```
    for (i0 = 0, i1 = C0.N - 1; i0 < C0.N; i1 = i0, i0++) {
        D = Perp(C0.E(i1));  // C0.E(i1) = C0.V(i0) - C0.V(i1));
        if (WhichSide(C1.V, C0.V(i0), D) > 0) {
            // C1 is entirely on 'positive' side of line C0.V(i0) + t * D
            return false;
        }
    }

    // Test edges of C1 for separation.  Because of the counterclockwise ordering,
    // the projection interval for C1 is [m,0] where m <= 0.  Only try to determine
    // if C0 is on the 'positive' side of the line.
    for (i0 = 0, i1 = C1.N - 1; i0 < C1.N; i1 = i0, i0++) {
        D = Perp(C1.E(i1));  // C1.E(i1) = C1.V(i0) - C1.V(i1));
        if (WhichSide(C0.V, C1.V(i0), D) > 0) {
            // C0 is entirely on 'positive' side of line C1.V(i0) + t * D
            return false;
        }
    }

    return true;
}

int WhichSide(PointSet S, Point P, Point D)
{
    // S vertices are projected to the form P + t * D.  Return value is +1 if all t > 0,
    // -1 if all t < 0, 0 otherwise (in which case the line splits the polygon).

    positive = 0; negative = 0; zero = 0;
    for (i = 0; i < C.N; i++) {
        t = Dot(D, S.V(i) - P);
        if (t > 0) positive++; else if (t < 0) negative++; else zero++;
        if (positive && negative || zero) return 0;
    }
    return positive ? 1 : -1;
}
```

An Asymptotically Better Alternative

Although the alternative implementation is roughly twice as fast as the direct implementation, both are of order $O(NM)$, where N and M are the number of vertices for the convex polygons. An asymptotically better alternative uses a form of bisection to find an extreme point of the projection of the polygon (O'Rourke 1998). The

bisection effectively narrows in on sign changes of the dot product of edges with the specified direction vector. For a polygon of N vertices, the bisection is of order $O(\log N)$, so the total algorithm is $O(\max\{N \log M, M \log N\})$.

Given two vertex indices i_0 and i_1 of a polygon with N vertices, the *middle index* of the indices is described by the following pseudocode:

```
int GetMiddleIndex(int i0, int i1, int N)
{
    if (i0 < i1)
        return (i0 + i1) / 2;
    else
        return (i0 + i1 + N) / 2 % N);
    }
```

The division of two integers returns the largest integer smaller than the real-value ratio, and the percent sign indicates modulo arithmetic. Observe that if $i_0 = i_1 = 0$, the function returns a valid index. The condition when $i_0 < i_1$ has an obvious result: the returned index is the average of the input indices, certainly supporting the name of the function. For example, if the polygon has $N = 5$ vertices, inputs $i_0 = 0$ and $i_1 = 2$ lead to a returned index of 1. The other condition handles wraparound of the indices. If $i_0 = 2$ and $i_1 = 0$, the implied set of ordered indices is $\{2, 3, 4, 0\}$. The middle index is selected as 3 since $3 = (2 + 0 + 5)/2 \pmod 5$.

The bisection algorithm to find the extreme value of the projection is

```
int GetExtremeIndex(ConvexPolygon C, Point D)
{
    i0 = 0; i1 = 0;
    while (true) {
        mid = GetMiddleIndex(i0,i1);
        next = (mid + 1) % C.N;
            E = C.V(next) - C.V(mid);
        if (Dot(D, E) > 0)  {
            if (mid != i0) i0 = mid; else return i1;
        } else  {
            prev = (mid + C.N - 1) % C.N;
                E = C.V(mid) - C.V(prev);
            if (Dot(D, E) < 0) i1 = mid; else return mid;
        }
    }
}
```

