Intersection of Moving Circle and Rectangle

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

Created: July 31, 2018
Last Modified: September 11, 2020

Contents

1 Introduction

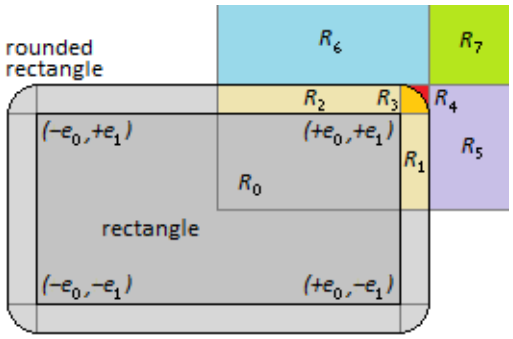
This document shows how to compute whether a circle and rectangle intersect, both moving with constant velocities. Both objects are considered to be solids; that is, the circle object is the region bounded by the circle and the rectangle object is the region bounded by the rectangle. Throughout the document I will continue to use the terms circle and rectangle for these objects.

It is possible that the circle and rectangle initially overlap in the sense that the common set of points is nonempty. If that set contains a single point, then the circle and rectangle are in contact at time zero. If they are not initially overlapping and if they will intersect at a later time, the first time of contact and the corresponding point of contact are computed. The remaining possibility is that the objects will not intersect at any time.

The circle has center $\mathbf{C}_{\rm cir}$, radius r and moves with constant velocity $\mathbf{V}_{\rm cir}$. The rectangle has center $\mathbf{C}_{\rm rct}$, unit-length axis directions $\mathbf{U}_0$ and $\mathbf{U}_1$ that are perpendicular and extents e_0 and e_1 in those directions. The four corners of the rectangle are $\mathbf{C}_{\rm rct} \pm e_0\mathbf{U}_0 \pm e_1\mathbf{U}_1$. The rectangle moves with constant velocity $\mathbf{V}_{\rm rct}$. The problem can be converted to a canonical form for the determination of first contact time and point, if they exist. Consider motion relative to the rectangle: the rectangle is stationary (zero velocity) and the circle moves with velocity $\mathbf{V}_{\rm rel} = \mathbf{V}_{\rm cir} - \mathbf{V}_{\rm rct}$. Now transform the rectangle to an axis-aligned rectangle centered at the origin and having axis directions $(1, 0)$ and $(0, 1)$. Define $R = [\mathbf{U}_0\,\mathbf{U}_1]$, which is a rotation matrix with the specified columns. The transformation of the rectangle to an axis-aligned rectangle causes the circle center to be transformed to $\mathbf{C} = R^{\rm T}(\mathbf{C}_{\rm cir} - \mathbf{C}_{\rm rct})$. The circle velocity is transformed to $\mathbf{V} = R^{\rm T}\mathbf{V}_{\rm rel} = R^{\rm T}(\mathbf{V}_{\rm cir} - \mathbf{V}_{\rm rct})$.

When the circle and rectangle are initially separated but will later intersect at a first contact time, they do so at a single point that is either on a rectangle edge or at a rectangle corner. To obtain a better geometric flavor of the problem, we can consider the equivalent problem involving Minkowski sums: the circle is reduced to a point (its center) and the rectangle is expanded by placing circles of radii r at each point of the rectangle and computing the union. Figure 1 shows the resulting expansion which is a rectangle with rounded corners. I will refer to the solid region as the rounded rectangle.

Figure 1. The rounded rectangle obtained is drawn in shades of gray, tan and and orange. The rectangle has black edges with the corners labeled by their coordinates. The distance between the edges of the rectangle and the edges of the rounded rectangle is r. The quarter circles centered at the corners have common radius r. The colored regions are described later in this document. For clarity, $\mathcal{R}_3$ refers to the orange region and $\mathcal{R}_4$ refers to the red region.



The center travels along the ray $\mathbf{C} + t\mathbf{V}$. If this ray intersects the rounded rectangle at a first contact time

$T \geq 0$, then the circle and rectangle intersect at that time at the first contact point. The point $\mathbf{C} + T\mathbf{V}$ is the center of the circle at first contact, and the first contact point can be computed from it. The details of this are provided in the remainder of the document.

2 Reduction to a Canonical Form

After the transformation that maps the rectangle to an axis-aligned rectangle with center at the origin, we can perform an additional transformation that maps $\mathbf{C}$ to the first quadrant and maps $\mathbf{V}$ accordingly. Specifically, any component of $\mathbf{C}$ that is negative can be negated and the corresponding component of $\mathbf{V}$ is negated. I will use the same names for center and velocity from this point on. Listing 1 contains pseudocode for converting the circle and rectangle to the canonical representation that allows intersection analysis in the first quadrant.

Listing 1. Conversion of the circle and rectangle to canonical form, analysis for intersections and then conversion back to the original coordinate system.

```
struct Circle { Vector2 center; Real radius; };
struct Rectangle { Vector2 center; Vector2 axis[2]; Real extent[2]; };

// The returned integer is -1 if the objects initially overlap, 0 if the
// objects are initially separated but do not intersect at a later time,
// or +1 if the objects are initially separated but do intersect at a later
// time.  In the last case, contactTime and contactPoint are valid quantities.
int GetFirstTimeAndContact(Circle circle, Vector2 circleVelocity,
    Rectangle rectangle, Vector2 rectangleVelocity, Real& contactTime, Vector2& contactPoint)
{
    // Transform the circle and velocity to be in the coordinate system of the
    // aligned rectangle centered at the origin.
    Vector2 cdiff = circle.center - rectangle.center;
    Vector2 vdiff = circleVelocity - rectangleVelocity;
    Vector2 C, V;
    for (int i = 0; i < 2; ++i)
    {
        C[i] = Dot(cdiff, rectangle.axis[i]);
        V[i] = Dot(vdiff, rectangle.axis[i]);
    }

    // Reflect components, if necessary, to transform C to the first quadrant.
    // Adjust the velocity accordingly.
    int sign[2];
    for (int i = 0; i < 2; ++i)
    {
        if (C[i] >= 0)
        {
            sign[i] = 1;
        }
        else
        {
            C[i] = -C[i];
            V[i] = -V[i];
            sign[i] = -1;
        }
    }

    int intersectionType = DoQuery(rectangle.extent, C, circle.radius, V, contactTime, contactPoint);

    if (intersectionType != 0)
    {
        // Translate back to the original coordinate system.
        for (int i = 0; i < 2; ++i)
        {
```

```
            contactPoint[i] = rectangle.center[i] + sign[i] * contactPoint[i] * rectangle.axis[i];
        }
    }
    return intersectionType;
}
```

After the canonical transformation, $\mathbf{C}$ is in one of 8 regions as shown in Figure 1.

Regions $\mathcal{R}_0$ through $\mathcal{R}_3$ are inside the rounded rectangle. If $\mathbf{C}$ is in one of these regions, the circle and rectangle are initially overlapping. Generally, one expects that the intersection of circle and rectangle is a set of positive area, although there is a small probability that the circle and rectangle are touching at a single point. Regardless, in both cases the implementation reports that the contact time is $T = 0$. The partitioning into the 4 regions allows us to select a point in the intersection, even if it is not considered a contact point.

If the initial center is outside the rounded rectangle, the circle might or might not intersect the rectangle depending on the circle velocity. The analysis is based on visibility of the rounded rectangle boundary components from $\mathbf{C}$. For the entire rounded rectangle, 4 of these components are line segments and 4 of these components are quarter circles centered at the corners. The latter components are referred to as *corner arcs*.

Region $\mathcal{R}_4$ is the region outside the quarter circle at vertex (e_0, e_1) but inside the smallest aligned rectangle containing the rounded rectangle. The upper-right vertex of this bounding rectangle is (e_0+r, e_1+r). Figure 1 shows $\mathcal{R}_4$ drawn in red. If $\mathbf{C} \in \mathcal{R}_4$, then only the corner arc centered at (e_0, e_1) is visible to $\mathbf{C}$. We can construct rays emanating from $\mathbf{C}$ that intersect the corner arc. Two of those rays are tangent to the corner arc. The rays between these, including the tangent rays, form a *2D cone* of rays. Any vector in the direction of a cone ray is a velocity vector that leads to intersection of the cone ray with the corner arc and, therefore, to intersection of the circle and rectangle.

If $\mathbf{C} \in \mathcal{R}_5$, the right segment of the rounded rectangle is visible to $\mathbf{C}$. Each of the corner arcs centered at the endpoints of the right segment, namely, (e_0, e_1) and $(e_0, -e_1)$ has points visible to $\mathbf{C}$. We can construct tangent rays from the center to those corner arcs. The resulting 2D cone of rays is such that any vector in the direction of a cone ray is a velocity vector that leads to intersection of the cone ray with the corner arc and, therefore, to intersection of the circle and rectangle.

If $\mathbf{C} \in \mathcal{R}_6$, the top segment of the rounded rectangle is visible to $\mathbf{C}$. Each of the corner arcs centered at the endpoints of the top segment, namely, (e_0, e_1) and $(-e_0, e_1)$ has points visible to $\mathbf{C}$. We can construct tangent rays from the center to those corner arcs. The resulting 2D cone of rays is such that any vector in the direction of a cone ray is a velocity vector that leads to intersection of the cone ray with the corner arc and, therefore, to intersection of the circle and rectangle.

If $\mathbf{C} \in \mathcal{R}_7$, the top and right segments of the rounded rectangle are visible to $\mathbf{C}$. The corner arcs centered at $(-e_0, e_1)$ and $(e_0, -e_1)$ are partially visible and the corner arc at (e_0, e_1) is fully visible. We can construct tangent rays from the center to the corner arcs $(-e_0, e_1)$ and $(e_0, -e_1)$. The resulting 2D cone of rays is such that any vector in the direction of a cone ray is a velocity vector that leads to intersection of the cone ray with the corner arc and, therefore, to intersection of the circle and rectangle

3 Intersection Analysis

In all cases, the determination of intersection is based on testing whether the center velocity is inside a 2D cone. The tests can be formulated without explicitly computing the tangent-ray directions.

