbf{\Delta}\cdot\mathbf{U}_2| > e_0|\mathbf{D}\cdot\mathbf{U}_1| + e_1|\mathbf{D}\cdot\mathbf{U}_0| \\
&|\mathbf{\Delta}\cdot\mathbf{U}_0| > e_0,\ \ (\mathbf{\Delta}\cdot\mathbf{U}_0)(\mathbf{D}\cdot\mathbf{U}_0) \geq 0 \\
&|\mathbf{\Delta}\cdot\mathbf{U}_1| > e_1,\ \ (\mathbf{\Delta}\cdot\mathbf{U}_1)(\mathbf{D}\cdot\mathbf{U}_1) \geq 0 \\
&|\mathbf{\Delta}\cdot\mathbf{U}_2| > e_2,\ \ (\mathbf{\Delta}\cdot\mathbf{U}_2)(\mathbf{D}\cdot\mathbf{U}_2) \geq 0
\end{aligned} \tag{2}$$

The first three are the same as for a line. The last three use the oriented box face normals for the separation tests. To illustrate these tests, see Figure 3.

Figure 3. Projections of an oriented box and a ray onto the line with direction $\mathbf{U}_i$ which is a normal to an oriented box face. The oriented box center $\mathbf{C}$ is subtracted from the oriented box as well as the ray origin $\mathbf{P}$. The translated oriented box projects to the interval $[-e_i, e_i]$, where e_i is the oriented box extent associated with $\mathbf{U}_i$. The translated ray is $\mathbf{\Delta} + t\mathbf{D}$, where $\mathbf{\Delta} = \mathbf{P} - \mathbf{C}$, and projects to $\mathbf{U}_i \cdot \mathbf{\Delta} + t\mathbf{U}_i \cdot \mathbf{D}$.



The projected oriented box is the finite interval $[-e_i, e_i]$. The projected ray is a semi-infinite interval on the t-axis. The origin is $\mathbf{\Delta}\cdot\mathbf{U}_i$, and the direction (a signed scalar) is $\mathbf{D}\cdot\mathbf{U}_i$. The top portion of the figure shows a positive signed direction and an origin that satisfies $\mathbf{\Delta}\cdot\mathbf{U}_i > e_i$. The finite interval and the semi-infinite interval are disjoint, in which case the original oriented box and ray are separated. If instead the projected ray direction is negative, $\mathbf{D}\cdot\mathbf{U}_i < 0$, the semi-infinite interval overlaps the finite interval. The original oriented box and ray are not separated by the axis with direction $\mathbf{U}_i$. Two similar configurations exist when $\mathbf{\Delta}\cdot\mathbf{U}_i < -e_i$. The condition $|\mathbf{\Delta}\cdot\mathbf{U}_i| > e_i$ says that the projected ray origin is farther away from zero than the projected box extents. The condition $(\mathbf{\Delta}\cdot\mathbf{U}_i)(\mathbf{D}\cdot\mathbf{U}_i) > 0$ guarantees that the projected ray points away from the projected oriented box.

The implementation of the test-intersection query for a ray and oriented box in 3 dimensions is shown in Listing 3.

Listing 3. Pseudocode for the test-intersection query between a ray and an oriented box in 3 dimensions. The function abs returns the absolute value of its scalar argument.

```
bool TestIntersection(Ray3 ray, OrientedBox3 box)
{
    Vector3 PmC = ray.P - box.C;
    Real DdU[3], ADdU[3], PmCdU[3], APmCdU[3];

    // The ray-specific tests.
    for (int i = 0; i < 3; ++i)
    {
        DdU[i] = Dot(ray.D, box.U[i]);
        ADdU[i] = abs(DdU[i]);
        PmCdU[i] = Dot(PmC, box.U[i]);
        APmCdU[i] = abs(PmCdU[i]);
        if (APmCdU[i] > box.e[i] and PmCdU[i] * DdU[i] >= 0)
        {
            return false;
        }
    }

    // The line-specific tests.
    Vector3 DxPmC = Cross(ray.D, PmC);
    Real ADxPmCdU[3];

    ADxPmCdU[0] = abs(Dot(DxPmC, box.U[0]));
    if (ADxPmCdU[0] > box.e[1] * ADdU[2] + box.e[2] * ADdU[1])
    {
        return false;
    }

    ADxPmCdU[1] = abs(Dot(DxPmC, box.U[1]));
    if (ADxPmCdU[1] > box.e[0] * ADdU[2] + box.e[2] * ADdU[0])
    {
        return false;
    }

    ADxPmCdU[2] = abs(Dot(DxPmC, box.U[2]));
    if (ADxPmCdU[2] > box.e[0] * ADdU[1] + box.e[1] * ADdU[0])
    {
        return false;
    }

    return true;
}
```

We can formulate the test-intersection for an aligned box by using the center-extent form of the box. Listing 4 contains pseudocode for the query.

