Geometric Algorithms
Assignment 2

Laura Baakman (s1869140)

September 25, 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 [2] 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. This degenerate case will be discussed later on in this report.

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` of which a part is shown in Listing 2. The code to compute `r_cross_s`, `u_numerator` and `t_numerator` was generated with Mathematica, see Listing 1.

A.1 Point in a Polygon

To test if a point q lies in a polygon P we can count the number of intersections with edges of P of a ray that starts at q and ends at some fixed point. If we have entered the polygon as many

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: Part of 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, vector, point, **kwargs):
        """Construct a LineSegment object."""
        super(LineSegment, self).__init__()
        self.vector = vector
        self.point = point

    @classmethod
    def from_point_list(cls, points):
        """Return a LineSegment object from p1 to p2."""
        [p1, p2] = points
        vector = [-p1[0] + p2[0], -p1[1] + p2[1]]
        return cls(vector, p1)

    def intersect(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 False
```

Listing 3: Part of the class `LineSegment`. It should be noted that `division` has been imported from `__future__`.

```
    def intersect_with_ray(self, ray):
        """Test if this line segments intersects with a line represented as a vector."""
        pdb.set_trace()
        p = self.point
        r = self.vector
        q = ray
        s = [-q[0], -q[1]]

        r_cross_s = -(r[1]*s[0]) + r[0]*s[1]
        if(r_cross_s):
            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):
                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):
                    return True
        return False
```

Listing 4: The method `_algorithm_init` in the class `PolygonIntersection`.

```
    def _algorithm_init(self):
        """Initialize the algorithm by selecting a random p and q."""
        global p, q, p_min, q_min, p_dot, q_dot
        p_min, q_min = len(P) - 1, len(Q) - 1
        p_dot = [P[p_min], P[p]]
        q_dot = [Q[q_min], Q[q]]
```

times as we have exited it, we are thus outside the polygon. If the number of intersections is odd the point lies inside the polygon.

The intersection of a ray and a line segment, the edges of P, is essentially the same as the intersection of two line segments with the notable difference that the scaling parameter of the ray should be greater than zero instead of greater than or equal to zero and smaller than or equal to one.

If the scaling parameter is zero the point lies on a edge, or vertex of the polygon instead of inside it. The code used to test if a line segment and a ray intersect is presented in Listing 3. It should be noted that we require $t \in [0, 1)$ to avoid counting intersections of a ray and a vertex of the polygon twice.

The code in Listing 3 is used to test if a point q lies in a circle by calling this method on all edges of the polygon in combination with the point q as a parameter. See the method `point_in_polygon` in Listing 6.

The Algorithm

The implementation of the algorithm presented by O'Rourke et al. [1] is split over three methods in the class `PolygonIntersection` namely: `_algorithm_init` (Listing 4), `_algorithm_step` (Listing 5) and `_algorithm_finalize` (Listing 6).

`_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.

These methods closely follow the pseudo-code from O'Rourke et al.

Listing 5: The method `_algorithm_step` in the class `PolygonIntersection`.

```
def _algorithm_step(self):
    """Execute one iteration of the do-while of the algorithm."""
    def in_half_plane(x, plane_dot):
        """Test if the point x is in the half plane defined by plane_dot and plane_min."""
        return (
            -(plane_dot[0][1] * plane_dot[1][0]) + plane_dot[0][0] * plane_dot[1][1] +
            plane_dot[0][1] * x[0] - plane_dot[1][1] * x[0] -
            plane_dot[0][0] * x[1] + plane_dot[1][0] * x[1]
            >= 0
        )

    def advance(advancing_set, inside):
        """Handle the advancement of the active edge."""
        def advance_p():
            """Advance p."""
            global p, p_min, p_dot
            if (inside == 'p'):
                global intersection_points
                intersection_points.append(p)
                pdb.set_trace()
            p = (p + 1) % len(P)
            p_min = (p - 1 + len(P)) % len(P)
            p_dot = [P[p_min], P[p]]

        def advance_q():
            """Advance q."""
            global q, q_min, q_dot
            if (inside == 'q'):
                global intersection_points
                intersection_points.append(q)
                pdb.set_trace()
            q = (q + 1) % len(Q)
            q_min = (q - 1 + len(Q)) % len(Q)
            q_dot = [Q[q_min], Q[q]]

