Vanden Heede Pauline

Navigation Mesh Generation

Gameplay programming
Research project
2021-2022

Digital Arts and Entertainment
Howest.be

Contents

1 Abstract

2 Introduction

For most game environments it's easily predictable where objects can and can't move[2]. With this information we can create a set of geometry to describe the walkable surface of a 3D environment, called a navigation mesh (nav mesh). The geometry of a nav mesh is build out of a set of convex polygons, this can be triangles or quadrilaterals, the latter giving a more compact and convenient representation [3].

There are different approaches to the generation of navigation meshes.[1]

Manually creating the navigation mesh. Manually creating a navigation mesh is done by a level designer in a level editor. This mesh then gets loaded into the game and can directly be used. Generating the navigation mesh. In this mehtod the raw game geometry is used to build a navigation mesh through code. Combination of manually creating and generating the navigation mesh. To get an optimal navigation mesh, our navigation mesh needs to meet some requirements. The first requirement tells us that the polygons of which the navigation mesh is made need to be convex, as mentioned before. The reason why these need to be convex is simple. This is the only way our navigation mesh can guarantee that an agent inside the polygon can reach any other point inside the polygon in a straight line without colliding with the environment. This gives us the big advantage of fewer expensive collision checks with the environment. Secondly, as we only can go out from the given raw game geometry our navigation mesh generator needs to be robust in handling degeneracy[3]. We cannot expect that data is given in a certain format that aids the generator. This means that we need to have a way to handle incomplete data or data that is different than we expect. If we do not want to go this route we need to query all the elements in the game environment with a certain flag set; e.g. elite engine navigation mesh Other requirements of the navigation mesh are simplicity, completeness and consistent. We want our navigation mesh to have as few polygons as possible that covers the whole area where an AI can go and that there is no randomness, any equal map should have an equal outcome. [3]

[3] [2]

3 Boolean operations on polygons

There are different kind of algorithms to execute the different boolean operations on polygons. These boolean operations are; AND, NOT, UNION, INTERSECTION and XOR. For the navigation mesh, and to filter out the obstructions we need the NOT boolean operation.

- Greiner-Hormann clipping algorithm
- Vatti Clipping algorithm
- Bentley-Ottmann algorithm
- Sutherland-Hodgman algorithm
- Weiler-Atherton clipping algorithm

Next, the different algorithms will be discussed.

3.1 Greiner-Hormann clipping algorithm

This algorithm is used for polygon clipping. More performant than the Vatti clipping algorithm. It's based on the definition of the inside of a polygon based on the winding number. The regions with an odd winding number lay inside the polygon; also known as the even-odd rule.

Winding number or winding index of a closed curve in the plane around a given point is an integer representing the total number of times that curve travels counterclockwise around the point. So it depends on the orientation of the curve, if it's negative then the curve travels around the point clockwise.

INPUT: 2 lists of polygons
ALGORITHM:

1. pairwise intersections between edges of the polygons. Additional vertices are inserted into both polygons at the points of intersection. The intersection hold a pointer to it's counterpart in the other polygon.
2. intersection is marked as an entry intersection of an exit intersection. This can be done by evaluating the even-odd rule at the first vertex, so we know if this vertex is inside or outside the other polygon. Following the polygon's borders, the intersections are marked with alternating flags. The next intersection after an entry intersection must be an exit intersection.
3. Generating the result. It starts at an unprocessed intersection and picks the direction of traversal based on the entry/exit flag; for an entry intersection it traverses forward, and for an exit intersection it traverses in reverse. Vertices are added to the result until the next intersection is found. The algorithm then switches to the corresponding intersection vertex in the other polygon and picks the traversal direction again using the same rule. If the next intersection has been processed, the algorithm finishes the current component of the output and starts again from an unprocessed intersection. The output is complete when there are no more unprocessed intersection.

A big shortcoming of this algorithm is that it cannot handle degeneracies such as common edges or intersections exactly at a vertex.

3.2 Vatti clipping algorithm

clipping of any number of arbitrarily shaped subject polygons by any number of arbitrarily shaped clip polygons. Compared to Sutherland-Hodgman and Weiler-Atherton it is not restricted to the types of polygons. Complex polygons (e.g. self-itnersecting) or with holes in can be processed. Only usable in 2D space.

This algorithm starts with the lowermost edges and works towards the top (conceptually similar to the Bentley-Ottmann algorithm). This approach divides the problem space by scanlines.

3.3 Bentley-Ottmann algorithm

This is a sweep line algorithm for listing all crossings in a set of line segments. It takes $O((n+k)logn)$ time in which k is the number of crossings or intersections and n is the number of line segments. In cases where $k = O(\frac{n^2}{logn})$ it takes $O(n^2)$.