Listing 4. Pseudocode for the test-intersection query between a ray and an aligned box in 3 dimensions.

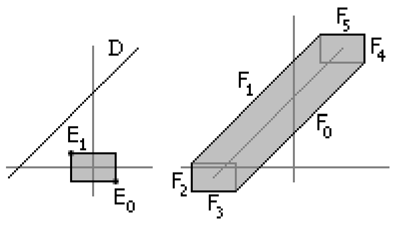
```
// P is the ray origin, D is the unit-length ray direction and e contains
// the box extents. The box center has been translated to the origin and
// the ray has been translated accordingly.
bool DoRayQuery(Vector3 P, Vector3 D, Vector3 e)
{
    for (int i = 0; i < 3; ++i)
    {
        if (abs(P[i]) > e[i] and P[i] * D[i] >= 0)
        {
            return false;
        }
    }
    return DoLineQuery(P, D, e);
}

bool TestIntersection(Ray3 ray, AlignedBox3 box)
{
    // Get the centered form of the aligned box.
    Vector3 C = (box.max + box.min) / 2;
    Vector3 e = (box.max − box.min) / 2;

    // Transform the ray to the aligned−box coordinate system.
    Vector3 P = ray.P − C;
    Vector3 D = ray.D;

    return DoRayQuery(P, D, e);
}

// The query for the ray and oriented box is formulated to use DoRayQuery.
bool TestIntersection(Ray3 ray, OrientedBox3 box)
{
    // Transform the ray to the oriented−box coordinate system.
    Vector3 delta = ray.P − box.C;
    Vector3 P = { Dot(delta, box.U[0]), Dot(delta, box.U[1]), Dot(delta, box.U[2]);
    Vector3 D = { Dot(ray.D, box.U[0]), Dot(ray.D, box.U[1]), Dot(ray.D, box.U[2]);

    return DoRayQuery(P, D, box.e);
}
```

The GTE files IntrRay3AlignedBox3.h and IntrRay3OrientedBox3.h implement this design.

2.3 Segments and Boxes

The segment is represented in center-direction-extent form, $\mathbf{P} + t\mathbf{D}$, where $\mathbf{P}$ is the center of the segment, $\mathbf{D}$ is a unit-length direction vector, ε is the extent (radius) of the segment and $|t| \le \varepsilon$. The line segment has length 2ε. The semi-infinite convex polyhedron that corresponds to the Minkowski difference of a ray and an oriented box becomes a finite object in the case of a segment and an oriented box. Figure 4 illustrates the process in two dimensions.

Figure 4. An oriented box and a segment. The oriented box is extruded along the segment in the direction $\mathbf{D}$. The faces F_0 and F_1 are generated by oriented box edges and $\mathbf{D}$. The faces F_2 and F_3 are contributed from the oriented box at one endpoint; the faces F_4 and F_5 are contributed from the oriented box at the other endpoint.

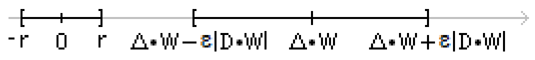


The infinite convex polyhedron has the same three pairs of parallel faces as for the ray and line, but the polyhedron has the oriented box as an end cap on both ends of the segment. The oriented box contributes three additional faces and corresponding normal vectors, a total of six potential separating axes. The separation tests are