I assume that the objects have been transformed to canonical form for intersection analysis in the first quadrant, as described in the previous section. The center $\mathbf{C}$ is in the first quadrant. Listing 2 has pseudocode for determining which of the regions $\mathcal{R}_i$ contains $\mathbf{C}$. Define $\mathbf{\Delta} = \mathbf{C} - (e_0, e_1)$.

Listing 2. Pseudocode for determining which $\mathcal{R}_i$ contains $\mathbf{C}$.

```
int DoQuery(Vector2 K, Vector2 C, Real radius, Vector2 V,
    Real& contactTime, Real& contactPoint)
{
    Vector2 delta = C - K;
    if (delta[1] <= radius)
    {
        if (delta[0] <= radius)
        {
            if (delta[1] <= 0)
            {
                if (delta[0] <= 0)
                {
                    return InRegion0(C, contactTime, contactPoint);
                }
                else
                {
                    return InRegion1(K, C, delta, radius, contactTime, contactPoint);
                }
            }
            else
            {
                if (delta[0] <= 0)
                {
                    return InRegion2(K, C, delta, radius, contactTime, contactPoint);
                }
                else
                {
                    if (delta[0] * delta[0] + delta[1] * delta[1] <= radius * radius)
                    {
                        return InRegion3(K, delta, radius, contactTime, contactPoint);
                    }
                    else
                    {
                        return InRegion4(K, delta, radius, V, contactTime, contactPoint);
                    }
                }
            }
        }
        else
        {
            return InRegion5(K, C, delta, radius, V, contactTime, contactPoint);
        }
    }
    else
    {
        if (delta[0] <= radius)
        {
            return InRegion6(K, C, delta, radius, V, contactTime, contactPoint);
        }
        else
        {
            return InRegion7(K, C, delta, radius, V, contactTime, contactPoint);
        }
    }
}
```

3.1 Analysis When C is in $\mathcal{R}_0$

The circle center is inside the original rectangle. The intersection type is set to -1. The overlap has positive area, so there is no first contact point. However, we choose $\mathbf{C}$ as one of the intersection points and return it as the contact point. The contact time is 0. Listing 3 contains pseudocode for this case.

Listing 3. Intersection analysis when $\mathbf{C} \in \mathcal{R}_0$.

```
int InRegion0(Vector2 C, Real& contactTime, Vector2& contactPoint)
{
    contactTime = 0;
    contactPoint = C;
    return -1;
}
```

3.2 Analysis When C is in $\mathcal{R}_1$

The circle intersects the right edge of the original rectangle in at least one point. The intersection is a single point when $\mathbf{C}$ is r units from the right edge, in which case the contact point is (e_0, c_1). The overlap has positive area when $\mathbf{C}$ is $d < r$ units from the right edge. In this case, an intersection point (e_0, c_1) is reported as the first contact. The distinction between the two cases is represented by the intersection type (-1 or $+1$). The contact time is 0. Listing 4 contains pseudocode for this case.

Listing 4. Intersection analysis when $\mathbf{C} \in \mathcal{R}_1$.

```
int InRegion1(Vector2 K, Vector2 C, Vector2 delta, Real radius, Real& contactTime, Vector2& contactPoint)
{
    contactTime = 0;
    contactPoint = { K[0], C[1] };
    return (delta[0] < radius ? -1 : 1);
}
```

3.3 Analysis When C is in $\mathcal{R}_2$

The circle intersects the top edge of the original rectangle in at least one point. The intersection is a single point when $\mathbf{C}$ is r units from the top edge, in which case the contact point is (c_0, e_1). The overlap has positive area when $\mathbf{C}$ is $d < r$ units from the top edge. In this case, an intersection point (c_0, e_1) is reported as the first contact. The distinction between the two cases is represented by the intersection type (-1 or $+1$). The contact time is 0. Listing 5 contains pseudocode for this case.

Listing 5. Intersection analysis when $\mathbf{C} \in \mathcal{R}_2$.

```
int InRegion2(Vector2 K, Vector2 C, Vector2 delta, Real radius, Real& contactTime, Vector2& contactPoint)
{
    contactTime = 0;
    contactPoint = { C[0], K[1] };
    return (delta[1] < radius ? -1 : 1);
}
```

3.4 Analysis When C is in $\mathcal{R}_3$

The circle intersects the corner $\mathbf{K} = (e_0, e_1)$ of the original rectangle. The intersection is a single point when $\mathbf{C} = \mathbf{K}$, in which case $|\mathbf{C} - \mathbf{K}| = r$. The overap has positive area when $|\mathbf{C} - \mathbf{K}| < r$. In either case, $\mathbf{K}$ is reported as the contact point. The distinction between the two cases is represented by the intersection type (-1 or $+1$). The contact time is $t = 0$. Listing 6 contains pseudocode for this case.

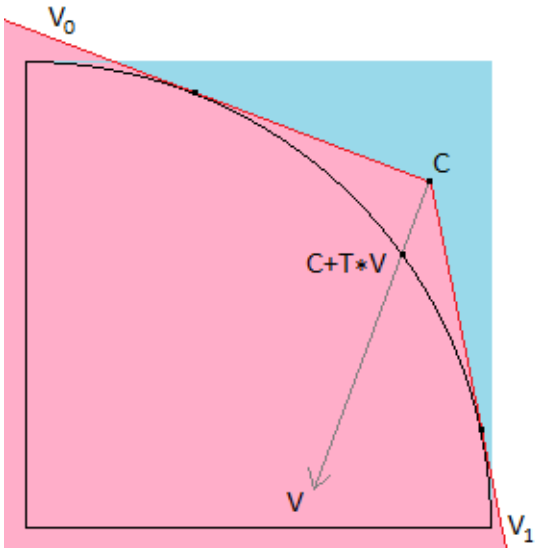
Listing 6. Intersection analysis when $\mathbf{C} \in \mathcal{R}_3$.

```
int InRegion3(Vector2 K, Vector2 delta, Real radius, Real& contactTime, Vector2& contactPoint)
{
    Real sqrDistance = delta[0] * delta[0] + delta[1] * delta[1];
    Real sqrRadius = radius * radius;
    contactTime = 0;
    contactPoint = K;
    return (sqrDistance < sqrRadius ? -1 : 1);
}
```

3.5 Analysis When C is in $\mathcal{R}_4$

The circle and rectangle are initially separated. The 2D cone of velocity vectors that lead to an intersection in positive time is shown in Figure 2. The tangent rays to the cone have directions $\mathbf{V}_0$ and $\mathbf{V}_1$.

Figure 2. An illustration of the 2D cone of velocity vectors for $\mathbf{C} \in \mathcal{R}_4$.



The cone can be computed indirectly by ray-circle intersection. The ray is parameterized by $\mathbf{X}(t) = \mathbf{C} + t\mathbf{V}$ for $t \geq 0$. The circle is $|\mathbf{X} - \mathbf{K}|^2 = r^2$, where $\mathbf{K} = (e_0, e_1)$. Substituting the ray equation into the circle equation, we obtain a quadratic equation

$$a_2 t^2 + 2a_1 t + a_0 = |\mathbf{V}|^2 t^2 + 2(\mathbf{V} \cdot \mathbf{\Delta})t + |\mathbf{\Delta}|^2 - r^2 = 0 \tag{1}$$

where $\mathbf{\Delta} = \mathbf{C} - \mathbf{K}$. From the geometric configuration, we know that $a_2 > 0$ and $a_0 > 0$. The quadratic formula allows us to solve for the smallest real-valued root T (if it exists),

$$T = \frac{-a_1 - \sqrt{a_1^2 - a_2 a_0}}{a_2} \tag{2}$$

To have a real-valued root, we need $a_1^2 - a_2 a_0 \geq 0$. For the roots to be positive, we additionally need $a_1 < 0$. The geometric conditions for existence of the intersection are

$$0 > \mathbf{V} \cdot \mathbf{\Delta}, \;\; 0 \leq (\mathbf{V} \cdot \mathbf{\Delta})^2 - |\mathbf{V}|^2 \left(|\mathbf{\Delta}|^2 - r^2\right) = r^2 |\mathbf{V}|^2 - \left(\mathbf{V} \cdot \mathbf{\Delta}^{\perp}\right)^2 \tag{3}$$

where $(x, y)^{\perp} = (y, -x)$ and $(x_0, y_0) \cdot (x_1, y_1)^{\perp} = x_0 y_1 - x_1 y_0$. The 2D cone tangent rays occur when the argument of the square root in the quadratic formula is 0, which may be written as

$$\left(\frac{\mathbf{V}}{|\mathbf{V}|} \cdot \mathbf{\Delta}^{\perp}\right)^2 = r^2 \tag{4}$$

If $\mathbf{P}$ is the intersection of a tangent ray with the circle, this condition turns out to be geometrically equivalent to the orthogonality of $\mathbf{P} - \mathbf{K}$ and $\mathbf{V}$. For rays strictly inside the 2D cone, $(\mathbf{V} \cdot \mathbf{\Delta}^{\perp})^2 / |\mathbf{V}|^2 < r^2$. The contact point is $\mathbf{K}$ and the contact time is $T > 0$. Listing 7 contains pseudocode for this case.