Using the bisection method, the intersection testing pseudocode is

```
bool TestIntersection(ConvexPolygon C0, ConvexPolygon C1)
{
    // Test edges of C0 for separation.  Because of the counterclockwise ordering,
    // the projection interval for C0 is [m, 0] where m <= 0.  Only try to determine
    // if C1 is on the 'positive' side of the line.
    for (i0 = 0, i1 = C0.N - 1; i0 < C0.N; i1 = i0, i0++)  {
        D = Perp(C0.E(i1)); // C0.E(i1) = C0.V(i0) - C0.V(i1));
        min = GetExtremeIndex(C1, -D);
        diff = C1.V(min) - C0.V(i0);
        if (Dot(D, diff) > 0) {
            // C1 is entirely on 'positive' side of line C0.V(i0) + t * D
            return false;
        }
    }

    // Test edges of C1 for separation.  Because of the counterclockwise ordering,
    // the projection interval for C1 is [m, 0] where m <= 0.  Only try to determine
    // if C0 is on the 'positive' side of the line.
    for (i0 = 0, i1 = C1.N - 1; i0 < C1.N; i1 = i0, i0++) {
        D = Perp(C1.E(i1));  // C1.E(i1) = C1.V(i0) - C1.V(i1));
        min = GetExtremeIndex(C0, -D);
        diff = C0.V(min) - C1.V(i0);
        if (Dot(D,diff) > 0) {
            // C0 is entirely on 'positive' side of line C1.V(i0) + t * D
            return false;
        }
    }

    return true;
}
```

7.7.3 Separation of Moving Convex Polygons

The method of separating axes extends to convex polygons moving with constant velocity. The algorithm is attributed to Ron Levine in a post to the GD algorithms mailing list (Levine 2000). If C_0 and C_1 are convex polygons with velocities $\vec{w}_0$ and $\vec{w}_1$, it can be determined via projections if the polygons will intersect for some time $T \geq 0$. If they do intersect, the first time of contact can be computed. It is enough to work with a stationary polygon C_0 and a moving polygon C_1 with velocity $\vec{w}$ since we can always use $\vec{w} = \vec{w}_1 - \vec{w}_0$ to perform the calculations as if C_0 were not moving.

If C_0 and C_1 are initially intersecting, then the first time of contact is $T = 0$. Otherwise the convex polygons are initially disjoint. The projection of C_1 onto a line with direction $\vec{d}$ not perpendicular to $\vec{w}$ is itself moving. The speed of the projection along the line is $\omega = (\vec{w} \cdot \vec{d})/\|\vec{d}\|^2$. If the projection interval of C_1 moves away from the projection interval of C_0, then the two polygons will never intersect. The cases when intersection might happen are those when the projection intervals for C_1 move toward those of C_0.

The intuition for how to predict an intersection is much like that for selecting the potential separating directions in the first place. If the two convex polygons intersect at a first time $T_{\text{first}} > 0$, then their projections are not separated along any line at that time. An instant before first contact, the polygons are separated. Consequently there must be at least one separating direction for the polygons at time $T_{\text{first}} - \varepsilon$ for small $\varepsilon > 0$. Similarly, if the two convex polygons intersect at a last time $T_{\text{last}} > 0$, then their projections are also not separated at that time along any line, but an instant after last contact, the polygons are separated. Consequently there must be at least one separating direction for the polygons at time $T_{\text{last}} + \varepsilon$ for small $\varepsilon > 0$. Both T_{first} and T_{last} can be tracked as each potential separating axis is processed. After all directions are processed, if $T_{\text{first}} \le T_{\text{last}}$, then the two polygons do intersect with first contact time T_{first}. It is also possible that $T_{\text{first}} > T_{\text{last}}$, in which case the two polygons cannot intersect.