$$\begin{aligned}
&|\mathbf{D} \times \mathbf{\Delta} \cdot \mathbf{U}_0| > e_1|\mathbf{D} \cdot \mathbf{U}_2| + e_2|\mathbf{D} \cdot \mathbf{U}_1| \\
&|\mathbf{D} \times \mathbf{\Delta} \cdot \mathbf{U}_1| > e_0|\mathbf{D} \cdot \mathbf{U}_2| + e_2|\mathbf{D} \cdot \mathbf{U}_0| \\
&|\mathbf{D} \times \mathbf{\Delta} \cdot \mathbf{U}_2| > e_0|\mathbf{D} \cdot \mathbf{U}_1| + e_1|\mathbf{D} \cdot \mathbf{U}_0| \\
&|\mathbf{\Delta} \cdot \mathbf{U}_0| > e_0 + \varepsilon|\mathbf{D} \cdot \mathbf{U}_0| \\
&|\mathbf{\Delta} \cdot \mathbf{U}_1| > e_1 + \varepsilon|\mathbf{D} \cdot \mathbf{U}_1| \\
&|\mathbf{\Delta} \cdot \mathbf{U}_2| > e_2 + \varepsilon|\mathbf{D} \cdot \mathbf{U}_1|
\end{aligned} \tag{3}$$

where ε is the extent of the segment. Figure 5 shows a typical separation of the projections on a separating axis with direction $\mathbf{W}$.

Figure 5. Projections of an oriented box and a segment onto a line with direction $\mathbf{W}$. The oriented box center $\mathbf{C}$ is subtracted from the oriented box as well as the segment origin $\mathbf{P}$. The translated oriented box projects to an interval $[-r, r]$. The translated segment is $\mathbf{\Delta} + t\mathbf{D}$, where $\mathbf{\Delta} = \mathbf{P} - \mathbf{C}$, and projects to $\mathbf{W} \cdot \mathbf{\Delta} + t\mathbf{W} \cdot \mathbf{D}$.



The intervals are separated when $\mathbf{W} \cdot \mathbf{\Delta} - \varepsilon|\mathbf{W} \cdot \mathbf{D}| > r$ or when $\mathbf{W} \cdot \mathbf{\Delta} + \varepsilon|\mathbf{W} \cdot \mathbf{D}| < -r$. These may be combined into a joint statement: $|\mathbf{W} \cdot \mathbf{\Delta}| > r + \varepsilon|\mathbf{W} \cdot \mathbf{D}|$.

The implementation of the test-intersection query for a segment and oriented box in 3 dimensions is shown in Listing 5.

Listing 5. Pseudocode for the test-intersection query between a segment and an oriented box in 3 dimensions. The function abs returns the absolute value of its scalar argument.

```
bool TestIntersection(Segment3 segment, OrientedBox3 box)
{
    Vector3 PmC = segment.P - box.C;
    Real ADdU[3], PmCdU[3], ADxPmCdU[3];

    // The segment-specific tests.
    for (int i = 0; i < 3; ++i)
    {
        ADdU[i] = abs(Dot(segment.D, box.U[i]));
        PmCdU[i] = abs(Dot(PmC, box.U[i]));
        if (PmCdU[i] > box.e[i] + segment.e * ADdU[i])
        {
            return false;
        }
    }

    // The line-specific tests.
    Vector3 DxPmC = Cross(segment.D, PmC);

    ADxPmCdU[0] = abs(Dot(DxPmC, box.U[0]));
    if (ADxPmCdU[0] > box.e[1] * ADdU[2] + box.e[2] * ADdU[1])
    {
        return false;
    }
    ADxPmCdU[1] = abs(Dot(DxPmC, box.U[1]));
    if (ADxPmCdU[1] > box.e[0] * ADdU[2] + box.e[2] * ADdU[0])
    {
        return false;
    }

    ADxPmCdU[2] = abs(Dot(DxPmC, box.U[2]));
    if (ADxPmCdU[2] > box.e[0] * ADdU[1] + box.e[1] * ADdU[0])
    {
        return false;
    }

    return true;
}
```

We can formulate the test-intersection for an aligned box by using the center-extent form of the box. Listing 6 contains pseudocode for the query.