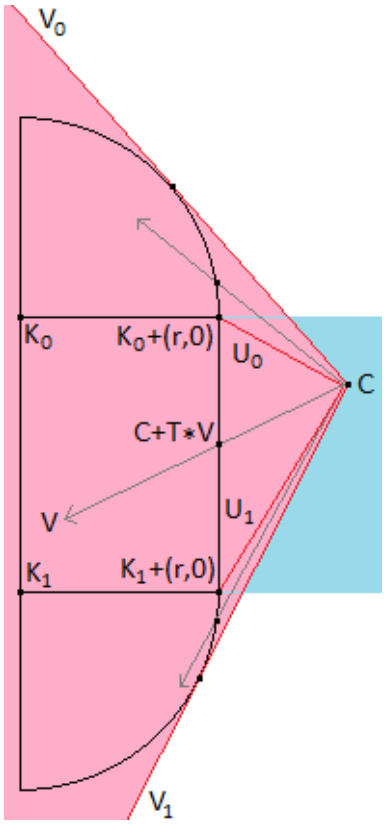
Listing 7. Intersection analysis when $\mathbf{C} \in \mathcal{R}_4$.

```
int InRegion4(Vector2 K0, Vector2 delta0, Real radius, Vector2 V, Real& contactTime, Real& contactPoint)
{
    Real negDotVDelta0 = -Dot(V, delta0);
    if (negDotVDelta0 > 0)
    {
        Real sqrRadius = radius * radius;
        Real dotVV = Dot(V, V);
        Real dotVPerpDelta0 = Dot(V, Perp(delta0));
        Real discr = sqrRadius * dotVV - dotVPerpDelta0 * dotVPerpDelta0;
        if (discr >= 0)
        {
            return IntersectsArc(K0, negDotVDelta0, discr, dotVV, contactTime, contactPoint);
        }
        else
        {
            // outside cone of valid vectors
            return NoIntersection(contactTime, contactPoint);
        }
    }
    else
    {
        // outside cone of valid vectors
        return NoIntersection(contactTime, contactPoint);
    }
}

int IntersectsArc(Vector2 K, Real q0, Real q1, Real q2, Real& contactTime, Vector2& contactPoint)
{
    contactTime = (q0 - sqrt(q1)) / q2;
    contactPoint = K;
    return +1;
}

int NoIntersection(Real& contactTime, Vector2& contactPoint)
{
    contactTime = 0;  // invalid
    contactPoint = Vector2(0, 0);  // invalid
    return 0;
}
```

3.6 Analysis When C is in $\mathcal{R}_5$

The circle and rectangle are initially separated. The 2D cone of velocity vectors that lead to an intersection in positive time is shown in Figure 3. The corner arcs for the right edge have centers $\mathbf{K}_0 = (e_0, e_1)$ and $\mathbf{K}_1 = (e_0, -e_1)$. The tangent rays have directions $\mathbf{V}_0$ and $\mathbf{V}_1$. Two other important vectors are shown, $\mathbf{U}_0 = \mathbf{K}_0 + (r, 0) - \mathbf{C}$ and $\mathbf{U}_1 = \mathbf{K}_1 + (r, 0) - \mathbf{C}$. In the discussion, the general velocity vector is $\mathbf{V} = (\nu_0, \nu_1)$.

Figure 3. An illustration of the 2D cone of velocity vectors for $\mathbf{C} \in \mathcal{R}_5$. Three different cone rays are shown, each having a direction parallel to a velocity vector that leads to intersection. The ray can intersect the top-right corner arc, the right edge or the bottom-right corner arc of the rounded rectangle.



The cone can be computed indirectly by ray-circle intersections. One tangent ray corresponds to the quarter circle centered at $\mathbf{K}_0$ and the other tangent ray corresponds to the quarter circle centered at $\mathbf{K}_1$. Define $\mathbf{\Delta}_0 = \mathbf{C} - \mathbf{K}_0$ and $\mathbf{\Delta}_1 = \mathbf{C} - \mathbf{K}_1$. Using the same quadratic equation approach as for $\mathcal{R}_4$, the corner arcs have tangent rays for which the velocity vector $\mathbf{V}_i$ must satisfy

$$\frac{\left(\mathbf{V}_i \cdot \mathbf{\Delta}_i^{\perp}\right)^2}{|\mathbf{V}_i|^2} = r^2 \tag{5}$$

for $i = 0$ and $i = 1$. All vectors $\mathbf{V}$ strictly in the cone bounded by these two vectors must satisfy

$$\frac{\left(\mathbf{V} \cdot \boldsymbol{\Delta}_i^{\perp}\right)^2}{|\mathbf{V}|^2} < r^2 \tag{6}$$

The other two vectors are $\mathbf{U}_i = (r, 0) - \boldsymbol{\Delta}_i$ for $i = 0$ and $i = 1$. The velocity vectors $\mathbf{V}$ in the subcone bounded by $\mathbf{V}_0$ and $\mathbf{U}_0$ lead to intersections of the circle with the rectangle corner $\mathbf{K}_0$. Those velocity vectors must satisfy

$$\mathbf{V} \cdot \boldsymbol{\Delta}_0 < 0, \;\; \frac{\left(\mathbf{V} \cdot \boldsymbol{\Delta}_0^{\perp}\right)^2}{|\mathbf{V}|^2} < r^2, \;\; \mathbf{V} \cdot \mathbf{U}_0^{\perp} \geq 0 \tag{7}$$

The first two conditions guarantee that the ray intersects the quarter circle at some positive time T, which is equivalent to the $\mathbf{V}/|\mathbf{V}|$ being a counterclockwise rotation of $\mathbf{V}_0/|\mathbf{V}_0|$. This is also equivalent to $\mathbf{V} \cdot \mathbf{V}_0^{\perp} \leq 0$, which requires explicitly computing a tangent vector $\mathbf{V}_0$, but the former test is less expensive to compute. The third condition says that $\mathbf{V}/|\mathbf{V}|$ is a clockwise rotation of $\mathbf{U}_0/|\mathbf{U}_0|$. A similar construction shows that the velocity vectors $\mathbf{V}$ in the subcone bounded by $\mathbf{U}_1$ and $\mathbf{V}_1$ are characterized by

$$\mathbf{V} \cdot \boldsymbol{\Delta}_1 < 0, \;\; \frac{\left(\mathbf{V} \cdot \boldsymbol{\Delta}_1^{\perp}\right)^2}{|\mathbf{V}|^2} < r^2, \;\; \mathbf{V} \cdot \mathbf{U}_1^{\perp} \leq 0 \tag{8}$$

The velocity vectors in the subcone bounded by $\mathbf{U}_0$ and $\mathbf{U}_1$ lead to intersections of the circle with points on the right edge of the rounded rectangle. The velocity vectors $\mathbf{V}$ for which this happens are characterized by

$$\mathbf{V} \cdot \mathbf{U}_0^{\perp} \leq 0, \;\; \mathbf{V} \cdot \mathbf{U}_1^{\perp} \geq 0 \tag{9}$$

The contact point and contact time are determined from the intersection of the ray and the right edge of the rounded rectangle. If $\mathbf{C} = (c_0, c_1)$ and $\mathbf{V} = (\nu_0, \nu_1)$, then the contact time is determined by the x-components, $c_0 + T\nu_0 = e_0 + r$. The contact time T and the contact point $\mathbf{P}$ are

$$T = \frac{e_0 + r - c_0}{\nu_0} > 0, \;\; \mathbf{P} = (e_0, c_1 + T\nu_1) \tag{10}$$

All velocity vectors not in the cone do not lead to an intersection. Those vectors $\mathbf{V}$ satisfy the conditions

$$\nu_0 \geq 0 \;\text{ or }\; \left(\nu_1 \geq 0 \;\text{ and }\; \frac{\left(\mathbf{V} \cdot \boldsymbol{\Delta}_0^{\perp}\right)^2}{|\mathbf{V}|^2} > r^2\right) \text{or} \;\; \left(\nu_1 \leq 0 \;\text{ and }\; \frac{\left(\mathbf{V} \cdot \boldsymbol{\Delta}_1^{\perp}\right)^2}{|\mathbf{V}|^2} > r^2\right) \tag{11}$$

The pseudocode effectively is a search of the subcones, looking for that subcone that contains a ray corresponding to a velocity vector $\mathbf{V}$. The subcones, listed as pairs of bounding vectors, and the conditions that define them are in Table 1.

Table 1. The subcones of the visibility cone for region $\mathcal{R}_5$. The left column shows the bounding vectors as endvectors of an interval that represents the angle constraints for the subcone. A vector $\mathbf{V}$ is in the a subcone when the right-column conditions are satisfied. Observe that those $\mathbf{V} = (\nu_0, \nu_1)$ leading to intersection must be in the second or third quadrants; that is, $\nu_0 < 0$. If $\mathbf{V}$ is in another quadrant, there is no intersection and therefore no subcone searching.

subcone interval	defining conditions
$[(0,-1), \mathbf{V}_1]$	$\nu_1 \le 0$ and $\left(\mathbf{V} \cdot \mathbf{\Delta}_1^\perp\right)^2 \ge r^2 \lvert\mathbf{V}\rvert^2$
$[\mathbf{V}_1, \mathbf{U}_1]$	$\left(\mathbf{V} \cdot \mathbf{\Delta}_1^\perp\right)^2 \le r^2 \lvert\mathbf{V}\rvert^2$ and $\mathbf{V} \cdot \mathbf{U}_1^\perp \le 0$
$[\mathbf{U}_1, \mathbf{U}_0]$	$\mathbf{V} \cdot \mathbf{U}_1^\perp \ge 0$ and $\mathbf{V} \cdot \mathbf{U}_0^\perp \le 0$
$[\mathbf{U}_0, \mathbf{V}_0]$	$\mathbf{V} \cdot \mathbf{U}_0^\perp \ge 0$ and $\left(\mathbf{V} \cdot \mathbf{\Delta}_0^\perp\right)^2 \le r^2 \lvert\mathbf{V}\rvert^2$
$[\mathbf{V}_0, (0,1)]$	$\left(\mathbf{V} \cdot \mathbf{\Delta}_0^\perp\right)^2 \ge r^2 \lvert\mathbf{V}\rvert^2$ and $\nu_0 \le 0$

Listing 8 contains pseudocode for this case.