Pseudocode for testing for intersection of two moving convex polygons is given below. The time interval over which the event is of interest is $[0, T_{\text{max}}]$. If knowing an intersection at *any* future time is desired, then set $T_{\text{max}} = \infty$. Otherwise, T_{max} is finite. The function is implemented to indicate there is no intersection on $[0, T_{\text{max}}]$, even though there might be an intersection at some time $T > T_{\text{max}}$.

```
bool TestIntersection(ConvexPolygon C0, Point W0, ConvexPolygon C1, Point W1,
    float tmax, float& tfirst, float& tlast)
{
    W = W1 - W0;  // process as if C0 is stationary, C1 is moving
    tfirst = 0;  tlast = INFINITY;

    // test edges of C0 for separation
    for (i0 = 0, i1 = C0.N - 1; i0 < C0.N; i1 = i0, i0++) {
        D = Perp(C0.E(i1)); // C0.E(i1) = C0.V(i0) - C0.V(i1));
        ComputeInterval(C0, D, min0, max0);
        ComputeInterval(C1, D, min1, max1);
        speed = Dot(D, W);
        if (NoIntersect(tmax, speed, min0, max0, min1, max1, tfirst, tlast))
            return false;
    }

    // test edges of C1 for separation
    for (i0 = 0, i1 = C1.N - 1; i0 < C1.N; i1 = i0, i0++) {
```

```
        D = Perp(C1.E(i1)); // C1.E(i1) = C1.V(i0) - C1.V(i1));
        ComputeInterval(C0, D, min0, max0);
        ComputeInterval(C1, D, min1, max1);
        speed = Dot(D, W);
        if (NoIntersect(tmax, speed, min0, max0, min1, max1, tfirst, tlast))
            return false;
    }
    return true;
}

bool NoIntersect(float tmax, float speed, float min0, float max0,
    float min1, float max1, float& tfirst, float& tlast)
{
    if (max1 < min0) {
        // interval(C1) initially on 'left' of interval(C0)
        if (speed <= 0) return true;  // intervals moving apart
        t = (min0 - max1) / speed;  if (t > tfirst) tfirst = t;
        if (tfirst > tmax) return true;
        t = (max0 - min1) /speed;  if (t < tlast) tlast = t;
        if (tfirst > tlast) return true;
    } else if (max0 < min1) {
        // interval(C1) initially on 'right' of interval(C0)
        if (speed >= 0) return true;  // intervals moving apart
        t = (max0 - min1)/speed;  if ( t > tfirst ) tfirst = t;
        if (tfirst > tmax) return true;
        t = (min0 - max1)/speed;  if ( t < tlast ) tlast = t;
        if (tfirst > tlast) return true;
    }  else {
        // interval(C0) and interval(C1) overlap
        if (speed > 0) {
            t = (max0 - min1) / speed;  if (t < tlast) tlast = t;
            if (tfirst > tlast) return true;
        } else if (speed < 0) {
            t = (min0 - max1) / speed;  if (t < tlast) tlast = t;
            if (tfirst > tlast) return true;
        }
    }
    return false;
}
```

The following example illustrates the ideas. The first box is the unit cube $0 \le x \le 1$ and $0 \le y \le 1$ and is stationary. The second box is initially $0 \le x \le 1$ and $1 + \delta \le y \le 2 + \delta$ for some $\delta > 0$. Let its velocity be $(1, -1)$. Whether or not the second box intersects the first box depends on the value of δ. The only potential separating

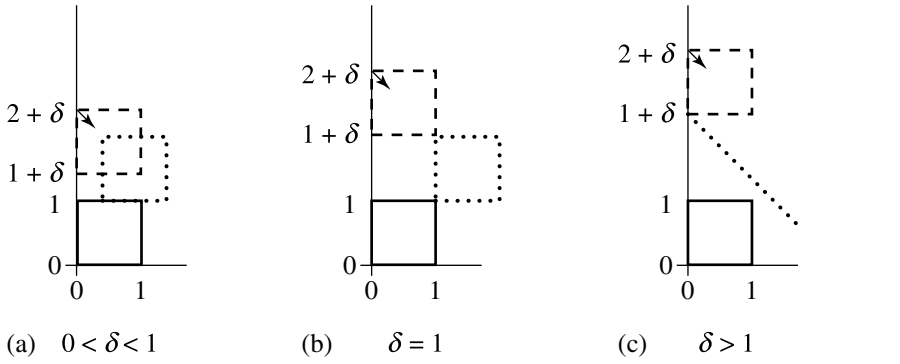

Figure 7.17 (a) Edge-edge intersection predicted. (b) Vertex-vertex intersection predicted. (c) No intersection predicted.

axes are (1, 0) and (0, 1). Figure 7.17 shows the initial configuration for three values of δ, one where there will be an edge-edge intersection, one where there will be a vertex-vertex intersection, and one where there is no intersection. The black box is stationary. The dashed box is moving. The black vector indicates the direction of motion. The dotted boxes indicate where the moving box first touches the stationary box. In Figure 7.17(c) the dotted line indicates that the moving box will miss the stationary box. For $\vec{d} = (1, 0)$, the pseudocode produces `min0 = 0`, `max0 = 1`, `min1 = 0`, `max1 = 1`, and `speed = 1`. The projected intervals are initially overlapping. Since the speed is positive, `T = (max0 - min1)/speed = 1 < TLast = INFINITY` and `TLast` is updated to 1. For $\vec{d} = (0, 1)$, the pseudocode produces `min0 = 0`, `max0 = 1`, `min1 = 1 + delta`, `max1 = 2 + delta`, and `speed = -1`. The moving projected interval is initially on the right of the stationary projected interval. Since the speed is negative, `T = (max0 - min1)/speed = delta > TFirst = 0` and `TFirst` is updated to `delta`. The next block of code sets `T = (min0 - max1)/speed = 2 + delta`. The value `TLast` is not updated since $2 + \delta < 1$ cannot happen for $\delta > 0$. On exit from the loop over potential separating directions, $T_{\text{First}} = \delta$ and $T_{\text{last}} = 1$. The objects intersect if and only if $T_{\text{first}} \le T_{\text{last}}$, or $\delta \le 1$. This condition is consistent with Figure 7.17. Figure 7.17(a) has $\delta < 1$, and Figure 7.17(b) has $\delta = 1$; intersections occur in both cases. Figure 7.17(c) has $\delta > 1$, and no intersection occurs.