Listing 6. Pseudocode for the test-intersection query between a segment and an aligned box in 3 dimensions.

```
// P is the segment center, D is the unit-length segment direction, segE is the
// segment extent and boxE contains the box extents.  The box center has been
// translated to the origin and the segment has been translated accordingly.
bool DoSegmentQuery(Vector3 P, Vector3 D, Real segE, Vector3 boxE)
{
    for (int i = 0; i < 3; ++i)
    {
        if (abs(P[i]) > segE[i] + r * abs(D[i]))
        {
            return false;
        }
    }
    return DoLineQuery(P, D, boxE);
}

// The query for the segment and aligned box, where the Segment3 object stores
// segment endpoints P0 and P1.
bool TestIntersection(Segment3 segment, AlignedBox3 box)
{
    // Get the centered form of the aligned box.
    Vector3 C = (box.max + box.min) / 2;
    Vector3 boxE = (box.max - box.min) / 2;

    // Transform the segment to the aligned-box coordinate system and convert
    // it to the center-direction-extent form.
    Vector3 P = (segment.P0 + segment.P1) / 2 - C;
    Vector3 D = segment.P1 - segment.P0;
    Real segE = Normalize(D);  // Normalize makes D unit length and returns Length(D)/2

    return DoSegmentQuery(P, D, segE, boxE);
}

// The query for the segment and oriented box but reformulated to use DoSegmentQuery.
bool TestIntersection(Segment3 segment, OrientedBox3 box)
{
    // Transform the segment to the oriented-box coordinate system.
    Vector3 P0 = { Dot(segment.P0, box.U[0]), Dot(segment.P0, box.U[1]), Dot(segment.P0, box.U[2]);
    Vector3 P1 = { Dot(segment.P1, box.U[0]), Dot(segment.P1, box.U[1]), Dot(segment.P1, box.U[2]);
    Vector3 P = (P0 + P1) / 2 - box.C;
    Vector3 D = P1 - P0;
    Real segE = Normalize(D);  // Normalize makes D unit length and returns Length(D)/2
    D = { Dot(D, box.U[0]), Dot(D, box.U[1]), Dot(D, box.U[2]);

    return DoSegmentQuery(P, D, segE, box.e);
}
```

The GTE files IntrSegment3AlignedBox3.h and IntrSegment3OrientedBox3.h implement this design.

3 Find-Intersection Query

The line-box find-intersection query is based on Liang–Barsky clipping [2, 1] of parametric lines against the box faces one at a time. The idea is to start with a t interval $[t_0, t_1]$ representing the current linear component (line, ray or segment). Initially, the interval is infinite: $t_0 = -\infty$ and $t_1 = \infty$. The line is converted to the box coordinate system. The line origin $\mathbf{P}$ is mapped to $\mathbf{P}' = (x_p, y_p, z_p)$ in the box coordinate system via

$$\mathbf{P} = \mathbf{C} + x_p\mathbf{U}_0 + y_p\mathbf{U}_1 + z_p\mathbf{U}_2 \tag{4}$$

Thus, $x_p = \mathbf{U}_0 \cdot (\mathbf{P} - \mathbf{C})$, $y_p = \mathbf{u}_1 \cdot (\mathbf{P} - \mathbf{C})$, and $z_p = \mathbf{u}_2 \cdot (\mathbf{P} - \mathbf{C})$. The line direction $\mathbf{D}$ is mapped to $\mathbf{D}' = (x_d, y_d, z_d)$ in the box coordinate system via

$$\mathbf{D} = x_d\mathbf{U}_0 + y_d\mathbf{U}_1 + z_d\mathbf{U}_2 \tag{5}$$

Thus, $x_d = \mathbf{U}_0 \cdot \mathbf{D}$, $y_p = \mathbf{U}_1 \cdot \mathbf{D}$, and $z_d = \mathbf{U}_2 \cdot \mathbf{D}$. In the box coordinate system, the box is naturally axis aligned. If (x, y, z) is a box point, then $|x| \le e_0$, $|y| \le e_1$, and $|z| \le e_2$.