        {
            'p': advance_p,
            'q': advance_q
        }.get(advancing_set)()

    intersection = LineSegment.from_point_list(p_dot).intersect(
        LineSegment.from_point_list(q_dot)
    )

    inside = None

    if(intersection):
        if(not self._first_intersection):
            self._first_intersection = intersection
        elif(intersection == self._first_intersection):
            global intersection_points
            intersection_points.append(intersection)
            pdb.set_trace()
            raise StopIteration(
                "Terminated the algorithm, since we have reached the end of the intersection."
            )
        if(in_half_plane(P[p], q_dot)):
            inside = 'p'
        else:
            inside = 'q'

    q_dot_cross_p_dot = (
        p_dot[0][1] * q_dot[0][0] - p_dot[1][1] * q_dot[0][0] - p_dot[0][0] * q_dot[0][1] +
        p_dot[1][0] * q_dot[0][1] - p_dot[0][1] * q_dot[1][0] + p_dot[1][1] * q_dot[1][0] +
        p_dot[0][0] * q_dot[1][1] - p_dot[1][0] * q_dot[1][1]
    )
    if (q_dot_cross_p_dot >= 0):
        if(in_half_plane(P[p], q_dot)):
            advance('q', inside)
        else:
            advance('p', inside)
    else:
        if(in_half_plane(Q[q], p_dot)):
            advance('p', inside)
        else:
            advance('q', inside)
```

Listing 6: The method `_algorithm_finalize` in the class `PolygonIntersection`.

```
def _algorithm_finalize(self):
    """Finalize the algorithm, is called when no intersection was found."""
    def point_in_polygon(point, polygon):
        """Test if the point lies in the polygon."""
        def intersecting_vectors((p1, p2), point):
            """Test if the linesegment intersects with the ray."""
            ls = LineSegment.from_point_list([p1, p2])
            return ls.intersect_with_ray(point)

        intersected_edges = sum(
            [1 for edge in zip(polygon, polygon[1:] + polygon[:1])
                if intersecting_vectors(edge, point)]
        )
        print intersected_edges
        return not (intersected_edges % 2 == 0)

    global P, Q, intersection_points
    p = random.randint(0, len(P) - 1)
    q = random.randint(0, len(Q) - 1)
    if (point_in_polygon(p, Q)):
        intersection_points = P
    elif (point_in_polygon(q, P)):
        intersection_points = Q
    else:
        intersection_points = []
```

Listing 7: The method `__init__` in the class `PolygonIntersection`.

```
def __init__(self, set_P, set_Q):
    """Construct a PolygonIntersection object."""
    super(PolygonIntersection, self).__init__()
    global P, Q
    P = set_P
    Q = set_Q
    self._current_step = 1
    self._max_steps = 2 * (len(P) + len(Q))
    self._first_intersection = None

    self._algorithm_init()
```

General Implementation

To be able to give a step by step visualization of the algorithm I have made the class in which it is implemented an iterator. This allows the user to simply call the method **`next`** on the `PolygonIntersection` object.

The `__init__` method, see Listing 7, 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 8) 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, see Listing 9. Storing the resulting object, `pg`, globally allows us to call **`next`**`()` in `display()` when a certain key is pressed, see Listing 10.

Executing the algorithm on the provided sets `P` and `Q` results in the following set of intersections:

Listing 8: The method **next** in the class `PolygonIntersection`.

```
def next(self):
    """Take the next step."""
    print "\nStep {}".format(self._current_step)
    if self._current_step <= self._max_steps:
        self._current_step = self._current_step + 1
        self._algorithm_step()
    else:
        self._algorithm_finalize()
        raise StopIteration(
            "Terminated the algorithm, since more than 2 * |P| * |Q| steps have been taken."
        )
```

Listing 9: The construction of the `PolygonIntersection` object.

```
global pg
if argv is None:
    argv = sys.argv
pg = PolygonIntersection(R, S)
```

Listing 10: Going one step forward.

```
if key == 'n':  # do one step of the actual algorithm from the paper
    try:
        pg.next()
    except StopIteration:
        pdb.set_trace()
        "Found the intersection: {}".format(intersection_points)
        raise SystemExit
```

B

-

References

[1] 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.

[2] 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).