7.7.4 Intersection Set for Stationary Convex Polygons

The find-intersection query for two stationary convex polygons is a special example of Boolean operations on polygons. Section 13.5 provides a general discussion for computing Boolean operations. In particular there is a discussion on linear time

computation for the intersection of convex polgons. That is, if the two polygons have N and M vertices, the order of the intersection algorithm is $O(N + M)$. A less efficient algorithm, but one perhaps easier to understand, clips the edges of each polygon against the other polygon. The order of this algorithm is $O(NM)$. Of course the asymptotic analysis applies to large N and M, so the latter algorithm is potentially a good choice for triangles and rectangles.

7.7.5 Contact Set for Moving Convex Polygons

Given two moving convex objects C_0 and C_1, initially not intersecting and with velocities $\vec{w}_0$ and $\vec{w}_1$, we showed earlier how to compute the first time of contact T, if it exists. Assuming it does, the sets $C_0 + T\vec{w}_0 = \{X + T\vec{w}_0 : X \in C_0\}$ and $C_1 + T\vec{w}_1 = \{X + T\vec{w}_1 : X \in C_1\}$ are just touching with no interpenetration. Figure 7.14 shows the various configurations.

The `TestIntersection` function can be modified to keep track of which vertices or edges are projected to the end points of the projection interval. At the first time of contact, this information is used to determine how the two objects are oriented with respect to each other. If the contact is vertex-edge or vertex-vertex, then the contact set is a single point, a vertex. If the contact is edge-edge, the contact set is a line segment that contains at least one vertex. Each end point of the projection interval is either generated by a vertex or an edge. A two-character label is associated with each polygon to indicate the projection type. The single-character labels are `V` for a vertex projection and `E` for an edge projection. The four two-character labels are `VV`, `VE`, `EV`, and `EE`. The first letter corresponds to the minimum of the interval, and the second letter corresponds to the maximum of the interval. It is also necessary to store the projection interval and the vertex or edge indices of the components that project to the interval extremes. A convenient data structure is

```
Configuration
{
    float min, max;
    int index[2];
    char type[2];
};
```

where the projection interval is [min, max]. For example, if the projection type is `EV`, `index[0]` is the index of the edge that projects to the minimum, and `index[1]` is the index of the vertex that projects to the maximum.

Two configuration objects are declared, `Cfg0` for polygon C_0 and `Cfg1` for polygon C_1. In the first loop in `TestIntersection`, the projection of C_0 onto the line containing vertex V_{i_0} and having direction perpendicular to $\vec{e}_{i_1} = V_{i_0} - V_{i_1}$ produces a projection type whose second index is `E` since the outer pointing edge normal is used. The first index can be either `V` or `E` depending on the polygon. The pseudocode is

```
void ProjectNormal(ConvexPolygon C, Point D, int edgeindex, Configuration Cfg)
{
    Cfg.max = Dot(D, C.V(edgeindex));  // = Dot(D, C.V((edgeindex + 1) % C.N))
        Cfg.index[1] = edgeindex;
    Cfg.type[0] = 'V';
    Cfg.type[1] = 'E';

    Cfg.min = Cfg.max;
    for (i = 1, j = (edgeindex + 2) % C.N; i < C.N; i++, j = (j + 1) % C.N) {
            value = Dot(D, C.V(j));
        if (value < Cfg.min) {
            Cfg.min = value;
            Cfg.index[0] = j;
        }  else if (value == Cfg.min) {
            // Found an edge parallel to initial projected edge.  The
            // remaining vertices can only project to values larger than
            // the minimum.  Keep the index of the first visited end point.
            Cfg.type[0] = 'E';
            return;
        }  else { // value > Cfg.min
            // You have already found the minimum of projection, so when
            // the dot product becomes larger than the minimum, you are
            // walking back towards the initial edge.  No point in
            // wasting time to do this, just return since you now know
            // the projection.
            return;
        }
    }
}
```