Listing 8. Intersection analysis when $\mathbf{C} \in \mathcal{R}_5$.

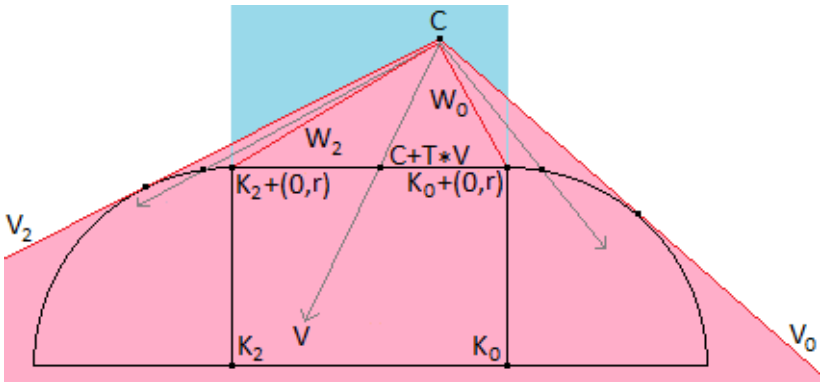
```
int InRegion5(Vector2 K0, Vector2 C, Vector2 delta0, Real radius, Vector2 V,
    Real& contactTime, Real& contactPoint)
{
    if (V[0] < 0)
    {
        Real dotVPerpDelta0 = Dot(V, Perp(delta0));
        Real dotVPerpU0 = -radius * V[1] - dotVPerpDelta0;
        if (dotVPerpU0 <= 0)
        {
            Vector2 K1(K0[0], -K0[1]);
            Vector2 delta1 = C - K1;
            Real dotVPerpDelta1 = Dot(V, Perp(delta1));
            Real dotVPerpU1 = -radius * V[1] - dotVPerpDelta1;
            if (dotVPerpU1 >= 0)
            {
                return IntersectsRightEdge(K0, C, radius, V, contactTime, contactPoint);
            }
            else
            {
                Real sqrRadius = radius * radius;
                Real dotVV = Dot(V, V);
                Real discr = sqrRadius * dotVV - dotVPerpDelta1 * dotVPerpDelta1;
                if (discr >= 0)
                {
                    return IntersectsArc(K1, -Dot(V, delta1), discr, dotVV, contactTime, contactPoint);
                }
                else  // outside V1-tangent ray
                {
                    return NoIntersection(contactTime, contactPoint);
                }
            }
        }
        else
        {
            Real sqrRadius = radius * radius;
            Real dotVV = Dot(V, V);
            Real discr = sqrRadius * dotVV - dotVPerpDelta0 * dotVPerpDelta0;
            if (discr >= 0)
            {
```

```
                return IntersectsArc(K0, -Dot(V, delta0), discr, dotVV, contactTime, contactPoint);
            }
            else  // outside V0-tangent ray
            {
                return NoIntersection(contactTime, contactPoint);
            }
        }
    }
    else  // circle moves away from rectangle
    {
        return NoIntersection(contactTime, contactPoint);
    }
}

int IntersectsRightEdge(Vector2 K0, Vector2 C, Real radius, Vector2 V, Real& contactTime, Vector2& contactPoint)
{
    contactTime = (K0[0] + radius - C[0]) / V[0];
    contactPoint = Vector2(K0[0], C[1] + contactTime * V[1]);
    return +1;
}
```

3.7 Analysis When C is in $\mathcal{R}_6$

The analysis for region $\mathcal{R}_6$ is similar to that of $\mathcal{R}_5$. The circle and rectangle are initially separated. The 2D cone of velocity vectors that lead to an intersection in positive time is shown in Figure 4. The corner arcs for the top edge have centers $\mathbf{K}_0 = (e_0, e_1)$ and $\mathbf{K}_2 = (-e_0, e_1)$. The tangent rays have directions $\mathbf{V}_0$ and $\mathbf{V}_2$. Two other important vectors are shown, $\mathbf{W}_0 = \mathbf{K}_0 + (0, r) - \mathbf{C}$ and $\mathbf{W}_2 = \mathbf{K}_2 + (0, r) - \mathbf{C}$. In the discussion, the general velocity vector is $\mathbf{V} = (\nu_0, \nu_1)$.

Figure 4. An illustration of the 2D cone of velocity vectors for $\mathbf{C} \in \mathcal{R}_6$. Three different cone rays are shown, each having a direction parallel to a velocity vector that leads to intersection. The ray can intersect the top-left corner arc, the top edge or the top-right corner arc of the rounded rectangle.



The cone can be computed indirectly by ray-circle intersections. One tangent ray corresponds to the quarter circle centered at $\mathbf{K}_0$ and the other tangent ray corresponds to the quarter circle centered at $\mathbf{K}_2$. Define $\mathbf{\Delta}_0 = \mathbf{C} - \mathbf{K}_0$ and $\mathbf{\Delta}_2 = \mathbf{C} - \mathbf{K}_2$. Using the same quadratic equation approach as for $\mathcal{R}_4$, the corner arcs

have tangent rays for which the velocity vector $\mathbf{V}_i$ must satisfy

$$\frac{\left(\mathbf{V}_i \cdot \mathbf{\Delta}_i^{\perp}\right)^2}{|\mathbf{V}_i|^2} = r^2 \tag{12}$$

for $i = 0$ and $i = 2$. All vectors $\mathbf{V}$ strictly in the cone bounded by these two vectors must satisfy

$$\frac{\left(\mathbf{V} \cdot \mathbf{\Delta}_i^{\perp}\right)^2}{|\mathbf{V}|^2} < r^2 \tag{13}$$

The other two vectors are $\mathbf{W}_i = (0, r) - \mathbf{\Delta}_i$ for $i = 0$ and $i = 2$. The velocity vectors $\mathbf{V}$ in the subcone bounded by $\mathbf{V}_0$ and $\mathbf{W}_0$ lead to intersections of the circle with the rectangle corner $\mathbf{K}_0$. Those velocity vectors must satisfy

$$\mathbf{V} \cdot \mathbf{\Delta}_0 < 0, \quad \frac{\left(\mathbf{V} \cdot \mathbf{\Delta}_0^{\perp}\right)^2}{|\mathbf{V}|^2} < r^2, \quad \mathbf{V} \cdot \mathbf{W}_0^{\perp} \le 0 \tag{14}$$

The first two conditions guarantee that the ray intersects the quarter circle at some positive time T, which is equivalent to the $\mathbf{V}/|\mathbf{V}|$ being a clockwise rotation of $\mathbf{V}_0/|\mathbf{V}_0|$. This is also equivalent to $\mathbf{V} \cdot \mathbf{V}_0^{\perp} \ge 0$, which requires explicitly computing a tangent vector $\mathbf{V}_0$, but the former test is less expensive to compute. The third condition says that $\mathbf{V}/|\mathbf{V}|$ is a counterclockwise rotation of $\mathbf{W}_0/|\mathbf{W}_0|$. Observe that the third condition has its inequality reversed compared to that of equation 7. A similar construction shows that the velocity vectors $\mathbf{V}$ in the subcone bounded by $\mathbf{W}_2$ and $\mathbf{V}_1$ are characterized by

$$\mathbf{V} \cdot \mathbf{\Delta}_2 < 0, \quad \frac{\left(\mathbf{V} \cdot \mathbf{\Delta}_2^{\perp}\right)^2}{|\mathbf{V}|^2} < r^2, \quad \mathbf{V} \cdot \mathbf{W}_2^{\perp} \ge 0 \tag{15}$$

Observe that the third condition has its inequality reversed compared to that of equation 8.

The velocity vectors in the subcone bounded by $\mathbf{W}_0$ and $\mathbf{W}_2$ lead to intersections of the circle with points on the top edge of the rounded rectangle. The velocity vectors $\mathbf{V}$ for which this happens are characterized by

$$\mathbf{V} \cdot \mathbf{W}_0^{\perp} \ge 0, \quad \mathbf{V} \cdot \mathbf{W}_2^{\perp} \le 0 \tag{16}$$

Observe that these conditions have their inequalities reversed compared to that of equation 9. The contact point and contact time are determined from the intersection of the ray and the top edge of the rounded rectangle. If $\mathbf{C} = (c_0, c_1)$ and $\mathbf{V} = (\nu_0, \nu_1)$, then the contact time is determined by the y-components, $c_1 + T\nu_1 = e_1 + r$. The contact time T and the contact point $\mathbf{P}$ are

$$T = \frac{e_1 + r - c_1}{\nu_1}, \quad \mathbf{P} = (c_0 + T\nu_0, e_1) \tag{17}$$

All velocity vectors not in the cone do not lead to an intersection. Those vectors $\mathbf{V}$ satisfy the conditions

$$\nu_1 \ge 0 \text{ or } \left(\nu_0 \ge 0 \text{ and } \frac{\left(\mathbf{V} \cdot \mathbf{\Delta}_0^{\perp}\right)^2}{|\mathbf{V}|^2} > r^2\right) \text{ or } \left(\nu_0 \le 0 \text{ and } \frac{\left(\mathbf{V} \cdot \mathbf{\Delta}_2^{\perp}\right)^2}{|\mathbf{V}|^2} > r^2\right) \tag{18}$$

Observe that in these equations, the components ν_0 and ν_1 are swapped compared to that of equation 11

The pseudocode effectively is a search of the subcones, looking for that subcone that contains a ray corresponding to a velocity vector $\mathbf{V}$. The subcones, listed as pairs of bounding vectors, and the conditions that define them are in Table 2.

