Solutions to Homework 2

Solution 1:
(a) (iii): Allows the mesh to deform smoothly as the joints rotate. When mesh vertices are bound to a single joint, bending the joint can cause the mesh to “crack open.” Binding vertices to multiple joints allows the skin to interpolate smoothly between the movement of nearby joints, thus allowing the mesh to bend smoothly.

(b) In the Ramer-Douglas-Peucker algorithm, the next vertex to be added is the one that has the maximum distance to its associated line segment.

(c) (i) 3: The \((x, y)\) coordinates of the segments reference point, and an angle \(\theta\) to encode its rotation.

(ii) 5 (but I’ll grant partial credit for 6): The obvious answer is six, which involves the \((x, y, z)\) coordinates of the segments reference point, and the Euler angles \((\theta_x, \theta_y, \theta_z)\) that encode its rotation. Observe, however, that you can get away with 5. If one of the coordinate axes is aligned with the line (which is quite reasonable), then one of the rotational degrees of freedom can be eliminated, since rotating a line segment about its central axis does not modify its shape.

(iii) 7: The \((x, y, z)\) of its reference point, the Euler angles \((\theta_x, \theta_y, \theta_z)\) of its rotation, and the opening angle \(\phi\) of the scissors. Note that two opening angles are not needed, because we can think of the reference point as being rigidly attached to one half of the scissors, we need only describe the rotation of the other half.

(d) A path along the medial axis maximizes the distance to the closest obstacle, thus the path (locally) maximizes the clearance from the obstacles.

(e) (i), (ii), and (iii) are all true.

(f) Cohesion is the tendency of the flock to stick together. It can be implemented as a force that draws each boid to the center of mass of the flock.

Solution 2: We compute the configuration obstacle by the process of tracing \(a\) around \(b\), tracing out the path formed by the reference point. The result is shown in Fig. 1.

Solution 3:
(a) We first consider Dijkstra’s algorithm. Table 1 (left) shows the \(d\)-values associated with each discovered node. At each stage (each row of the table), the node with the lowest \(d\)-value (underlined) is selected for processing in the next stage. Undiscovered nodes are indicated using “–” and the \(d\)-values of finished nodes are left blank (since their values do not change).
(b) Next, let us consider the A* algorithm. The $h$ values for the various vertices are $h(s) = 9$, $h(a) = 11$, $h(b) = 6$, $h(c) = 2$, and $h(t) = 0$.

Each entry of Table 1 (right) contains the pair of values $d[u] + h(u)$. (Each $f(u)$-value is just the sum $d(u) + h(u)$.) At each stage the discovered node with the smallest $f$-value is chosen for processing. As above, to indicate that a node is finished, we give its $d$-value as “–” since it will not change again.

(c) Both algorithms generate the same path (of length 11). The A* algorithm terminates one stage earlier than Dijkstra’s, and is more efficient in that sense. (The difference in this example is negligible, but for much larger graphs, the difference can be quite significant. Of course, there is no guarantee that A* will be any better than Dijkstra. It all depends on how good the heuristic value is at estimating the remaining path length.)

Solution 4:

(a) $e' = e.\text{prev}$, $e'' = e.\text{next}$, $a = e.\text{org}$ and $b = e.\text{twin.org}$. (We’ll be lenient in the grading regarding $e'$ and $e''$, since I mislabeled them in my figure in the homework handout.)
(b) Recall that, given a half edge $e$, we can compute the next half edge in counterclockwise order about $e$’s origin node as $e$.onext $\leftarrow$ $e$.prev.twin. Using this operation, the first edge in counterclockwise order about $e$’s origin from the obstacle edge $e'$ is given as

$$e'$.twin.onext = e$.prev.twin.onext.$$

We can enumerate the remaining half edges about $a$ by applying onext operations until reaching $e$. This is shown in the code block below left.

```cpp
// add edges incident to e's origin
while (ei != e) {
    add ei to L;
    ei = ei.onext;
}

// add edges incident to e's destination
while (ei != e.next) {
    add ei to L;
    ei = ei.onext;
}
```

(Referring to Fig. 2(a) the fragment on the left adds the half edges $\langle e_1, e_2 \rangle$ to $L$.) By applying the analogous operation to $e$ in place of $e'$, we obtain the edges incident to $e$’s destination. This is shown in the code block above on the right. (Referring Fig. 2(a), this adds the half edges $\langle e_3, \ldots, e_6 \rangle$.)

(c) Let $p_i$ denote the point that we desire. Recall that $a$ and $b$ are the origin and destination of $e$, respectively. Let $\vec{v}_i = b_i - a_i$ denote the vector directed $a_i$ to $b_i$, and let $\vec{v} = b - a$ be the vector directed from $a$ to $b$. It will simplify things to normalize both of these vectors to unit length by dividing their lengths (e.g., $\vec{v} \leftarrow \vec{v}/|\vec{v}|$). In this manner, we can express any point at distance $\alpha$ from $a_i$ along the edge as $a_i + \alpha \vec{v}_i$.