The projection of C_1 onto an edge normal line of C_0 can lead to any projection type. The pseudocode is

```
void ProjectGeneral(ConvexPolygon C, Point D, Configuration Cfg)
{
    Cfg.min = Cfg.max = Dot(D, C.V(0));
    Cfg.index[0] = Cfg.index[1] = 0;

    for (i = 1; i < C.N; i++) {
        value = Dot(D, C.V(i));
        if (value < Cfg.min)  {
            Cfg.min = value;
            Cfg.index[0] = i;
        }  else if (value > Cfg.max) {
```

```
            Cfg.max = value;
            Cfg.index[1] = i;
        }
    }

    Cfg.type[0] = Cfg.type[1] = 'V';
    for (i = 0; i < 2; i++) {
        if (Dot(D, C.E(Cfg.index[i] - 1)) == 0) {
            Cfg.index[i] = Cfg.index[i] - 1;
            Cfg.type[i] = 'E';
        }  else if (Dot(D, C.E(Cfg.index[i] + 1)) == 0) {
            Cfg.type[i] = 'E';
        }
    }
}
```

The index arithmetic for the edges of `C` is performed modulo `C.N` so that the resulting index is within range.

The `NoIntersect` function accepted as input the projection intervals for the two polygons. Now those intervals are stored in the configuration objects, so `NoIntersect` must be modified to reflect this. In the event that there will be an intersection between the moving polygons, it is necessary that the configuration information be saved for later use in determining the contact set. As a result, `NoIntersect` must keep track of the configuration objects corresponding to the current first time of contact. Finally, the contact set calculation will require knowledge of the order of the projection intervals. `NoIntersect` will set a flag with value $+1$ if the intervals intersect at the maximum of the C_0 interval and the minimum of the C_1 interval or with value -1 if the intervals intersect at the minimum of the C_0 interval and the maximum of the C_1 interval. The modified pseudocode is

```
bool NoIntersect(float tmax, float speed, Configuration Cfg0,
    Configuration Cfg1, Configuration& Curr0, Configuration& Curr1,
    int& side, float& tfirst, float& tlast)
{
    if (Cfg1.max < Cfg0.min) {
        if (speed <= 0) return true;
        t = (Cfg0.min - Cfg1.max) / speed;
        if (t > tfirst)  {
            tfirst = t;  side = -1;  Curr0 = Cfg0;  Curr1 = Cfg1;
        }
        if (tfirst > tmax  return true;
        t = (Cfg0.max - Cfg1.min) / speed;  if (t < tlast) tlast = t;
        if (tfirst > tlast) return true;
    } else if (Cfg0.max < Cfg1.min) {
```

```
        if (speed >= 0) return true;
        t = (Cfg0.max - Cfg1.min) / speed;
        if (t > tfirst)  {
            tfirst = t;  side = +1;  Curr0 = Cfg0;  Curr1 = Cfg1;
        }
        if (tfirst > tmax) return true;
        t = (Cfg0.min - Cfg1.max) / speed;  if (t < tlast) tlast = t;
        if (tfirst > tlast) return true;
    } else {
        if (speed > 0) {
            t = (Cfg0.max - Cfg1.min) / speed;  if (t < tlast) tlast = t;
            if (tfirst > tlast) return true;
        } else if (speed < 0) {
            t = (Cfg0.min - Cfg1.max) / speed;  if (t < tlast) tlast = t;
            if (tfirst > tlast) return true;
        }
    }
    return false;
}
```