So a vertical line (L) moves from left to right (or from top to bottom) across the plane, intersecting the input line segments in sequence as it moves.

- No two line segment endpoints or crossings have the same x-coordinate
- No line segment endpoint lies upon another line segment
- No three line segments intersect at a single point.

In such case, L will always intersect the input line segments in a set of points whose vertical ordering changes only at a finite set of discrete events. A discrete event can either be associated with an endpoint (left of right) of a line-segment or intersection point of two line-segments. The continuous motion of L can be broken down into a finite sequence of steps. and simulated by an algorithm that runs in a finite amount of time.

2 types of events can happen. L sweeps across an endpoint of a line segment s, the intersection of L with s is added to or removed from the vertically ordered set of intersection points. Events are easy to predict as endpoints are known already from the input to the algorithm. The remaining events occur when L sweeps across a crossing between (or intersection of) two line segments s and t. these events may also be predicted from the fact that, just prior to the event, the points of intersection of L with s and t are adjacent in the vertical ordering of the intersection points.

Algorithm 1 Bentley-Ottmann algorithm

```
procedure INTERSECTION OF LINE SEGMENTS
    Priority queue Q(x-coordinate);
    Binary search tree T(y-coordinate);
    while !Q.Empty() do
        Event = Q.Top();
        if Event.p == left endpoint then
            T.Insert(s);
            r = T.Above(s);
            t = T.Below(s);
            if Q.Find(r,t) then
                Q.Remove(r,t);
                if Cross(s,r) then
                    Q.Add(s,r);
                end if
                if Cross(s,t) then
                    Q.Add(s,t);
                end if
            end if
        end if
        if Event.p == right endpoint then
            r = T.Above(s);
            t = T.Below(s);
            T.Remove(s);
            if Q.Find(r,t) then
                if Cross(r,t) then
                    Q.Add(r,t);
                end if
            end if
        end if
        if Event.p == crossing point of s and t then
            T.Swap(s,t);
            r = T.Below(t,s);
            u = T.Above(t,s);
            if r then
                Q.Remove(r,s);
            end if
            if u then
                Q.Remove(t,u);
            end if
            if Cross(r,t) then
                Q.Add(r,t);
            end if
            if Cross(s,u) then
                Q.Add(s,u);
            end if
        end if
    end while
end procedure
```

3.4 Sutherland-Hodgman algorithm

Extends each line of the convex clip polygon in turn and selecting only vertices from the subject polygon that are on the visible side.

Algorithm 2 Sutherland-Hodgman algorithm

```
procedure CLIPPING OF POLYGON(clip polygon, subject polygon)
    List output = subject polygon;
    for Edge in clip polygon do
        List input = output;
        output.Clear();
        int i = 0;
        for i < input.Count() do
            Point current = input[i];
            Point previous = input[i-1 % input.Count()];
            Point intersect = Intersection(previous, current, Edge);
            if current inside Edge then
                if previous not inside Edge then
                    output.Add(intersect);
                end if
                output.Add(current);
            end if
            if previous inside Edge then
                output.Add(intersect);
            end if
        end for
    end for
end procedure
```

3.5 Weiler-Atherton algorithm

It allows clipping of a subject of candidate polygon by an arbitrarily shaped clipping polygon/area/region.

A polygon needs to meet several preconditions

- Oriented clockwise
- Not self-intersecting
- support holes (counter clockwise polygons inside their parent polygon). Requires additional algorithms to decide which polygons are holes after which merging of the polygons can be performed using a variant of the algorithm.

Algorithm 3 Weiler-Atherton algorithm

```
procedure Clipping of polygon(clip polygon, subject polygon)
    List clipVertices = clip polygon;
    List subjectVertices = subject polygon;
    for Vertex in subjectVertices do Vertex.Clip = Inside(clip polygon, Vertex);
    end for
    List intersections;
    for Edge in subject polygon do
        if Intersect(clip polygon, Edge) then
            clipVertices.Add(Intersection);
            subjectVertices.Add(Intersection);
            intersections.Add(IntersectingEdge, Egde, intersection);
        end if
    end for
    List inboundIntersections;
    for Intersect in intersections do
        if subjectVertices[Intersect.point.index + 1].Clip == false then
            inboundIntersections.Add(Intersect);
        end if
    end for
    for Intersect in inboundIntersections(clockwise order) AND Intersect.point != start position do
    end for
    if inboundIntersections.Count() == 0 then
        State 1 clip polygon inside subject polygon. Return clip polygon for
clipping and subject polygon for merging.
        State 2 subject polygon inside clip polygon. Return subject polygon for
clipping and clip polygon for merging.
        State 3 No overlap. Return none for clipping and both for merging.
    end if
end procedure
```

3.6 Determining intersections of edges between multiple polygons

4 Expanding of polygons

When we want to subtract the obstacles of a level from the generation map, we need to take into account the player radius because we do not want the player to be able to go half into these obstacles. For this we need an algorithm to expand the obstacle polygons.

