Geometric Algorithms
Assignment 2

Laura Baakman (s1869140)

October 29, 2014

A

The Intersection of two Line Segments

From here on we will define the cross product of two dimension vectors $\mathbf{v}$ and $\mathbf{w}$ as following:

$$\mathbf{v} \times \mathbf{w} = -v_2 w_1 + v_1 w_2 \tag{1}$$

where q_n represents the n'th element of vector $\mathbf{q}$.

In this section we will consider the intersection of the line segments l_1 and l_2 [3] which are defined as:

$$l_1 = \mathbf{p} + t\mathbf{r} \tag{2}$$
$$l_2 = \mathbf{q} + u\mathbf{s}. \tag{3}$$

Any point lies on l_1 if and only if that point can be expressed as (2) with $0 \leq t \leq 1$. Using this we can define the intersection $\mathbf{I}$ of the two line segments as following: the line segments l_1 and l_2 intersect if we can find values for t and u such that $t, u \in [0, 1]$ and:

$$\mathbf{p} + t\mathbf{r} = \mathbf{q} + u\mathbf{s} \tag{4}$$

Rewriting this equation gives us expressions for u and t:

$$t = \frac{\mathbf{q} - \mathbf{p} \times \mathbf{s}}{\mathbf{r} \times \mathbf{s}} \tag{5}$$
$$u = \frac{\mathbf{q} - \mathbf{p} \times \mathbf{r}}{\mathbf{r} \times \mathbf{s}} \tag{6}$$

If the denominator, $(\mathbf{r} \times \mathbf{s})$, of (5) or (6) is zero the lines are parallel, since the cross product of two parallel vectors is zero.

If we know that we are not dividing by zero we can compute u and t and check if they are in the range $[0, 1]$.

This intersection test is implemented in the class `LineSegment`, see Listing 2 which is part of the module `utilities`. The code to compute `r_cross_s`, `u_numerator` and `t_numerator` was generated with Mathematica, see Listing 1.

Point in a Polygon

Since we know that the polygon P is convex we can test quite simply if the point p is inside the polygon by translating the polygon so that p becomes the origin of P. The point p is now in the

Listing 1: Mathematica code used to compute the value of `r_cross_s`, `u_numerator` and `t_numerator`.

```
    rExtended = {r000, r001, 0};
    sExtended = {s000, s001, 0};
    qExtended = {q000, q001, 0};
    pExtended = {p000, p001, 0};

    rCrossS = Part[Cross[rExtended, sExtended], 3];
    tNumerator = Part[Cross[(qExtended - pExtended), sExtended], 3];
    uNumerator = Part[Cross[(qExtended - pExtended), rExtended], 3];
```

Listing 2: The class `LineSegment`. It should be noted that `division` has been imported from `__future__`.

```
class LineSegment(object):

    """This class stores a line as a vector and a point on the line."""

    def __init__(self, points):
        """
        Construct a LineSegment object.

        Input:
            points: list of two points of the form [[x1, y1], [x2, y2]].
        """
        super(LineSegment, self).__init__()
        [p1, p2] = points
        self.vector = [-p1[0] + p2[0], -p1[1] + p2[1]]
        self.point = p1

    def intersect_line_segment(self, other):
        """Find the intersection of this LineSegment with other."""
        p = self.point
        r = self.vector
        q = other.point
        s = other.vector

        r_cross_s = -(r[1]*s[0]) + r[0]*s[1]

        if(r_cross_s):
            u_numerator = p[1]*r[0] - q[1]*r[0] - p[0]*r[1] + q[0]*r[1]

            u = u_numerator / r_cross_s
            if (u >= 0 and u <= 1):
                t_numerator = p[1]*s[0] - q[1]*s[0] - p[0]*s[1] + q[0]*s[1]
                t = t_numerator / r_cross_s
                if (t >= 0 and t <= 1):
                    x = p[0] + r[0] * t
                    y = p[1] + r[1] * t
                    return [x, y]
        return None
```

polygon if all angles of the from the origin to the vertices of the polygon are in the range $[0, \pi]$. Since we are only interested in the sign of the angle it suffices to take the outer product as defined in Equation 1. If the signs of all these cross products are equal the point p lies inside the polygon P. [1] The method `point_in_polygon` in the module `utilities` uses this method to test if a point lies in a polygon, see Listing 3.