Table 2. The subcones of the visibility cone for region $\mathcal{R}_6$. The left column shows the bounding vectors as endvectors of an interval that represents the angle constraints for the subcone. A vector $\mathbf{V}$ is in the a subcone when the right-column conditions are satisfied. Observe that those $\mathbf{V} = (\nu_0, \nu_1)$ leading to intersection must be in the third or fourth quadrants; that is, $\nu_1 < 0$. If $\mathbf{V}$ is in another quadrant, there is no intersection and therefore no subcone searching.

subcone interval	defining conditions
$[(-1,0), \mathbf{V}_2]$	$\nu_0 \le 0$ and $\left(\mathbf{V} \cdot \mathbf{\Delta}_2^{\perp}\right)^2 \ge r^2 \lvert\mathbf{V}\rvert^2$
$[\mathbf{V}_2, \mathbf{W}_2]$	$\left(\mathbf{V} \cdot \mathbf{\Delta}_2^{\perp}\right)^2 \le r^2 \lvert\mathbf{V}\rvert^2$ and $\mathbf{V} \cdot \mathbf{W}_2^{\perp} \ge 0$
$[\mathbf{W}_2, \mathbf{W}_0]$	$\mathbf{V} \cdot \mathbf{W}_2^{\perp} \le 0$ and $\mathbf{V} \cdot \mathbf{W}_0^{\perp} \ge 0$
$[\mathbf{W}_0, \mathbf{V}_0]$	$\mathbf{V} \cdot \mathbf{W}_0^{\perp} \ge 0$ and $\left(\mathbf{V} \cdot \mathbf{\Delta}_0^{\perp}\right)^2 \le r^2 \lvert\mathbf{V}\rvert^2$
$[\mathbf{V}_0, (1,0)]$	$\left(\mathbf{V} \cdot \mathbf{\Delta}_0^{\perp}\right)^2 \ge r^2 \lvert\mathbf{V}\rvert^2$ and $\nu_0 \ge 0$

Listing 9 contains pseudocode for this case.

Listing 9. Intersection analysis when $\mathbf{C} \in \mathcal{R}_6$.

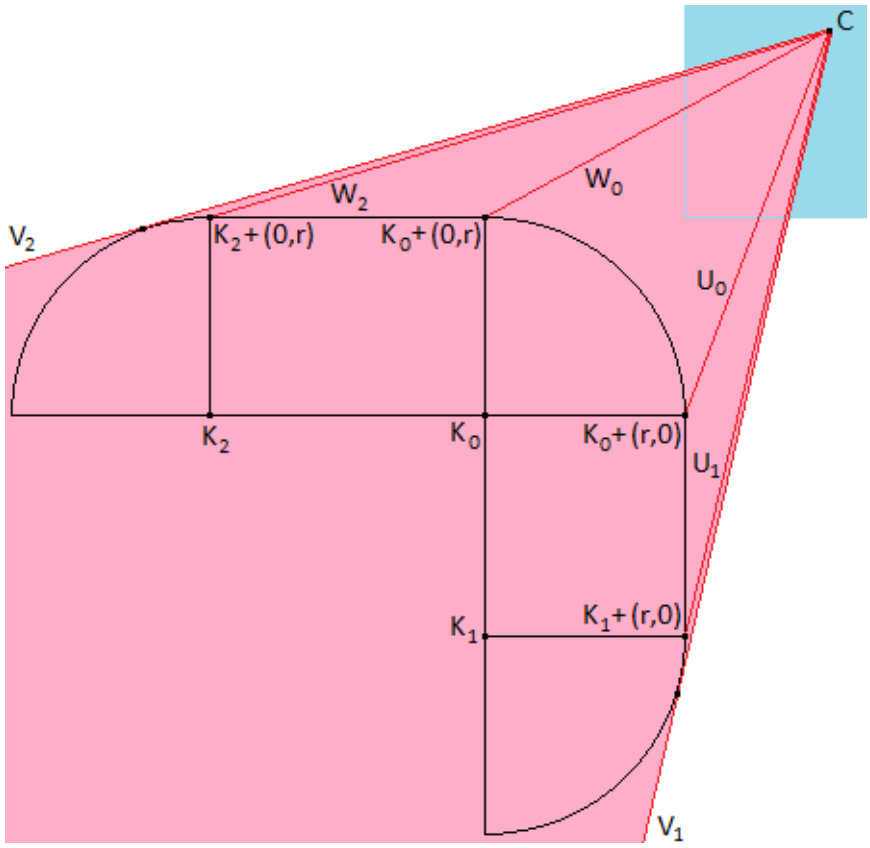
```
int InRegion6(Vector2 K0, Vector2 C, Real radius, Vector2 delta0, Vector2 V,
    Real& contactTime, Real& contactPoint)
{
    if (V[1] < 0)
    {
        Real dotVPerpDelta0 = Dot(V, Perp(delta0));
        Real dotVPerpW0 = radius * V[0] - dotVPerpDelta0;
        if (dotVPerpW0 >= 0)
        {
            Vector2 K2(-K0[0], K0[1]);
            Vector2 delta2 = C - K2;
            Real dotVPerpDelta2 = Dot(V, Perp(delta2));
            Real dotVPerpW2 = radius * V[0] - dotVPerpDelta2;
            if (dotVPerpW2 <= 0)
            {
                return IntersectsTopEdge(K0, C, radius, V, contactTime, contactPoint);
            }
            else
            {
                Real sqrRadius = radius * radius;
                Real dotVV = Dot(V, V);
                Real discr = sqrRadius * dotVV - dotVPerpDelta2 * dotVPerpDelta2;
                if (discr >= 0)
                {
                    return IntersectsArc(K2, -Dot(V, delta2), discr, dotVV, contactTime, contactPoint);
                }
                else  // outside V2-tangent ray
                {
                    return NoIntersection(contactTime, contactPoint);
                }
            }
        }
        else
        {
            Real sqrRadius = radius * radius;
            Real dotVV = Dot(V, V);
            Real discr = sqrRadius * dotVV - dotVPerpDelta0 * dotVPerpDelta0;
            if (discr >= 0)
            {
```

```
                return IntersectsArc(K0, -Dot(V, delta0), discr, dotVV, contactTime, contactPoint);
            }
            else  // outside V0-tangent ray
            {
                return NoIntersection(contactTime, contactPoint);
            }
        }
    }
    else  // circle moves away from rectangle
    {
        return NoIntersection(contactTime, contactPoint);
    }
}

int IntersectsTopEdge(Vector2 K0, Vector2 C, Real radius, Vector2 V, Real& contactTime, Vector2& contactPoint)
{
    contactTime = (K0[1] + radius - C[1]) / V[1];
    contactPoint = Vector2(C[0] + contactTime * V[0], K0[1]);
    return +1;
}
```

3.8 Analysis When C is in $\mathcal{R}_7$

The final case is a combination of the ideas from regions $\mathcal{R}_5$ and $\mathcal{R}_6$. The naming of points and vectors for those cases is used in this case. Figure 5 shows the 2D cone of rays that lead to intersection of the circle and rectangle.

Figure 5. An illustration of the 2D cone of velocity vectors for $\mathbf{C} \in \mathcal{R}_7$. A cone ray can intersect the top-left corner arc, the top edge, the top-right corner arc, the right edge or the bottom-right corner arc of the rounded rectangle.



The pseudocode effectively is a search of the subcones, looking for that subcone that contains a ray corresponding to a velocity vector $\mathbf{V}$. The subcones, listed as pairs of bounding vectors, and the conditions that define them are in Table 3.

Table 3. The subcones of the visibility cone for region $\mathcal{R}_7$. The left column shows the bounding vectors as endvectors of an interval that represents the angle constraints for the subcone. A vector $\mathbf{V}$ is in the a subcone when the right-column conditions are satisfied. Observe that those $\mathbf{V} = (\nu_0, \nu_1)$ leading to intersection must be in the third quadrant; that is, $\nu_0 < 0$ and $\nu_1 < 0$. If $\mathbf{V}$ is in another quadrant, there is no intersection and therefore no subcone searching.

subcone interval	defining conditions
$[(0,-1), \mathbf{V}_1]$	$\nu_1 \le 0$ and $\left(\mathbf{V}\cdot\mathbf{\Delta}_1^{\perp}\right)^2 \ge r^2\lvert\mathbf{V}\rvert^2$
$[\mathbf{V}_1, \mathbf{U}_1]$	$\left(\mathbf{V}\cdot\mathbf{\Delta}_1^{\perp}\right)^2 \le r^2\lvert\mathbf{V}\rvert^2$ and $\mathbf{V}\cdot\mathbf{U}_1^{\perp} \le 0$
$[\mathbf{U}_1, \mathbf{U}_0]$	$\mathbf{V}\cdot\mathbf{U}_1^{\perp} \ge 0$ and $\mathbf{V}\cdot\mathbf{U}_0^{\perp} \le 0$
$[\mathbf{U}_0, \mathbf{W}_0]$	$\mathbf{V}\cdot\mathbf{U}_0^{\perp} \ge 0$ and $\mathbf{V}\cdot\mathbf{W}_0^{\perp} \le 0$
$[\mathbf{W}_0, \mathbf{W}_2]$	$\mathbf{V}\cdot\mathbf{W}_0^{\perp} \ge 0$ and $\mathbf{V}\cdot\mathbf{W}_2^{\perp} \le 0$
$[\mathbf{W}_2, \mathbf{V}_2]$	$\mathbf{V}\cdot\mathbf{W}_2^{\perp} \ge 0$ and $\left(\mathbf{V}\cdot\mathbf{\Delta}_2^{\perp}\right)^2 \le r^2\lvert\mathbf{V}\rvert^2$
$[\mathbf{V}_2, (-1,0)]$	$\left(\mathbf{V}\cdot\mathbf{\Delta}_2^{\perp}\right)^2 \ge r^2\lvert\mathbf{V}\rvert^2$ and $\nu_0 \le 0$

Listing 10 contains pseudocode for this case.