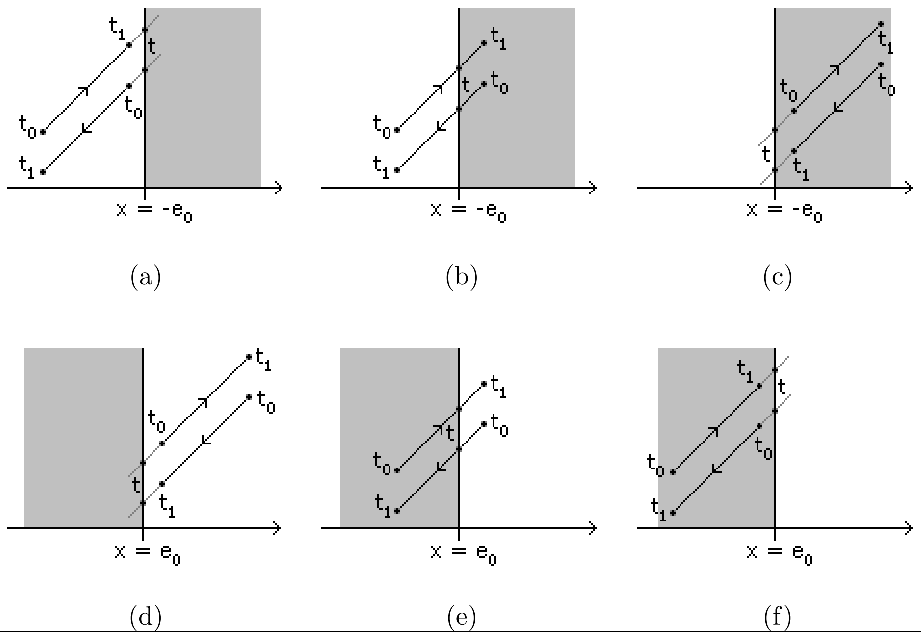
3.1 Liang–Barsky Clipping

Figure 6 illustrates the clipping of the line $(x_p, y_p, z_p) + t(x_d, y_d, z_d)$ against the x-faces of the box. The top three images in the figure show clipping against the face $x = -e_0$, and the bottom three images show clipping against the face $x = e_0$. The clipping algorithm depends on the orientation of the line's direction vector relative to the x-axis. For a plane $x = a$, the intersection point of the line and plane is determined by

$$x_p + tx_d = a \tag{6}$$

We need to solve for t. The three cases are $x_d > 0$, $x_d < 0$, and $x_d = 0$.

Figure 6. Line clipping against the x-faces of the box. The cases are referenced in the pseudocode that occurs later in this section.



Consider $x_d > 0$. In Figure 6, the relevant linear components are those with the arrow showing the components pointing generally in the positive x-direction. The t-value for the intersection is $t = (-e_0 - x_p)/x_d$. The decision on clipping is illustrated in the pseudocode

```
if (t > t1) then
    cull the linear component;   // Figure 6 (a)
else if (t > t0) then
    t0 = t;                      // Figure 6 (b)
else
    do nothing;                  // Figure 6 (c)
```

If $x_d < 0$, the relevant linear components in Figure 6 are those with the arrow showing the components pointing generally in the negative x direction. The pseudocode is

```
if (t < t0) then
    cull the linear component;   // Figure 6 (a)
else if (t < t1) then
    t1 = t;                      // Figure 6 (b)
else
    do nothing;                  // Figure 6 (c)
```

Finally, if $x_d = 0$, the linear component is parallel to the x-faces. The component is either outside the box, in which case it is culled, or inside or on the box, in which case no clipping against the x faces needs to be performed. The pseudocode is shown in Listing

```
n = -e0 - xp;
if (n > 0) then
    cull the linear component;
else
    do nothing;
```

A similar construction applies for the face $x = e_0$, except that the sense of direction is reversed. The bottom three images of Figure 6 apply here. The equation to solve is $x_p + tx_d = e_0$. If $x_d \neq 0$, the solution is $t = (e_0 - x_p)/t_d$. If $x_d > 0$, the pseudocode is

```
if (t < t0) then
    cull the linear component;   // Figure 6 (d)
else if (t < t1) then
    t1 = t;                      // Figure 6 (e)
else
    do nothing;                  // Figure 6 (f)
```

If $x_d < 0$, the pseudocode is

```
if (t > t1) then
    cull the linear component;   // Figure 6 (d)
else if (t > t0) then
    t0 = t;                      // Figure 6 (e)
else
    do nothing;                  // Figure 6 (f)
```

If $x_d = 0$, the linear component is parallel to the x-faces. The component is either outside the box, in which case it is culled, or inside or on the box, in which case no clipping against the x-faces needs to be performed. The pseudocode is

```
n = e0 - xp;
if (n < 0) then
    cull the linear component;
else
    do nothing;
```

A goal of the construction is to keep the clipping code to a minimum. With this in mind, notice that the clipping code for the case $x = e_0$ and $x_d > 0$ is the same as that for the case $x = -e_0$ and $x_d < 0$. The clipping code for the case $x = e_0$ and $x_d < 0$ is the same as that for the case $x = -e_0$ and $x_d > 0$. The cases when $x_d = 0$ differ only by the test of n—in one case tested for positivity, in the other for negativity. We may consolidate the six code blocks into three by processing the $x = e_0$ cases using $-x_d$ and $-n = x_p - e_0$. The pseudocode of Listing 7 shows the consolidated code. One optimization occurs: a division is performed to compute t only if the linear component is not culled.