With the indicated modifications, `TestIntersection` has the equivalent formulation:

```
bool TestIntersection(ConvexPolygon C0, Point W0, ConvexPolygon C1, Point W1,
                      float tmax, float& tfirst, float& tlast)
{
    W = W1 - W0;  // process as if C0 stationary and C1 moving
    tfirst = 0;  tlast = INFINITY;

    // process edges of C0
    for (i0 = 0, i1 = C0.N - 1; i0 < C0.N; i1 = i0, i0++) {
        D = Perp(C0.E(i1)); // = C0.V(i0) - C0.V(i1));
        ProjectNormal(C0, D, i1, Cfg0);
        ProjectGeneral(C1, D, Cfg1);
        speed = Dot(D, W);
        if (NoIntersect(tmax, speed, Cfg0, Cfg1, Curr0, Curr1, side, tfirst,
                        tlast))
            return false;
    }

    // process edges of C1
    for (i0 = 0, i1 = C1.N - 1; i0 < C1.N; i1 = i0, i0++) {
        D = Perp(C1.E(i1)); // = C1.V(i0) - C1.V(i1));
        ProjectNormal(C1, D, i1, Cfg1);
```

```
        ProjectGeneral(C0, D, Cfg0);
        speed = Dot(D, W);
        if (NoIntersect(tmax, speed, Cfg0, Cfg1, Curr0, Curr1, side, tfirst,
                        tlast))
            return false;
    }

    return true;
}
```

The `FindIntersection` pseudocode has exactly the same implementation as `TestIntersection`, but with one additional block of code after the two loops that is reached if there will be an intersection. When the polygons will intersect at time T, they are effectively moved with their respective velocities and the contact set is calculated. Let $U_i^{(j)} = V_i^{(j)} + T\vec{w}^{(j)}$ represent the polygon vertices after motion. In the case of edge-edge contact, for the sake of argument suppose that the contact edges are $\vec{e}_0^{\,(0)}$ and $\vec{e}_0^{\,(1)}$. Figure 7.18 illustrates the configurations for two triangles: Because of the counterclockwise ordering of the polygons, observe that the two edge directions are parallel, but in opposite directions. The edge of the first polygon is parameterized as $U_0^{(0)} + s\vec{e}_0^{\,(0)}$ for $s \in [0, 1]$. The edge of the second polygon has the same parametric form, but with $s \in [s_0, s_1]$ where

$$s_0 = \frac{\vec{e}_0^{\,(0)} \cdot \left(U_1^{(1)} - U_0^{(0)}\right)}{||\vec{e}_0||^2} \quad \text{and} \quad s_1 = \frac{\vec{e}_0^{\,(0)} \cdot \left(U_0^{(1)} - U_0^{(0)}\right)}{||\vec{e}_0||^2}$$

The overlap of the two edges occurs for $\bar{s} \in I = [0, 1] \cap [s_0, s_1] \neq \emptyset$. The corresponding points in the contact set are $V_0^{(0)} + T\vec{w}^{(0)} + \bar{s}\vec{e}_0^{\,(0)}$.

In the event the two polygons are initially overlapping, the contact set is more expensive to construct. This set can be constructed by standard methods involving Boolean operations on polygons.

The pseudocode is shown below. The intersection is a convex polygon and is returned in the last two arguments of the function. If the intersection set is nonempty, the return value of the function is `true`. The set must itself be convex. The number of vertices in the set is stored in `quantity`, and the vertices in counterclockwise order are stored in the array `I[]`. If the return value is `false`, the last two arguments of the function are invalid and should not be used.

```
bool FindIntersection(ConvexPolygon C0, Point W0, ConvexPolygon C1, Point W1,
    float tmax, float& tfirst, float& tlast, int& quantity, Point I[])
{
    W = W1 - W0;  // process as if C0 stationary and C1 moving
    tfirst = 0;  tlast = INFINITY;
```