Listing 10. Intersection analysis when $\mathbf{C} \in \mathcal{R}_7$.

```
int InRegion7(Vector2 K0, Vector2 C, Real radius, Vector2 delta0, Vector2 V,
    Real& contactTime, Vector2& contactPoint)
{
    if (V[0] < 0 && V[1] < 0)
    {
        Real dotVPerpDelta0 = Dot(V, Perp(delta0));
        Real dotVPerpW0 = radius * V[0] - dotVPerpDelta0;
        if (dotVPerpW0 <= 0)
        {
            Real dotVPerpU0 = -radius * V[1] - dotVPerpDelta0;
            if (dotVPerpU0 >= 0)
            {
                Real sqrRadius = radius * radius;
                Real dotVV = Dot(V, V);
                Real discr = sqrRadius * dotVV - dotVPerpDelta0 * dotVPerpDelta0;
                return IntersectsArc(K0, -Dot(V, delta0), discr, dotVV, contactTime, contactPoint);
            }
            else
            {
                Vector2<Real> K1{ K0[0], -K0[1] };
                Vector2<Real> delta1 = C - K1;
                Real dotVPerpDelta1 = Dot(V, Perp(delta1));
                Real dotVPerpU1 = -radius * V[1] - dotVPerpDelta1;
                if (dotVPerpU1 >= 0)
                {
                    return IntersectsRightEdge(K0, C, radius, V, contactTime, contactPoint);
                }
                else
                {
                    Real sqrRadius = radius * radius;
                    Real dotVV = Dot(V, V);
                    Real discr = sqrRadius * dotVV - dotVPerpDelta1 * dotVPerpDelta1;
                    if (discr >= 0)
                    {
```

```
                    return IntersectsArc(K1, -Dot(V, delta1), discr, dotVV, contactTime, contactPoint);
                }
                else
                {
                    // outside V1-tangent ray
                    return NoIntersection(contactTime, contactPoint);
                }
            }
        }
    }
    else
    {
        Vector2<Real> K2{ -K0[0], K0[1] };
        Vector2<Real> delta2 = C - K2;
        Real dotVPerpDelta2 = Dot(V, Perp(delta2));
        Real dotVPerpW2 = radius * V[0] - dotVPerpDelta2;
        if (dotVPerpW2 <= 0)
        {
            return IntersectsTopEdge(K0, C, radius, V, contactTime, contactPoint);
        }
        else
        {
            Real sqrRadius = radius * radius;
            Real dotVV = Dot(V, V);
            Real discr = sqrRadius * dotVV - dotVPerpDelta2 * dotVPerpDelta2;
            if (discr >= 0)
            {
                return IntersectsArc(K2, -Dot(V, delta2), discr, dotVV, contactTime, contactPoint);
            }
            else
            {
                // outside V2-tangent ray
                return NoIntersection(contactTime, contactPoint);
            }
        }
    }
}
else
{
    // circle moves away from rectangle
    return NoIntersection(contactTime, contactPoint);
}
}
```

---

4 Code Reduction

4.1 Reduction by Initializing the Output Variables before Analysis

In many of the InRegion* functions, the contact time and contact point are set to invalid numbers when there is no intersection. We can eliminate these cases by introducing a data structure for the output of the intersection query,

```
struct Result
{
    int intersectionType;  // in {-1, 0, 1}
    Real contactTime;
    Vector2 contactPoint;
};
```

and initializing the members to zero (invalid time/point) before calling the InRegion* functions. This eliminates the blocks of code that call NoIntersection.

4.2 Reduction using Symmetry

The previous section provided intersection analysis for each of the 8 regions $\mathcal{R}_0$ through $\mathcal{R}_7$. Pseudocode was given for each of these regions. Figure 1 shows that there is additional symmetry that can be used to reduce the amount of source code by sharing. Specifically, observe that regions $\mathcal{R}_1$ and $\mathcal{R}_2$ have the same geometric structure where we can process $\mathcal{R}_2$ using the code for $\mathcal{R}_1$ but with a rectangle reflected through the line $y = x$. The reflected rectangle has extents (e_1, e_0). Naturally, the circle center $\mathbf{C}$ and the velocity $\mathbf{V}$ must be reflected for the analysis. Similarly, regions $\mathcal{R}_5$ and $\mathcal{R}_6$ have the same geometric structure, and a reflection of the rectangle, circle center and velocity can be used for the analysis.

The regions are renamed as follows to be more suggestive of the association of regions with rectangle features. Table 4 shows the new names. The acronyms used in the subscripts are ovr for *overlapping* and sep for *separated*, both referring to the time-zero state of the objects. They also include int for *interior* and unb for *unbounded*.

Table 4. The renaming of regions to associate the new names with rectangle features. The functions that process the regions is shown. Observe the function sharing for some of the regions.

old name	new name	function name
$\mathcal{R}_0$	$\mathcal{R}_{\text{int-ovr}}$	InteriorOverlap
$\mathcal{R}_1$	$\mathcal{R}_{\text{x-ovr}}$	EdgeOverlap(0,1,. . .)
$\mathcal{R}_2$	$\mathcal{R}_{\text{y-ovr}}$	EdgeOverlap(1,0,. . .)
$\mathcal{R}_3$	$\mathcal{R}_{\text{xy-ovr}}$	VertexOverlap
$\mathcal{R}_4$	$\mathcal{R}_{\text{xy-sep}}$	VertexSeparated
$\mathcal{R}_5$	$\mathcal{R}_{\text{x-unb}}$	EdgeUnbounded(0,1,. . .)
$\mathcal{R}_6$	$\mathcal{R}_{\text{y-unb}}$	EdgeUnbounded(1,0,. . .)
$\mathcal{R}_7$	$\mathcal{R}_{\text{xy-unb}}$	VertexUnbounded

Although I could have used sep instead of unb for regions $\mathcal{R}_5$ and $\mathcal{R}_6$, the extension of the algorithm to moving spheres and boxes has a finer decomposition into regions, some of which correspond to a feature that requires both subscript tags. The idea in 2D is similar to that of having 3 vertex-related regions: $\mathcal{R}_{\text{xy-ovr}}$, $\mathcal{R}_{\text{xy-sep}}$ and $\mathcal{R}_{\text{xy-unb}}$.

4.3 The Shared Functions

We have only 2 shared functions, so the code reduction is not too extensive. However, when extending the idea to moving sphere-box intersections, the sharing becomes more significant.

The implementation of InRegionEdgeOverlap is shown in Listing 11.

Listing 11. The functions InRegion1 and InRegion2 are consolidated into the shared function InRegionEdgeOverlap. It is equivalent to InRegion1 when $\mathbf{C} \in \mathcal{R}_{\text{x-ovr}}$, but when $\mathbf{C} \in \mathcal{R}_{\text{y-ovr}}$, InRegion1 is effectively called with x- and y-components swapped.

```
void InRegionEdgeOverlap(int i0, int i1, Vector2 C, Real radius, Vector2 delta, Result& result)
{
    result.intersectionType = (delta[i0] < radius ? -1 : 1);
    result.contactTime = 0;
    result.contactPoint[i0] = C[i0] - delta[i0];
    result.contactPoint[i1] = C[i1];
}

// Add to the beginning of DoQuery a Result object to store the query results.
Result result = { 0, 0, Vector2(0, 0) };

// Replace the DoQuery line
//   return InRegion1(C, delta, radius, contactTime, contactPoint);
// by the line
InRegionEdgeOverlap(0, 1, C, delta, radius, result);

// Replace the DoQuery line
//   return InRegion2(C, delta, radius, contactTime, contactPoint);
// by the line
InRegionEdgeOverlap(1, 0, C, delta, radius, result);
```

The computation of contact points on an edge is consolidated as shown in Listing 12. The computation for contact with the vertex is also modify to deal with the swapping of x- and y-components.

Listing 12. The function IntersectArc needs to be passed the indices for swapping. The function is renamed to IntersectVertex. The functions IntersectsRightEdge and IntersectsTopEdge can be consolidated into the shared function IntersectsEdge. It is equivalent to calling IntersectsRightEdge when the contact point is on the right edge of the rectangle, but when the contact point is on the top edge, IntersectsRightEdge is effectively called with x- and y-components swapped.

```
void IntersectsVertex(int i0, int i1, Vector2 K, Real q0, Real q1, Real q2, Result& result)
{
    result.intersectionType = +1;
    result.contactTime = (q0 - sqrt(q1)) / q2;
    result.contactPoint[i0] = K[i0];
    result.contactPoint[i1] = K[i1];
}

void IntersectsEdge(int i0, int i1, Vector2 K0, Vector2 C, Real radius, Vector2 V, Result& result)
{
    result.intersectionType = +1;
    result.contactTime = (K0[i0] + radius - C[i0]) / V[i0];
    result.contactPoint[i0] = K0[i0];
    result.contactPoint[i1] = C[i1] + result.contactTime * V[i1];
}

// Replace the line
//   return IntersectsRightEdge(K0, C, radius, V, contactTime, contactPoint);
// with the line
IntersectsEdge(0, 1, K0, C, radius, V, result);

// Replace the line
//   return IntersectsTopEdge(K0, C, radius, V, contactTime, contactPoint);
// with the line
IntersectsEdge(1, 0, K0, C, radius, V, result);
```

The implementation of InRegionEdgeUnbounded is shown in Listing 13.

