Polygon Triangulation by Ear Clipping

CS 442/542

Due 11:59 pm, Thursday, September 18, 2015

1 Introduction

For this project you will write a JavaScript program that will triangulate a simple polygon (no holes / bow-ties) as described in Section 2. The program will be interpreted with the `Node.js` JavaScript engine as described in Section 3. You will be given a variety of test cases to test your code with. Section 4 describes how to submit your solution.

2 Triangulation

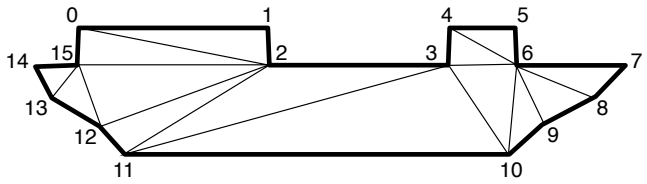

Figure 1: Triangulated polygon.

In order to fill polygons in OpenGL, they need to decomposed into convex polygons – typically quadrilaterals and/or triangles. Triangles are by far the most popular primitive, and the only option for OpenGL ES. Figure 1 shows a concave polygon partitioned into 14 triangles.

2.1 Ear Clipping

An *ear of a polygon* is a triangle formed by three consecutive vertices v_i, v_{i+1}, v_{i+2} where the chord joining v_i and v_{i+2} lies entirely inside the polygon. Figure 2 shows all three ears of a simple (concave) polygon. The *Two-Ears Theorem* tells us that every simple polygon (besides a triangle) has at least two non-overlapping ears. This suggests a simple algorithm for decomposing a simple polygon into triangles: while the polygon has more than three vertices, find a triangular ear and cut it off.

Identifying ears is usually done by first classifying each vertex v_i as either *convex* or *concave* (vertices v_1, v_4, v_5, and v_9 in Figure 2 are concave). Concave vertices (sometimes called "notches") imply that the triangle formed by v_{i-1}, v_i, and v_{i+1} can never be an ear. Note that vertex indices are always non-negative and are represented modulo N; thus the "next" and "previous" vertices are indexed as follows:

$$v_{i+1} = v_{(i+1) \bmod N}, \tag{1}$$

$$v_{i-1} = v_{(i-1+N) \bmod N}. \tag{2}$$

In order to identify the notches, we first need to know if the input polygon is clockwise (CW) or counterclockwise (CCW); If we do not know this up front, we can determine this by the sign of the polygon's

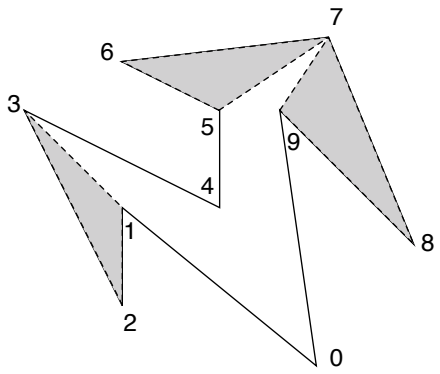

Figure 2: Three ears of a simple polygon.

area:

$$A = \frac{1}{2} \sum_{i=0}^{N-1} (x_i \cdot y_{i+1} - x_{i+1} \cdot y_i) \tag{3}$$

where $v_i = (x_i, y_i)$. If $A > 0$, then the polygon is CCW. Let A_i, be the (signed) area of the triangle representing the ith potential ear. If $A_i \cdot A < 0$, then vertex v_i is a notch. If $A_i \approx 0$ then this triangle is a "sliver" and we should avoid considering this triangle as an ear. We can conclude that v_{i-1}, v_i, v_{i+1} is an ear if v_i is not a notch and none of the remaining vertices lie inside the triangle formed by these vertices.

```
0   if |V| < 3 return empty array (no triangles).
1   Compute area A of input polygon using Equation 3.
2   Build index array I = {0, 1, ..., |V| − 1}.
3   Let array T (initially empty) hold triangle indices.
4   while |I| > 3
5       Let a = 0.0 and n = 0 (area and index of biggest ear)
6       for i = 0 ... |I| − 1
7           Compute area A_i of △v_{I_{i−1}}, v_{I_i}, v_{I_{i+1}}.
8           if |A_i| > a and A_i · A > 0
9               Let ear = true
10              for I_j ∈ I − {I_{i−1}, I_i, I_{i+1}} and ear
11                  ear = v_{I_j} outside △v_{I_{i−1}}, v_{I_i}, v_{I_{i+1}}
12              if ear
13                  n = i
14                  a = |A_i|
15      Add (I_{n−1}, I_n, I_{n+1}) to T.
16      Remove I_n from I.
17  Add last remaining triangle indices (I_0, I_1, I_2) to T.
```

Figure 3: Triangulation Algorithm for simple polygon with (distinct) vertices $V = \{v_0, \ldots, v_{N-1}\}$.

Our simple triangulation algorithm is listed in Figure 3. We begin by computing A for the input polygon $V = \{v_0, \ldots, v_{N-1}\}$ which is used to weed out notches. The array I holds the indices of the "current polygon" and initially references all the vertices. The triangle indices are stored in T – one triangle is added to T during each pass of the while loop beginning on line 4. The algorithm proceeds by cutting off the polygon's largest ear until only three vertex indices remain. n indexes the largest ear and a records its area. Line 8 filters out notches and ears smaller than a. Lines 10 and 11 determine if any of the remaining vertices lie

inside the current triangle (described below).If a new ear has been found we store its index in n and update a with the new ear's area. Relying on the Two Ears Theorem to guarantee that we have found an ear, we save the ear as the next output triangle (Line 15) and remove the index I_n from I (Line 16).