When we do this, we can do this in different ways. But some ways have limitations.

4.1 Orthogonal polygons

When we know our polygons are axis-aligned, thus being orthogonal, this algorithm is very easy. We can expand the corner, as much in the y-direction as the x-direction, creating the effect on Figure 1 Expanding of an orthogonal polygon.

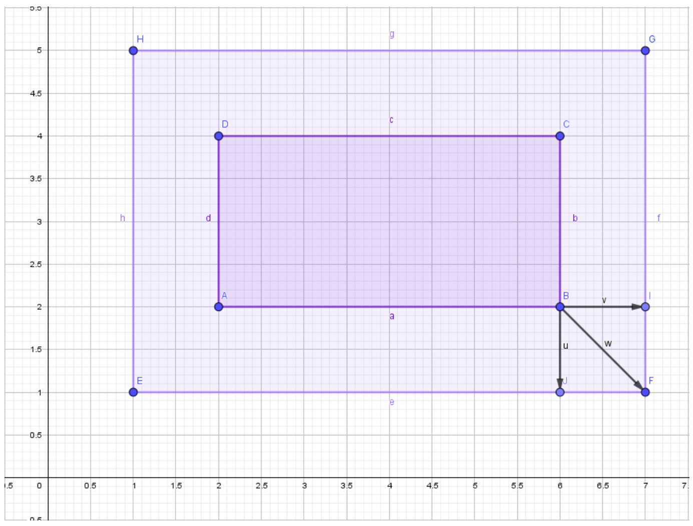

Figure 1: Expanding of an orthogonal polygon.

We can calculate v and u with the points of our polygon. We subtract first and then make it a normalized vector so that we can multiply it with the player radius.

$$v = Normalized(B - A) * offset \tag{1}$$

$$u = Normalized(B - C) * offset \tag{2}$$

We can generalise this to;

$$direction_1 = Normalized(current\ vertex - previous\ vertex) * radius \tag{3}$$

$$direction_2 = Normalized(current\ vertex - next\ vertex) * radius \tag{4}$$

To get to the expanded vertex, we just add these 2 directions.

$$totaldirection = direction_1 + direction_2 \tag{5}$$

$$F = B + totaldirection \tag{6}$$

4.2 Non-Orthogonal polygons

For non-orthogonal polygons, polygons of which the edges are not aligned with the axes, this algorithm becomes more difficult if we want to expand the polygon.

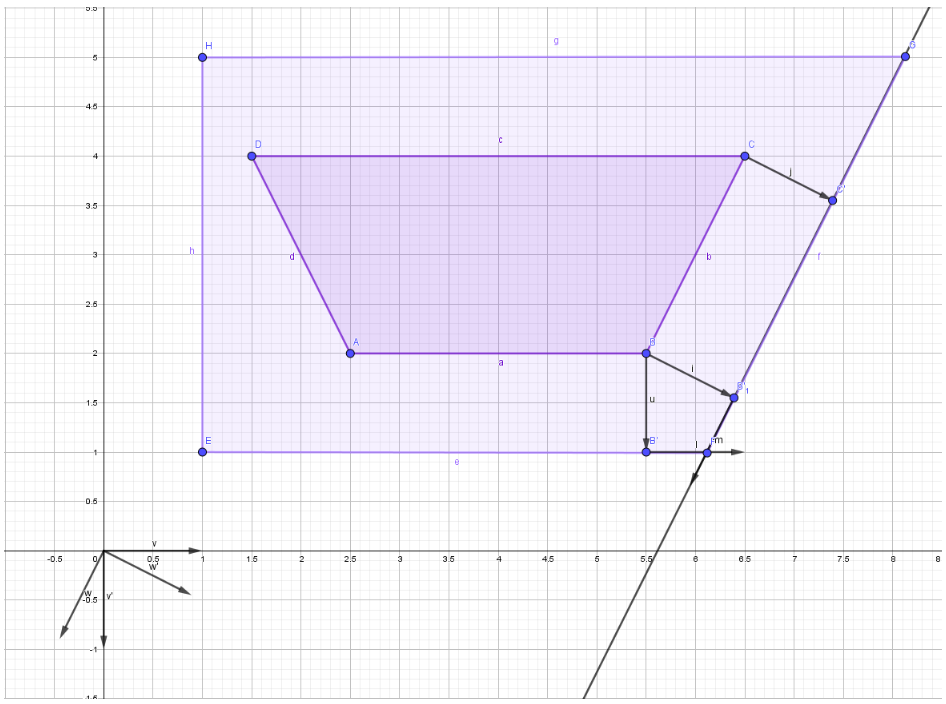

Figure 2: Expanding of a non-orthogonal polygon.