Listing 7. Liang–Barsky clipping for a linear component against a face of a box.

```
bool Clip(Real denom, Real numer, Real& t0, Real& t1)
{
    if (denom > 0)
    {
        if (numer > denom * t1)
        {
            return false;
        }
        if (numer > denom * t0)
        {
            t0 = numer / denom;
        }
        return true;
    }
    else if (denom < 0)
    {
        if (numer > denom * t0)
        {
            return false;
        }
        if (numer > denom * t1)
        {
            t1 = numer / denom;
        }
        return true;
    }
    else
    {
        return numer <= 0;
    }
}
```

The return value is false if the current linear component is culled; otherwise, the return value is true, indicating that the linear component was clipped or just kept as is.

3.2 Lines and Boxes

The clipping against all the faces is encapsulated by the find-intersection queries for a line and a box and is shown in Listing 8. As in the test-intersection queries, the code is written for aligned boxes. For oriented boxes, a transformation is applied to convert to a configuration involving aligned boxes after which the query for the aligned box is applied.

Listing 8. Clipping of a line against an aligned box or an oriented box. The returned numPoints is 0 if the line does not intersect the box (entirely clipped), 1 if the line intersects the box in a single point or 2 if the line intersects the box in a segment. The parameter interval is filled with 2 elements regardless of the value of numPoints. This ensures that we do not have uninitialized memory.

```
// P is the line origin, D is the unit-length line direction, and e contains
// the box extents.  The box center has been translated to the origin and
// the line has been translated accordingly.
void DoLineQuery(Vector3 P, Vector3 D, Vector3 e, int& numPoints, Real t[2])
{
    // Clip the line against the box faces using the 6 cases mentioned in Figure 6.
    Real t0 = -infinity, t1 = +infinity;
    bool notCulled =
        Clip(+D[0], -P[0] - box.e[0], t0, t1) &&
        Clip(-D[0], +P[0] - box.e[0], t0, t1) &&
        Clip(+D[1], -P[1] - box.e[1], t0, t1) &&
        Clip(-D[1], +P[1] - box.e[1], t0, t1) &&
        Clip(+D[2], -P[2] - box.e[2], t0, t1) &&
        Clip(-D[2], +P[2] - box.e[2], t0, t1);

    if (notCulled)
    {
        if (t1 > t0)
        {
            // The intersection is a segment P + t * D with t in [t0,t1].
            numPoints = 2;
            t[0] = t0;
            t[1] = t1;
        }
        else
        {
            // The intersection is a point P + t * D with t = t0.
            numPoints = 1;
            t[0] = t0;
            t[1] = t0;
        }
    }
    else
    {
        // The line does not intersect the box.  Return invalid parameters.
        numPoints = 0;
        t[0] = +infinity;
        t[1] = -infinity;
    }
}

// The line-box find-intersection query computes both the parameter interval t[]
// and the corresponding endpoints (if any) of the intersection linear component.
bool FindIntersection(Line3 line, AlignedBox3 box, int& numPoints, Real t[2], Vector3 point[2])
{
    // Get the centered form of the aligned box.
    Vector3 C = (box.max + box.min) / 2;
    Vector3 e = (box.max - box.min) / 2;

    // Transform the line to the aligned-box coordinate system.
    Vector3 P = line.P - C;
    Vector3 D = line.D;

    DoLineQuery(P, D, e, numPoints, t);
    for (int i = 0; i < numPoints; ++i)
    {
        point[i] = line.P + t[i] * line.D;
    }

    return numPoints > 0;
}

// The query for the line and oriented box is formulated to use DoLineQuery.
bool FindIntersection(Line3 line, OrientedBox3 box, int& numPoints, Real t[2], Vector3 point[2])
{
    // Transform the line to the oriented-box coordinate system.
    Vector3 delta = line.P - box.C;
    Vector3 P = { Dot(delta, box.U[0]), Dot(delta, box.U[1]), Dot(delta, box.U[2]);
    Vector3 D = { Dot(line.D, box.U[0]), Dot(line.D, box.U[1]), Dot(line.D, box.U[2]);

    DoLineQuery(P, D, e, numPoints, t);
    for (int i = 0; i < numPoints; ++i)
    {
        point[i] = line.P + t[i] * line.D;
    }

    return numPoints > 0;
}
```

The GTE files IntrLine3AlignedBox3.h and IntrLine3OrientedBox3.h implement this design.