Listing 3: The method `point_in_polygon`.

```
def point_in_polygon(point, polygon):
    """
    Return true if the point point is contained in the polygon polygon.

    Input:
        point: a 2D point as [x,y]
        polygon: a list of n points in CCW order: [[x1, y1], ..., [xn, yn]]
    """
    polygon_translated = [[vertex[0] - point[0], vertex[1] - point[1]] for vertex in polygon]
    polygon_shift = polygon_translated[1:]
    polygon_shift.append(polygon_translated[0])
    area = [
        1
        for (a, b)
        in zip(polygon_translated, polygon_shift)
        if (b[0] * a[1] - a[0] * b[1]) < 0
    ]
    return sum(area) in [0, len(polygon)]
```

Point in a Half-Plane

To test if a point lies inside a half-plane defined by an edge we have introduced the method `vertex_in_half_plane` in the module `utilities`, see Listing 4. This method uses the definition of the half plane given by O'Rourke et al.

Listing 4: The method `vertex_in_half_plane`.

```
def vertex_in_half_plane(vertex, half_plane):
    """
    Return true if the vertex vertex lies in the half plane half_plane.

    Input:
        vertex: a 2D vertex as [x,y]
        half-plane: a vector as its begin point (bx, by) and its endpoint
            (ex, ey) in a list: [[bx, by], [ex, ey]].
    """
    [p_min, p] = half_plane
    return (
        (p[1] * p_min[0] - p[0] * p_min[1] - p[1] * vertex[0] +
            p_min[1] * vertex[0] + p[0] * vertex[1] - p_min[0] * vertex[1]) >= 0
    )
```

The Algorithm

The algorithm presented by O'Rourke et al. finds intersections of the edge $\dot{p}$ and $\dot{q}$ by advancing the edge that points in the direction of the other edge. If neither $\dot{p}$ points at $\dot{q}$ nor $\dot{q}$ points at $\dot{p}$ the outside edge is advanced.

Implementation

The implementation of the algorithm presented by O'Rourke et al. [2] is implemented in the class `ConvexPolygonIntersection`. `_algorithm_init` executes all the code before the start of the loop in the algorithm. Each call of `_algorithm_step` executes one step of the algorithm. `_algorithm_finalize` handles the case where more than $2 \cdot (|P| + |Q|)$ steps have been taken. The code closesly follows the pseudo code presented by O'Rourke et al.

Since p_+, p_-, $\dot{p}$, q_+, q_-, $\dot{q}$ are all derived from p and q I have only stored the index of the current p and q in the variables `_p_idx` and `_q_idx`. To easily gain access to the derived variables getters and setters have been defined, see Listing 8.

Listing 5: The method `_algorithm_init` in the class `ConvexPolygonIntersection`.

```
def algorithm_init(self):
    """Initialization of the algorithm."""
    self._p_idx = 0
    self._q_idx = 0
```

Listing 6: The method `_algorithm_step` in the class `ConvexPolygonIntersection`.

```
def algorithm_step(self):
    """
    Step of the algorithm.

    Returns the intersection(s) or none is no intersection was found.
    """
    def q_dot_cross_p_dot():
        """Compute the dot product of q_dot and p_dot."""
        p = self.get_p()
        q = self.get_q()
        p_min = self.get_p_min()
        q_min = self.get_q_min()
        return (
            p[1] * q[0] - p_min[1] * q[0] - p[0] * q[1] + p_min[0] * q[1] -
            p[1] * q_min[0] + p_min[1] * q_min[0] + p[0] * q_min[1] - p_min[0] * q_min[1]
        )

    intersection = LineSegment(self.get_p_dot()).intersect_line_segment(
        LineSegment(self.get_q_dot()))
    inside = None
    if(intersection):
        if(not self._first_intersection):
            self._first_intersection = intersection
        else:
            if(self._first_intersection == intersection):
                raise StopIteration(
                    'The current intersection is equal to the first intersection.'
                )
        if(vertex_in_half_plane(self.get_p(), self.get_q_dot())):
            inside = 'P'
        else:
            inside = 'Q'
        self.intersections.append(intersection)
    if(q_dot_cross_p_dot() >= 0):
        if(vertex_in_half_plane(self.get_p(), self.get_q_dot())):
            intersection2 = self.advance_q(inside)
        else:
            intersection2 = self.advance_p(inside)
    else:
        if(vertex_in_half_plane(self.get_q(), self.get_p_dot())):
            intersection2 = self.advance_p(inside)
        else:
            intersection2 = self.advance_q(inside)

    if(intersection2):
        self.intersections.append(intersection2)
```