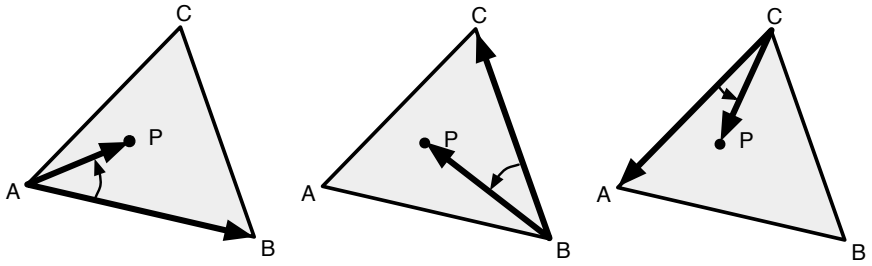

Figure 4: Cross products used to determine if P lies inside $\triangle ABC$.

Line 11 of our ear-clipping algorithm requires us to determine if a point P lies inside a triangle $\triangle ABC$ which is true if (and only if) the three cross products between the vector pairs shown in Figure 4 all point in the same direction. Since these vectors lie in the xy-plane, we are only concerned about the sign of the z-component (the other components are zero):

$$(\mathbf{u} \times \mathbf{v})_z = u_x \cdot v_y - u_y \cdot v_x. \tag{4}$$

Therefore $P \in \triangle ABC$ iff

$$\text{sign}((B - A) \times (P - A))_z = \text{sign}((C - B) \times (P - B))_z = \text{sign}((A - C) \times (P - C))_z. \tag{5}$$

The algorithm listed in Figure 5 determines if the point P lies inside triangle with vertices v_i, v_{i+1}, v_{i+2}.

```
1    Let d = 0
2    for i = 0, 1, 2
3        Let u = v_{i+1} − v_i and w = P − v_i
4        Let z = u_x · w_y − u_y · w_x
5        if i = 0
6            Set d = z
7        else if z · d < 0
8            Quit and answer false
9    Answer true
```

Figure 5: Algorithm to determine if the point P lies inside triangle $\triangle v_i, v_{i+1}, v_{i+2}$.

Note that our ear-clipping algorithm runs in $O(N^3)$ time (there are other triangulation algorithms that run in $O(N \log N)$ time). Since N is generally small and triangulation is performed "off-line" we do not concern ourselves with finding a faster solution.

3 Input/Output

Your Node JavaScript program should read the input polygon vertices from `stdin` in JSON format as shown in Figure 7. The program should then write the output triangle indices as an array of integers in JSON format as illustrated in Figure 8 Use Node's `process.stdin` and `process.stdout` streams for input and output respectively. The Node code below reads from `stdin` in chunks as data becomes available; When all the data is read, the input JSON is parsed and fed to my `triangulate` method whose output is converted to JSON and written to `stdout`.

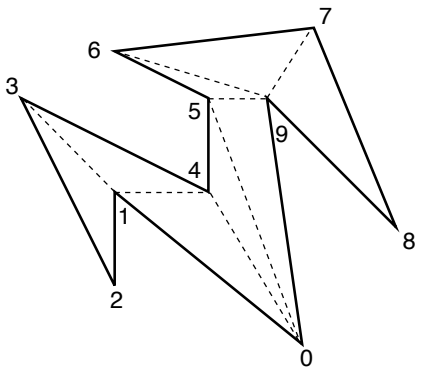

Figure 6: Polygon with 10 vertices decomposed into eight triangles.

```
[
    {"x" : 360.0, "y" : 333.0},
    {"x" : 216.0, "y" : 216.0},
    {"x" : 216.0, "y" : 288.0},
    {"x" : 144.0, "y" : 144.0},
    {"x" : 288.0, "y" : 216.0},
    {"x" : 288.0, "y" : 144.0},
    {"x" : 216.0, "y" : 108.0},
    {"x" : 369.0, "y" : 90.0},
    {"x" : 432.0, "y" : 243.0},
    {"x" : 333.0, "y" : 144.0}
]
```

Figure 7: Input vertices in JSON format for polygon illustrated in Figure 6.

```
process.stdin.setEncoding('utf8');

var inputChunks = [];

process.stdin.on('data', function(chunk) {
    inputChunks.push(chunk);
});

process.stdin.on('end', function() {
    var inputJSON = inputChunks.join();
    var verts = JSON.parse(inputJSON);
    var poly = new Polygon(verts);
    var triangles = poly.triangulate();
    process.stdout.write(JSON.stringify(triangles, null, 4) + '\n');
});
```

I then invoke my script with Node and redirect `stdin/stdout` to/from input/output files:

```
node PolygonTriangulate.js < test.json > triangles.json
```

```
[
    7,8,9,
    6,7,9,
    5,6,9,
    1,2,3,
    1,3,4,
    5,9,0,
    0,1,4,
    0,4,5
]
```

Figure 8: Output triangle indices (in JSON format) for triangulation in Figure 6.

4 What to submit

You will archive all your source code (name the primary script `PolygonTriangulate.js`), documentation, test examples, etc. . . and submit your solution electronically. Include a `README` text file containing the following:

- Name and email of primary author (modification/version history is nice).
- A brief description of what the project is for. Avoid implementation detail and focus on what a new user needs to know to use you program(s).
- How to run program(s) from source code.
- How to run program(s) (an example is nice).
- A list of all files in the archive.