Firstly, the point on the expanded edge parallel to the original edge needs to be calculated, to be able to check for the intersection between 2 lines.

$$v = Normalized(current\, vertex - previous\, vertex) * offset \tag{7}$$

For a 90 degree rotation clockwise;

$$v' = v.y, -v.x \tag{8}$$

$$w = Normalized(current\, vertex - next\, vertex) * offset \tag{9}$$

For a 90 degree rotation counter clockwise;

$$w' = -v.y, v.x \tag{10}$$

We will represent these lines as 2 rays(see the representation of Non-Orthogonal polygons) as this is easier to check for a valid result.

$$Ray = \begin{cases} POINT & origin \\ VECTOR & direction \end{cases}$$

To construct the ray, we have the point B' and B'_1 (see Figure 2 Expanding of a non-orthogonal polygon at p. 2). That can be calculated as follows;

$$B' = B + v' \tag{11}$$

$$B'_1 = B + w' \tag{12}$$

And the direction of both rays is needed, this is the same as the opposite of the vector that is formed by the edges. Which were calculated by v and w. The intersection formula requires the need to write the implicit equation of the ray as a Cartesian equation, this can be done by the following conversion formula.

$$\begin{aligned} ax+&by + c = 0 && (13)\\ a &= ray.direction.y && (14)\\ b &= -ray.direction.x && (15)\\ c &= (ray.point.y * ray.direction.x) - (ray.point.x * ray.direction.y) && (16)\end{aligned}$$

The intersection point then can be found with, for every $a_1 * b_2 - a_2 * b_1 != 0$ [1];

$$x = \frac{-(c_1 * b_2 - c_2 * b_1)}{a_1 * b_2 - a_2 * b_1} \tag{17}$$

$$y = \frac{-(a_1 * c_2 - a_2 * c_1)}{a_1 * b_2 - a_2 * b_1} \tag{18}$$

4.3 Experiments

First we tried the orthogonal solution, but as you can see from Figure 3 Orthogonal solution for expanding of polygons on a non-orthogonal polygon the polygon gets skewed.

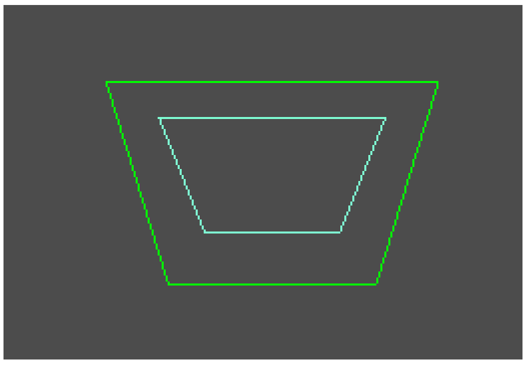

Figure 3: Orthogonal solution for expanding of polygons on a non-orthogonal polygon.

When the non-orthogonal solution is used, it can be seen that the polygon is expanded parallel to all edges, which is the desired result.

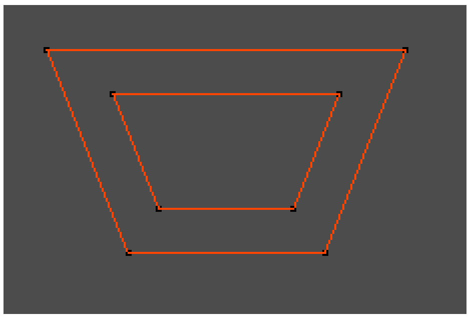

Figure 4: Non-orthogonal solution for expanding of polygons on a non-orthogonal polygon.

To make the difference clear, a figure is included with both expanded polygons on top of each other.

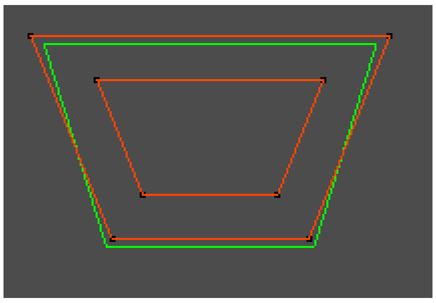

Figure 5: Comparison of both solutions for expanding of polygons.

5 Further Work

References

[1] "Intersection point of lines." (), [Online]. Available: `https://cp-algorithms.com/geometry/lines-intersection.html`.

[2] F. Martínez, C. Ogayar, J. R. Jiménez, and A. J. Rueda, "A simple algorithm for boolean operations on polygons," *Advances in Engineering Software*, vol. 64, pp. 11–19, Oct. 2013, ISSN: 0965-9978. DOI: `10.1016/J.ADVENGSOFT.2013.04.004`.

[3] J. O'Rourke, *Computational Geometry in C*, 2nd ed. 2021, ISBN: 9780511804120.