Listing 7: The method `_algorithm_finalize` in the class `ConvexPolygonIntersection`.

```
def algorithm_finalize(self):
    """
    Finalization of the algorithm.

    Test if one polygon is contained in the other.
    """
    if(point_in_polygon(self.get_p(), self.Q)):
        self.intersections = self.P
    elif(point_in_polygon(self.get_q(), self.P)):
        self.intersections = self.Q
```

Listing 8: The getters in the class `ConvexPolygonIntersection`.

```
    def get_p_min(self):
        """Return p min."""
        card_p = len(self.P)
        return self.P[(self._p_idx - 1 + card_p) % card_p]

    def get_q_min(self):
        """Return q min."""
        card_q = len(self.Q)
        return self.Q[(self._q_idx - 1 + card_q) % card_q]

    def get_p_plus(self):
        """Return p plus."""
        return self.P[(self._p_idx + 1) % len(self.P)]

    def get_q_plus(self):
        """Return p plus."""
        return self.Q[(self._q_idx + 1) % len(self.Q)]

    def get_p(self):
        """return p."""
        return self.P[self._p_idx]

    def get_q(self):
        """return q."""
        return self.Q[self._q_idx]

    def get_p_dot(self):
        """Return the begin and endpoints of the vector pdot."""
        return [self.get_p_min(), self.P[self._p_idx]]

    def get_q_dot(self):
        """Return the begin and endpoints of the vector qdot."""
        return [self.get_q_min(), self.Q[self._q_idx]]
```

General Implementation

To be able to give a step by step visualization of the algorithm `ConvexPolygonIntersection` is implemented as an iterator. This allows the user to simply call the method **`next`** on the `PolygonIntersection` object.

The `__init__` method, see Listing 9, of the iterator initializes the iterator and ensures that number of steps is limited before calling the earlier presented `_algorithm_init`.

The **`next`** method (Listing 10) of the iterator increases the step counter and checks if another step is allowed. If allowed it calls `_algorithm_step`. If no more steps are allowed `_algorithm_finalize` is called before raising a `StopIteration` exception.

The iterator is initiated by calling its constructor with two polygons of which the intersection needs to be computed. Storing the constructed object globally allows us to call **`next`**`()` in `display()` when a certain key is pressed.

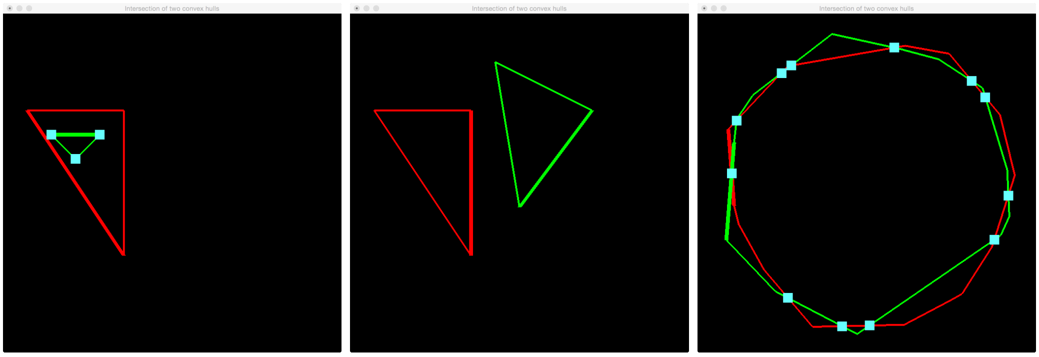
Listing 9: The method `__init__` in the class `PolygonIntersection`.

```
    def __init__(self, set_P, set_Q):
        """Constructor of the class convexPolygonIntersection.

        Input:
            set_P: vertices of a convex polygon in CCW order.
            set_Q: vertices of a convex polygon in CCW order.
        """
        super(ConvexPolygonIntersection, self).__init__()
        self.P = set_P
        self.Q = set_Q
        self._max_steps = 2 * (len(self.P) + len(self.Q))
        self._current_step = 0
        self._first_intersection = None
        self.intersections = []
        self.algorithm_init()
```