3.3 Rays and Boxes

The find-intersection query between a ray and a box uses the line-box query to produce an intersection interval $I = [t_0, t_1]$ (possibly empty). The ray-box intersection is generated by the intersection of I with the ray interval $[0, +\infty)$; that is, it is generated by $I \cap [0, +\infty)$. Listing 9 contains pseudocode for the query.

Listing 9. Clipping of a ray against an aligned box or an oriented box. The returned numPoints is 0 if the ray does not intersect the box (entirely clipped), 1 if the ray intersects the box in a single point or 2 if the ray intersects the box in a segment. The parameter interval is filled accordingly with numPoints elements.

```
// P is the ray origin, D is the unit-length ray direction, and e contains
// the box extents.  The box center has been translated to the origin and
// the ray has been translated accordingly.
void DoRayQuery(Vector3 P, Vector3 D, Vector3 e, int& numPoints, Real t[2])
{
    DoLineQuery(P, D, e, numPoints, t);
    if (numPoints > 0)
    {
        // The line containing the ray intersects the box in the interval
        // [t0,t1].  Compute the intersection of [t0,t1] and [0,+infinity).
        if (t[1] >= 0)
        {
            if (t[0] < 0)
            {
                t[0] = 0;
            }
        }
        else
        {
            // The line intersects the box but the ray does not.
            numPoints = 0;
        }
    }
}

// The ray-box find-intersection query computes both the parameter interval t[]
// and the corresponding endpoints (if any) of the intersection linear component.
bool FindIntersection(Ray3 ray, AlignedBox3 box, int& numPoints, Real t[2], Vector3 point[2])
{
    // Get the centered form of the aligned box.
    Vector3 C = (box.max + box.min) / 2;
    Vector3 e = (box.max - box.min) / 2;

    // Transform the ray to the aligned-box coordinate system.
    Vector3 P = ray.P - C;
    Vector3 D = ray.D;

    DoRayQuery(P, D, e, numPoints, t);
    for (int i = 0; i < numPoints; ++i)
    {
        point[i] = ray.P + t[i] * ray.D;
    }

    return numPoints > 0;
}

// The query for the ray and oriented box is formulated to use DoLineQuery.
bool FindIntersection(Ray3 ray, OrientedBox3 box, int& numPoints, Real t[2], Vector3 point[2])
{
    // Transform the ray to the oriented-box coordinate system.
    Vector3 delta = ray.P - box.C;
    Vector3 P = { Dot(delta, box.U[0]), Dot(delta, box.U[1]), Dot(delta, box.U[2]);
    Vector3 D = { Dot(line.D, box.U[0]), Dot(line.D, box.U[1]), Dot(line.D, box.U[2]);

    DoRayQuery(P, D, e, numPoints, t);
    for (int i = 0; i < numPoints; ++i)
    {
        point[i] = ray.P + t[i] * ray.D;
    }

    return numPoints > 0;
}
```

The GTE files IntrRay3AlignedBox3.h and IntrRay3OrientedBox3.h implement this design.

3.4 Segments and Boxes

The segment is represented in center-direction-extent form, $\mathbf{P} + t\mathbf{D}$, where $\mathbf{P}$ is the center of the segment, $\mathbf{D}$ is a unit-length direction vector, ε is the extent (radius) of the segment and $|t| \leq \varepsilon$. The line segment has length 2ε.

The find-intersection query between a segment and a box uses the line-box query to produce an intersection interval $I = [t_0, t_1]$ (possibly empty). The segment-box intersection is generated by the intersection of I with the segment interval $[s_0, s_1]$; that is, it is generated by $I \cap [s_0, s_1]$. Listing 10 contains pseudocode for the query.