Listing 13. The functions InRegion5 and InRegion6 are consolidated into the shared function InRegionEdgeUnbounded. It is equivalent to InRegion5 when $\mathbf{C} \in \mathcal{R}_{\text{x-unb}}$, but when $\mathbf{C} \in \mathcal{R}_{\text{y-unb}}$, InRegion5 is effectively called with x- and y-components swapped. Observe that the vector operations need to be expanded into scalar operations in order to consume the indices i0 and i1 properly.

```
int InRegionEdgeUnbounded(int i0, int i1, Vector2 K0, Vector2 C, Vector2 delta0, Real radius,
    Vector2 V, Result& result)
{
    if (V[i0] < 0)
    {
        Real dotVPerpDelta0 = V[i0] * delta0[i1] - V[i1] * delta0[i0];
        Real dotVPerpU0 = radius * V[i1] + dotVPerpDelta0;
        if (dotVPerpU0 >= 0)
        {
            Vector2 K1, delta1;
            K1[i0] = K0[i0];
            K1[i1] = -K0[i1];
            delta1[i0] = C[i0] - K1[i0];
            delta1[i1] = C[i1] - K1[i1];
            Real dotVPerpDelta1 = V[i0] * delta1[i1] - V[i1] * delta1[i0];
            Real dotVPerpU1 = radius * V[i1] + dotVPerpDelta1;
            if (dotVPerpU1 <= 0)
            {
                IntersectsEdge(i0, i1, K0, C, radius, V, result);
            }
            else
            {
                Real q2 = Dot(V, V);
                Real q1 = radius * radius * q2 - dotVPerpDelta1 * dotVPerpDelta1;
                if (q1 >= 0)
                {
                    Real q0 = -(V[i0] * delta1[i0] + V[i1] * delta1[i1]);
                    IntersectsVertex(i0, i1, K1, q0, q1, q2, result);
                }
            }
        }
        else
        {
            Real q2 = Dot(V, V);
            Real q1 = radius * radius * q2 - dotVPerpDelta0 * dotVPerpDelta0;
            if (q1 >= 0)
            {
                Real q0 = -(V[i0] * delta0[i0] + V[i1] * delta0[i1]);
                IntersectsVertex(i0, i1, K0, q0, q1, q2, result);
            }
        }
    }
}

// Replace the DoQuery line
//   return InRegion5(K, C, delta, radius, V, contactTime, contactPoint);
// by the line
InRegionEdgeUnbounded(0, 1, K, C, delta, radius, V, result);

// Replace the DoQuery line
//   return InRegion6(K, C, delta, radius, V, contactTime, contactPoint);
// by the line
InRegionEdgeUnbounded(1, 0, K, C, delta, radius, V, result);
```

---

5 Implementation

The GTE distribution has an implementation of the algorithm in files IntrAlignedBox2Circle2.h and IntrOrientedBox2Circle2.h, both sharing a member function DoQuery in the aligned-box case.

A sample application is GeometricTools/GTE/Samples/Intersection/MovingCircleRectangle. For a specified stationary rectangle, the query is performed for a moving circle. The rectangle is drawn as a blue wireframe. The rounded rectangle is drawn as a light-gray region. The circle is drawn as a red wireframe. The velocity vector has unit length in this application. The ray whose origin is the circle center and with the velocity as direction is drawn in green. Two rays tangent to the circle and parallel to the velocity are drawn to that you can see where they intersect the rounded rectangle. You can left-click-and-drag the mouse to modify the velocity vector. You can right-click-and-drag the mouse to move the circle center. If you press the + key, the rectangle is rotated clockwise. If you press the − key, the rectangle is rotated counterclockwise. Figure 6 shows a screen capture where the circle and rectangle are not initially overlapping but intersect at a later time.

Figure 6. The circle and rectangle are not initially overlapping but they intersect at later time.

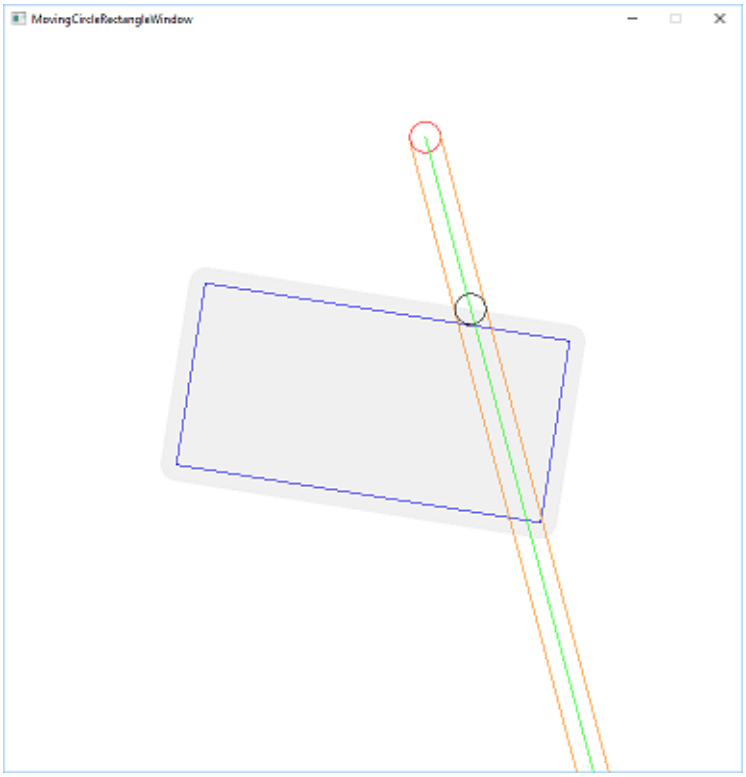
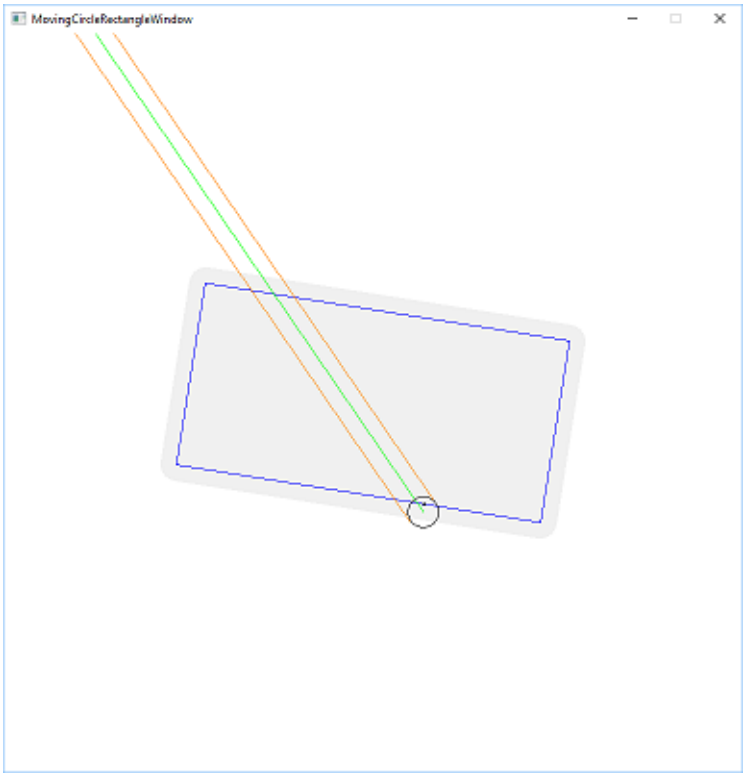


Figure 7 shows a screen capture where the circle and rectangle are initially overlapping.

Figure 7. The circle and rectangle are initially overlapping but some point is chosen as the contact point even though the intersection point set has positive area.



The source code for moving circle and axis-aligned rectangle is shown in Listing 14. This code is shared by the query involving an oriented rectangle.