Listing 10: The method **next** in the class `PolygonIntersection`.

```
def next(self):
    """Take the next step."""
    if(self._current_step <= self._max_steps):
        self.algorithm_step()
    else:
        self.algorithm_finalize()
        raise StopIteration('Executed the maximum number of steps.')
    self._current_step = self._current_step + 1
```

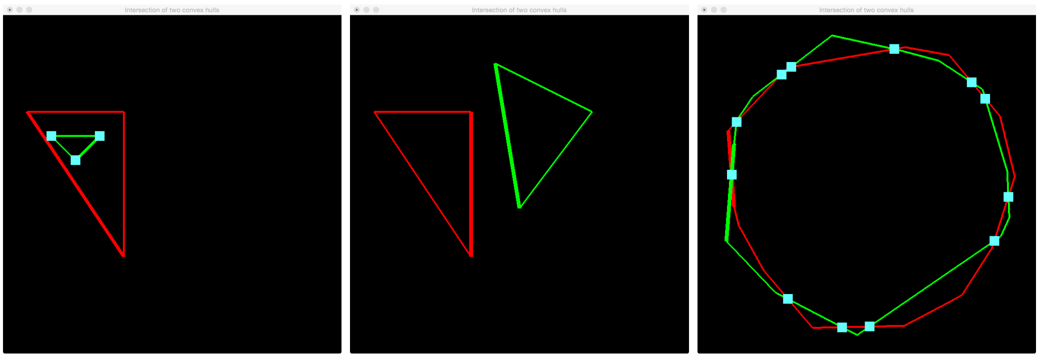


(a) contained **(b)** disjoint **(c)** intersecting

Figure 1: The final stage of the algorithm for: (a) two polygons where one is contained within the other, (b) two disjoint polygons and (c) two intersecting polygons. The found intersections points are coloured blue. The thicker edges denote the $\dot{p}$ and $\dot{q}$ at the last step.

To test the algorithm we have used it on two polygons that were, disjoint, two that were intersecting and on a polygon that contained the other polygon see Figure 1. Changing the initial q to the second to last vertex in the list of vertices representing the polygon yields the intersection in a different order and can change the final $\dot{q}$ and $\dot{p}$, see Figure 2.

Executing the algorithm on the provided sets `P` and `Q` results in the following set of intersections: [(70.6, 330), (80.7, 221), (174, 123), (194, 107), (407, 70.3), (567, 139), (595, 173), (643, 376), (614, 467), (356, 644), (299, 646), (187, 587)]



(a) contained **(b)** disjoint **(c)** intersecting

Figure 2: The final stage of the algorithm for the same polygons as in Figure 1 but with a different initial p and q.

B

References

[1] Robert Nowak. *An Efficient Test for a Point to Be in a Convex Polygon*. URL: http://demonstrations.wolfram.com/AnEfficientTestForAPointToBeInAConvexPolygon/ (visited on 10/28/2014).

[2] Joseph O'Rourke et al. "A new linear algorithm for intersecting convex polygons". In: *Computer Graphics and Image Processing* 19.4 (1982), pp. 384–391.

[3] Gareth Rees. *How do you detect where two line segments intersect?* URL: http://stackoverflow.com/questions/563198/how-do-you-detect-where-two-line-segments-intersect (visited on 09/18/2014).