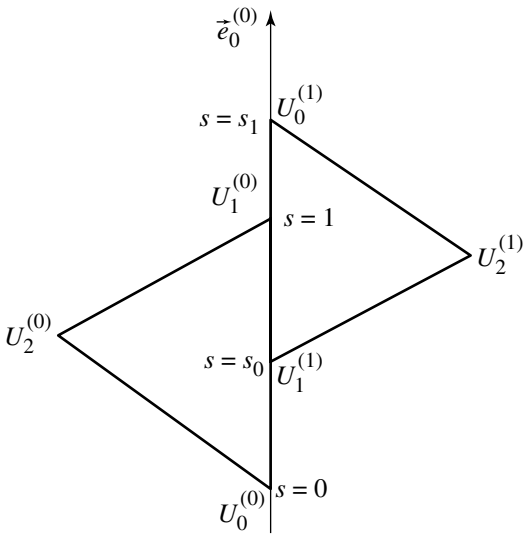

Figure 7.18 Edge-edge contact for two moving triangles.

```
// process edges of C0
for (i0 = 0, i1 = C0.N - 1; i0 < C0.N; i1 = i0, i0++) {
    D = Perp(C0.E(i1)); // C0.E(i1) = C0.V(i0) - C0.V(i1));
    ProjectNormal(C0, D, i1, Cfg0);
    ProjectGeneral(C1, D, Cfg1);
    speed = Dot(D, W);
    if (NoIntersect(tmax, speed, Cfg0, Cfg1, Curr0, Curr1, side, tfirst,
                    tlast))
        return false;
}

// process edges of C1
for (i0 = 0, i1 = C1.N - 1; i0 < C1.N; i1 = i0, i0++) {
    D = Perp(C1.E(i1)); // C1.E(i1) = C1.V(i0) - C1.V(i1));
    ProjectNormal(C1, D, i1, Cfg1);
    ProjectGeneral(C0, D, Cfg0);
    speed = Dot(D, W);
    if (NoIntersect(tmax, speed, Cfg0, Cfg1, Curr0, Curr1, side, tfirst,
                    tlast))
        return false;
}
```

```
    // compute the contact set
    GetIntersection(C0, W0, C1, W1, Curr0, Curr1, side, tfirst, quantity, I);
    return true;
}
```

The intersection calculator pseudocode is shown below. Observe how the projection types are used to determine if the contact is vertex-vertex, edge-vertex, or edge-edge.

```
void GetIntersection(ConvexPolygon C0, Point W0, ConvexPolygon C1, Point W1,
    Configuration Cfg0, Configuration Cfg1, int side, float tfirst,
    int& quantity, Point I[])
{
    if (side == 1) {  // C0-max meets C1-min
        if (Cfg0.type[1] == 'V') {
            // vertex-vertex or vertex-edge intersection
            quantity = 1;
            I[0] = C0.V(Cfg0.index[1]) + tfirst * W0;
        } else if (Cfg1.type[0] == 'V') {
            // vertex-vertex or edge-vertex intersection
            quantity = 1;
            I[0] = C1.V(Cfg1.index[0]) + tfirst * W1;
        } else {  // Cfg0.type[1] == 'E' && Cfg1.type[0] == 'E'
            // edge-edge intersection
            P = C0.V(Cfg0.index[1]) + tfirst * W0;
            E = C0.E(Cfg0.index[1]);
            U0 = C1.V(Cfg1.index[0]);
            U1 = C1.V((Cfg1.index[0]+ 1) % C1.N);
            s0 = Dot(E, U1 - P) / Dot(E, E);
            s1 = Dot(E,U0 - P) / Dot(E, E);
            FindIntervalIntersection(0, 1, s0, s1, quantity, interval);
            for (i = 0; i < quantity; i++)
                I[i] = P + interval[i] * E;
        }
    } else if (side == -1) { // C1-max meets C0-min
        if (Cfg1.type[1] == 'V') {
            // vertex-vertex or vertex-edge intersection
            quantity = 1;
            I[0] = C1.V(Cfg1.index[1]) + tfirst * W1;
        } else if (Cfg0.type[0] == 'V') {
            // vertex-vertex or edge-vertex intersection
            quantity = 1;
            I[0] = C0.V(Cfg0.index[0]) + tfirst * W0;
        } else { // Cfg1.type[1] == 'E' && Cfg0.type[0] == 'E'
```

```
            // edge-edge intersection
            P = C1.V(Cfg1.index[1]) + tfirst * W1;
            E = C1.E(Cfg1.index[1]);
            U0 = C0.V(Cfg0.index[0]);
            U1 = C0.V((Cfg0.index[0] + 1) % C0.N);
            s0 = Dot(E, U1 - P) / Dot(E, E);
            s1 = Dot(E, U0 - P) / Dot(E, E);
            FindIntervalIntersection(0, 1, s0, s1, quantity, interval);
            for (i = 0; i < quantity; i++)
                I[i] = P + interval[i] * E;
        }
    } else { // polygons were initially intersecting
        ConvexPolygon C0Moved = C0 + tfirst * W0;
        ConvexPolygon C1Moved = C1 + tfirst * W1;
        FindPolygonIntersection(C0Moved, C1Moved, quantity, I);
    }
}
```

The final case occurs when the two polygons were initially overlapping, so the first time of contact is $T = 0$. `FindPolygonIntersection` is a general routine for computing the intersection of two polygons.