Listing 14. C++ code for the moving circle and axis-aligned rectangle intersection query.

```
template <typename Real>
class FIQuery<Real, AlignedBox2<Real>, Circle2<Real>>
{
public:
    // Currently, only a dynamic query is supported.  A static query will
    // need to compute the intersection set of (solid) box and circle.
    struct Result
    {
        // The cases are
        // 1. Objects initially overlapping.  The contactPoint is invalid.
        //      intersectionType = -1
        //      contactTime = 0
        //      contactPoint = (0,0) (TODO: Set to point in intersection?)
        // 2. Objects initially separated but do not intersect later.  The
        //      contactTime and contactPoint are invalid.
        //      intersectionType = 0
        //      contactTime = 0
        //      contactPoint = (0,0)
        // 3. Objects initially separated but intersect later.
        //      intersectionType = +1
        //      contactTime = first time T > 0
        //      contactPoint = corresponding first contact
        int intersectionType;
        Real contactTime;
        Vector2<Real> contactPoint;
    };

    Result operator()(AlignedBox2<Real> const& box, Vector2<Real> const& boxVelocity,
        Circle2<Real> const& circle, Vector2<Real> const& circleVelocity)
    {
        // Translate the circle and box so that the box center becomes
        // the origin.  Compute the velocity of the circle relative to
        // the box.
        Real const zero(0), one(1), minusOne(-1), half(0.5);
        Vector2<Real> boxCenter = (box.max + box.min) * half;
        Vector2<Real> extent = (box.max - box.min) * half;
        Vector2<Real> C = circle.center - boxCenter;
        Vector2<Real> V = circleVelocity - boxVelocity;

        // Change signs on components, if necessary, to transform C to the
        // first quadrant.  Adjust the velocity accordingly.
        Real sign[2];
        for (int i = 0; i < 2; ++i)
        {
            if (C[i] >= zero)
            {
                sign[i] = one;
            }
            else
            {
                C[i] = -C[i];
                V[i] = -V[i];
                sign[i] = minusOne;
            }
        }

        Result result = { 0, zero, { zero, zero } };
        DoQuery(extent, C, circle.radius, V, result);

        if (result.intersectionType != 0)
        {
            // Translate back to the original coordinate system.
            for (int i = 0; i < 2; ++i)
            {
```

```
                if (sign[i] < zero)
                {
                    result.contactPoint[i] = -result.contactPoint[i];
                }
            }

            result.contactPoint += boxCenter;
        }
        return result;
    }

protected:
    void DoQuery(Vector2<Real> const& K, Vector2<Real> const& C,
        Real radius, Vector2<Real> const& V, Result& result)
    {
        Real const zero(0);
        Vector2<Real> delta = C - K;
        if (delta[1] <= radius)
        {
            if (delta[0] <= radius)
            {
                if (delta[1] <= zero)
                {
                    if (delta[0] <= zero)
                    {
                        InRegionInteriorOverlap(C, result);
                    }
                    else
                    {
                        InRegionEdgeOverlap(0, 1, C, delta, radius, result);
                    }
                }
                else
                {
                    if (delta[0] <= zero)
                    {
                        InRegionEdgeOverlap(1, 0, C, delta, radius, result);
                    }
                    else
                    {
                        Real sqrRadius = radius * radius;
                        Real sqrDistance = delta[0] * delta[0] + delta[1] * delta[1];
                        if (sqrDistance <= sqrRadius)
                        {
                            InRegionVertexOverlap(K, delta, radius, result);
                        }
                        else
                        {
                            InRegionVertexSeparated(K, delta, V, radius, result);
                        }
                    }

                }
            }
            else
            {
                InRegionEdgeUnbounded(0, 1, K, C, radius, delta, V, result);
            }
        }
        else
        {
            if (delta[0] <= radius)
            {
                InRegionEdgeUnbounded(1, 0, K, C, radius, delta, V, result);
            }
            else
            {
                InRegionVertexUnbounded(K, C, radius, delta, V, result);
            }
        }
    }
```

```
private:
    void InRegionInteriorOverlap(Vector2<Real> const& C, Result& result)
    {
        Real const zero(0);
        result.intersectionType = -1;
        result.contactTime = zero;
        result.contactPoint = C;
    }

    void InRegionEdgeOverlap(int i0, int i1, Vector2<Real> const& C,
        Vector2<Real> const& delta, Real radius, Result& result)
    {
        Real const zero(0);
        result.intersectionType = (delta[i0] < radius ? -1 : 1);
        result.contactTime = zero;
        result.contactPoint[i0] = C[i0] - delta[i0];
        result.contactPoint[i1] = C[i1];
    }

    void InRegionVertexOverlap(Vector2<Real> const& K0, Vector2<Real> const& delta,
        Real radius, Result& result)
    {
        Real const zero(0);
        Real sqrDistance = delta[0] * delta[0] + delta[1] * delta[1];
        Real sqrRadius = radius * radius;
        result.intersectionType = (sqrDistance < sqrRadius ? -1 : 1);
        result.contactTime = zero;
        result.contactPoint = K0;
    }

    void InRegionVertexSeparated(Vector2<Real> const& K0, Vector2<Real> const& delta0,
        Vector2<Real> const& V, Real radius, Result& result)
    {
        Real const zero(0);
        Real q0 = -Dot(V, delta0);
        if (q0 > zero)
        {
            Real dotVPerpD0 = Dot(V, Perp(delta0));
            Real q2 = Dot(V, V);
            Real q1 = radius * radius * q2 - dotVPerpD0 * dotVPerpD0;
            if (q1 >= zero)
            {
                IntersectsVertex(0, 1, K0, q0, q1, q2, result);
            }
        }
    }

    void InRegionEdgeUnbounded(int i0, int i1, Vector2<Real> const& K0, Vector2<Real> const& C,
        Real radius, Vector2<Real> const& delta0, Vector2<Real> const& V, Result& result)
    {
        const Real zero(0);
        if (V[i0] < zero)
        {
            Real dotVPerpD0 = V[i0] * delta0[i1] - V[i1] * delta0[i0];
            if (radius * V[1] + dotVPerpD0 >= zero)
            {
                Vector2<Real> K1, delta1;
                K1[i0] = K0[i0];
                K1[i1] = -K0[i1];
                delta1[i0] = C[i0] - K1[i0];
                delta1[i1] = C[i1] - K1[i1];
                Real dotVPerpD1 = V[i0] * delta1[i1] - V[i1] * delta1[i0];
                if (radius * V[1] + dotVPerpD1 <= zero)
                {
                    IntersectsEdge(i0, i1, K0, C, radius, V, result);
                }
                else
                {
                    Real q2 = Dot(V, V);
                    Real q1 = radius * radius * q2 - dotVPerpD1 * dotVPerpD1;
                    if (q1 >= zero)
                    {
```

```
                    Real q0 = -(V[i0] * delta1[i0] + V[i1] * delta1[i1]);
                    IntersectsVertex(i0, i1, K1, q0, q1, q2, result);
                }
            }
        }
        else
        {
            Real q2 = Dot(V, V);
            Real q1 = radius * radius * q2 - dotVPerpD0 * dotVPerpD0;
            if (q1 >= zero)
            {
                Real q0 = -(V[i0] * delta0[i0] + V[i1] * delta0[i1]);
                IntersectsVertex(i0, i1, K0, q0, q1, q2, result);
            }
        }
    }
}

void InRegionVertexUnbounded(Vector2<Real> const& K0, Vector2<Real> const& C, Real radius,
    Vector2<Real> const& delta0, Vector2<Real> const& V, Result& result)
{
    Real const zero(0);
    if (V[0] < zero && V[1] < zero)
    {
        Real dotVPerpD0 = Dot(V, Perp(delta0));
        if (radius * V[0] - dotVPerpD0 <= zero)
        {
            if (-radius * V[1] - dotVPerpD0 >= zero)
            {
                Real q2 = Dot(V, V);
                Real q1 = radius * radius * q2 - dotVPerpD0 * dotVPerpD0;
                Real q0 = -Dot(V, delta0);
                IntersectsVertex(0, 1, K0, q0, q1, q2, result);
            }
            else
            {
                Vector2<Real> K1{ K0[0], -K0[1] };
                Vector2<Real> delta1 = C - K1;
                Real dotVPerpD1 = Dot(V, Perp(delta1));
                if (-radius * V[1] - dotVPerpD1 >= zero)
                {
                    IntersectsEdge(0, 1, K0, C, radius, V, result);
                }
                else
                {
                    Real q2 = Dot(V, V);
                    Real q1 = radius * radius * q2 - dotVPerpD1 * dotVPerpD1;
                    if (q1 >= zero)
                    {
                        Real q0 = -Dot(V, delta1);
                        IntersectsVertex(0, 1, K1, q0, q1, q2, result);
                    }
                }
            }
        }
        else
        {
            Vector2<Real> K2{ -K0[0], K0[1] };
            Vector2<Real> delta2 = C - K2;
            Real dotVPerpD2 = Dot(V, Perp(delta2));
            if (radius * V[0] - dotVPerpD2 <= zero)
            {
                IntersectsEdge(1, 0, K0, C, radius, V, result);
            }
            else
            {
                Real q2 = Dot(V, V);
                Real q1 = radius * radius * q2 - dotVPerpD2 * dotVPerpD2;
                if (q1 >= zero)
                {
                    Real q0 = -Dot(V, delta2);
                    IntersectsVertex(1, 0, K2, q0, q1, q2, result);
```

```
                    }
                }
            }
        }
    }

    void IntersectsVertex(int i0, int i1, Vector2<Real> const& K,
        Real q0, Real q1, Real q2, Result& result)
    {
        result.intersectionType = +1;
        result.contactTime = (q0 - Function<Real>::Sqrt(q1)) / q2;
        result.contactPoint[i0] = K[i0];
        result.contactPoint[i1] = K[i1];
    }

    void IntersectsEdge(int i0, int i1, Vector2<Real> const& K0, Vector2<Real> const& C,
        Real radius, Vector2<Real> const& V, Result& result)
    {
        result.intersectionType = +1;
        result.contactTime = (K0[i0] + radius - C[i0]) / V[i0];
        result.contactPoint[i0] = K0[i0];
        result.contactPoint[i1] = C[i1] + result.contactTime * V[i1];
    }
};
```

The source code for moving circle and oriented rectangle is shown in Listing 15.

Listing 15. C++ code for the moving circle and oriented rectangle intersection query.

```
template <typename Real>
class FIQuery<Real, OrientedBox2<Real>, Circle2<Real>>
    :
    public FIQuery<Real, AlignedBox2<Real>, Circle2<Real>>
{
public:
    // See the base class for the definition of 'struct Result'.
    Result operator()(OrientedBox2<Real> const& box, Vector2<Real> const& boxVelocity,
        Circle2<Real> const& circle, Vector2<Real> const& circleVelocity)
    {
        // Transform the oriented box to an axis-aligned box centered at
        // the origin and transform the circle accordingly.  Compute the
        // velocity of the circle relative to the box.
        Real const zero(0), one(1), minusOne(-1);
        Vector2<Real> cdiff = circle.center - box.center;
        Vector2<Real> vdiff = circleVelocity - boxVelocity;
        Vector2<Real> C, V;
        for (int i = 0; i < 2; ++i)
        {
            C[i] = Dot(cdiff, box.axis[i]);
            V[i] = Dot(vdiff, box.axis[i]);
        }

        // Change signs on components, if necessary, to transform C to the
        // first quadrant.  Adjust the velocity accordingly.
        Real sign[2];
        for (int i = 0; i < 2; ++i)
        {
            if (C[i] >= zero)
            {
                sign[i] = one;
            }
            else
            {
                C[i] = -C[i];
                V[i] = -V[i];
                sign[i] = minusOne;
            }
```

```
        }

        Result result = { 0, zero, { zero, zero } };
        DoQuery(box.extent, C, circle.radius, V, result);

        if (result.intersectionType != 0)
        {
            // Transform back to the original coordinate system.
            result.contactPoint = box.center
                + (sign[0] * result.contactPoint[0]) * box.axis[0]
                + (sign[1] * result.contactPoint[1]) * box.axis[1];
        }
        return result;
    }
};
```