We consider two cases, depending on the angle $\theta$ between $\vec{v}_i$ and the edge $e$. If this angle is obtuse, then we just take the point at distance $r$ from $a_i$ along the edge (see Fig. 2(b)). If the edge is acute, then if follows from simple trigonometry that the point is at distance $r/\sin \theta$ from $a_i$ (see Fig. 2(c)). All that remains is to compute $\theta$. There are two cases. If $a_i = a$, then $\theta$ is angle between vector $\vec{v}_i$ and $\vec{v}$. Since both have been normalized, we can apply the dot product to obtain $\theta = \arccos(\vec{v}_i \cdot \vec{v})$. On the other hand, if $a_i = b$, then $\theta$ is the angle between $\vec{v}_i$ and $-\vec{v}$. The solution is given in the code block below.
Compute point at distance $r$ from edge $\overline{ab}$ along edge $\overline{ai}$

\[
\begin{align*}
vi &= \text{normalize}(bi - ai); \quad \text{// unit vector from ai to bi} \\
u &= \text{normalize}(b - a); \quad \text{// unit vector from a to b} \\
\text{if } (ai == a) \{ \quad \text{// compute the angle between e and ei} \\
    \theta &= \arccos( \dot{vi}, u ); \\
\} \text{ else } \{ \\
    \theta &= \arccos( \dot{vi}, -u ); \\
\} \\
\text{if } (\theta >= PI/2) \{ \quad \text{// obtuse angle?} \\
    pi &= ai + vi * r; \quad \text{// go distance r from ai along the edge ei} \\
\} \text{ else } \{ \\
    pi &= ai + vi * r / \sin(\theta); \quad \text{// find the point at distance r from edge e along ei} \\
\}
\end{align*}
\]

**Addition Note:** It was pointed out by an alert student in class that this is not complete. If the angle $\theta$ is extremely small (in particular, if it is smaller than $\arctan r/\|b - a\|$) then the edge may travel the entire length of edge $e$ before reaching distance $r$. To handle this, one more case should be added. In this case we need to compute the point on the edge that lies within distance $r$ of $b$. In order for this to happen, the triangle bounding this edge must be obtuse. So, if we assume that the triangulation consists only of acute triangles, this issue will not arise.

(d) Among the two points computing in part (c) for the two edges incident to $ai$, the final point to be selected is the one that is farther from $ai$. By our answer to part (c), we know that each point is expressed in the form $ai + \alpha \vec{vi}$. We choose the one that has the larger value of $\alpha$.

(e) For each edge $(ai, bi)$, from part (d) we know that one of the endpoints is expressed as $pi = ai + \alpha \vec{vi}$ and the other as $qi = bi + \beta(-\vec{vi})$, for some $\alpha, \beta \geq 0$. (We use $-\vec{vi}$ for the other endpoint, since the vector is directed in the reverse direction from $bi$ to $ai$.) Since we assume that $\vec{vi}$ has been normalized to unit length, these points are at distance $\alpha$ and $\beta$ from their respective endpoints. Therefore, the window is nonempty if and only if $\alpha + \beta$ is not greater than the length of the edge, that is, if $\alpha + \beta \leq \|\vec{vi}\|$.

**Solution to the Challenge Problem:** The counterexample to the conjecture that each edge has a single window can be constructed by considering an obtuse triangle $\triangle abc$, such that none of the three edges bound obstacle faces. (By our assumption, each vertex must touch an obstacle.) Suppose that $a$ is the obtuse vertex, then there is a point on the edge $\overline{bc}$ that is closer to $a$ than it is to either $b$ or $c$. By making $r$ just slightly larger than this distance, the edge $\overline{bc}$ is broken into two windows (see Fig. 3(a)).

I assert, however, that if all the edges of the navigation mesh are acute, then each edge has at most one window. Here is a sketch of the proof. If a triangle $\triangle abc$ is acute, then the triangle’s circumcenter (that is, the intersection of its perpendicular bisectors) lies within the triangle. (This follows from the fact that if a triangle is inscribed within a circle, then the angle at each vertex is half the angle of the chord subtended by the opposite side. If all angles are acute, then all the chords subtend arcs smaller than $\pi$, and hence the center is within the triangle.)

If this happens, then by a simple analysis of the perpendicular bisector’s, every point on $\overline{bc}$ is
closer to either $b$ or $c$ than to $a$ (see Fig. 3(b)). Therefore, $a$ cannot contribute to the window of this edge.

Since each vertex can contribute only to the windows of its incident edges, it follows that each edge has at most one window.