Listing 10. Clipping of a segment against an aligned box or an oriented box. The returned numPoints is 0 if the segment does not intersect the box (entirely clipped), 1 if the segment intersects the box in a single point or 2 if the segment intersects the box in a segment. The parameter interval is filled accordingly with numPoints elements.

```
// Pseudocode for computing the intersection of two intervals.
bool ComputeIntersection(Real interval0[2], Real interval1[2], int& numIntersections, Real overlap[2])
{
    if (interval0[1] < interval1[0] || interval0[0] > interval1[1])
    {
        overlap[0] = +infinity;
        overlap[1] = -infinity;
        numIntersections = 0;
    }
    else if (interval0[1] > interval1[0])
    {
        if (interval0[0] < interval1[1])
        {
            overlap[0] = (interval0[0] < interval1[0] ? interval1[0] : interval0[0]);
            overlap[1] = (interval0[1] > interval1[1] ? interval1[1] : interval0[1]);
            numIntersections = (overlap[0] != overlap[1] ? 2 : 1);
        }
        else  // interval0[0] == interval1[1]
        {
            overlap[0] = interval0[0];
            overlap[1] = overlap[0];
            numIntersections = 1;
        }
    }
    else  // interval0[1] == interval1[0]
    {
        overlap[0] = interval0[1];
        overlap[1] = overlap[0];
        numIntersections = 1;
    }
    return numIntersections > 0;
}

// P is the segment center, D is the unit-length segment direction, segE is the
// segment extent and boxE contains the box extents.  The box center has been
// translated to the origin and the segment has been translated accordingly.
void DoSegmentQuery(Vector3 P, Vector3 D, Real segE, Vector3 boxE, int& numPoints, Real t[2])
{
    DoLineQuery(P, D, boxE, numPoints, t);
    if (numPoints > 0)
    {
        // The line containing the segment intersects the box in the interval
        // [t0,t1]. Compute the intersection of [t0,t1] and the segment
        // interval [s0,s1].
        Real s[2] = { -segE, segE }, overlap[2];
        if (ComputeIntersection(t, s, numPoints, overlap)
        {
            for (int i = 0; i < numPoints; ++i)
            {
                t[i] = overlap[i];
            }
        }
    }
}

// The segment-box find-intersection query computes both the parameter interval t[]
// and the corresponding endpoints (if any) of the intersection linear component.
// The query assumes the Segment3 object stores the segment endpoints P0 and P1.
bool FindIntersection(Segment3 segment, AlignedBox3 box, int& numPoints, Real t[2], Vector3 point[2])
{
    // Get the centered form of the aligned box.
    Vector3 C = (box.max + box.min) / 2;
    Vector3 boxE = (box.max - box.min) / 2;

    // Transform the segment to the aligned-box coordinate system and convert
    // it to the center-direction-extent form.
    Vector3 P = (segment.P0 + segment.P1) / 2 - C;
    Vector3 D = segment.P1 - segment.P0;
    Real segE = Normalize(D);  // Normalize makes D unit length and returns Length(D)/2

    DoSegmentQuery(P, D, segE, boxE, numPoints, t);

    // The segment is in aligned box coordinates, transform back to the original space.
    for (int i = 0; i < numPoints; ++i)
    {
        point[i] = P + C + t[i] * D;
    }

    return numPoints > 0;
}

// The query for the segment and oriented box is formulated to use DoSegmentQuery.
bool FindIntersection(Segment3 segment, OrientedBox3 box, int& numPoints, Real t[2], Vector3 point[2])
{
    // Transform the segment to the oriented-box coordinate system.
    Vector3 P0 = { Dot(segment.P0, box.U[0]), Dot(segment.P0, box.U[1]), Dot(segment.P0, box.U[2]);
    Vector3 P1 = { Dot(segment.P1, box.U[0]), Dot(segment.P1, box.U[1]), Dot(segment.P1, box.U[2]);
    Vector3 P = (P0 + P1) / 2 - box.C;
    Vector3 D = P1 - P0;
    Real segE = Normalize(D);  // Normalize makes D unit length and returns Length(D)/2
   D = { Dot(D, box.U[0]), Dot(D, box.U[1]), Dot(D, box.U[2]);

    DoSegmentQuery(P, D, segE, box.e, numPoints, t);

    // The segment is in aligned box coordinates, transform back to the original space.
    for (int i = 0; i < numPoints; ++i)
    {
        // Compute the intersection point in the oriented-box coordinate system.
        Vector3 Y = P + t[i] * D;

        // Transform the intersection point to the original coordinate system.
        point[i] = box.C + Y[0] * box.U[0] + Y[1] * box.U[1] + Y[2] * box.U[2];
    }

    return numPoints > 0;
}
```

The GTE files IntrSegment3AlignedBox3.h and IntrSegment3OrientedBox3.h implement this design.

References

[1] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, Reading, MA, 1990.

[2] Y-D. Liang and B. A. Barsky. A new concept and method for line clipping. *ACM Transactions on Graphics*, 3(1):1–22, 1984.