CMSC 425: Lecture 15
Motion Panning: Navigation Meshes

Reading: Today's material is partially based on an article of Mikko Mononen, "Navigation Mesh Generation via Voxelization and Watershed Partitioning," 2009.

Motion For the next few lectures we will discuss one of the major issues in the design of AI and algorithms in games, namely planning the motion for non-player characters (NPCs). Motion is a remarkably complex topic, which can range from trivial (e.g., computing a straight-line path between two points in the plane) to very complex tasks, such as:

- Planning the coordinated motion of a group of agents who wish to move to a specified location amidst many obstacles
- Planning the motion of an articulated skeletal model subject to constraints, such as maintaining hand contact with a door handle or avoiding collisions while passing through through a narrow passageway
- Planning the motion of a character while navigating through a dense crowd of other moving people (who have their own destinations), or planning motion either to evade or to pursue the player
- Planning ad hoc motions, like that of a mountain climber jumping over boulders or climbing up the side of a cliff

Historically, much of the initial development of techniques in this area arose from other fields, such as robotics, autonomous vehicle navigation, and computational geometry. Game designers have some advantages in solving these problems, since the environment in which the NPCs move is under the control of the game designer. This means that a game designer can simplify motion planning by creating additional free space in the environment, thus making it easier to plan motion. (In contrast, the designer of an autonomous vehicle cannot remodel the world to make roads wider.) Nonetheless, the techniques that we will present for doing motion planning are broadly applicable, even though they may not need to be applied in their full generality.

Overview: Given the diverse nature of motion planning problems it is not surprising that the suite of techniques is quite large. We will take the approach of describing a few general ideas, that can be applied (perhaps with modifications) across a broad range of problems. These involve the following elements:

Single-object motion:

From objects to points: Methods such as *configuration spaces* can be applied to reduce the problem of moving a complex object (or assembly of objects) with multiple degrees of freedom (DOFs) to the motion of a single point through a multi-dimensional space.

Discretization: Methods such as *waypoints*, *roadmaps*, and *navigation meshes* are used to reduce the problem of moving a point in continuous space to computing a path in discrete graph-like structure.

Shortest paths: This includes efficient algorithms for computing shortest paths, updating them as conditions change, and representing them for later access.

Multiple-object motion:

Flocking: There exist methods for planning basic flocking behavior (as with birds and herding animals) and applications to simulating crowd motion.

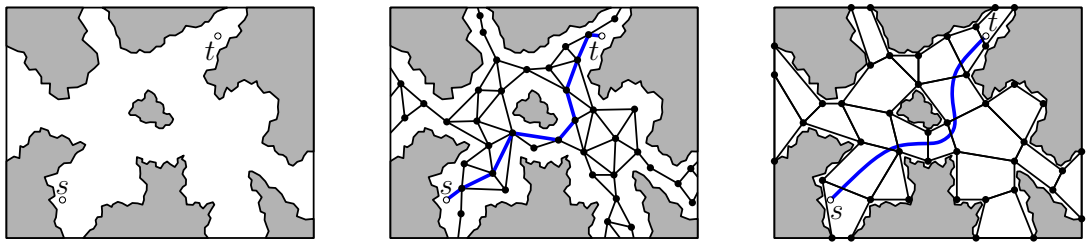
Purposeful crowd motion: Techniques such as *velocity obstacles* are used for navigating a single agent from an initial start point to a desired goal point through an environment of moving agents.

Guarding and Pursuit/Evasion: These include methods for solving motion-planning tasks where one agent is either hunting for or attempting to elude detection by the player or another agent.

Like many of the topics we have covered this semester, we could easily devote an entire course to this one topic, but we will instead try to sample some of the key ideas. In this lecture, we will focus on one of the most widely used concepts from this area, called a *navigation mesh*. This is the principal support feature that the Unity Engine provides for character navigation.

Navigation Meshes: A *navigation mesh* (or *NavMesh*) is a data structure used to model free-space, particularly for an agent that is moving along a two-dimensional surface. (Such a surface is formally referred to as a *two-manifold*). A navigation mesh is a spatial subdivision (more specifically, a cell-complex) whose faces are convex polygons, usually triangles. Each face of the mesh behaves like a node in a graph, and two nodes are joined by an edge if the associated faces share a common edge. Because the faces are convex, any point from inside one face can be reached by a straight line from any other point inside the same face. As with a waypoint system, there is an underlying graph which can be used for computing paths, but by storing the cell complex, the paths computed are not constrained to follow the waypoints.

For example, in Fig. 1(a) we show a possible workspace. In (b) we show a possible waypoint system, and in (c) we show a possible navigation mesh. We show a possible path using each representation between a start point s and destination t.



(a) (b) (c)

Fig. 1: (a) An environment, (b) a possible waypoint-based roadmap, and (c) a possible navigation map.

Because they provide a more faithful representation of the underlying free-space geometry, navigation meshes have a number of advantages over waypoint-based methods:

- They are capable of generating shorter and more natural paths than traditional waypoint methods.
- Waypoint methods can generate an excessive number of points to compensate for their shortcomings, and so navigation meshes can be considerably more space efficient.
- They can be used to plan the movement of multiple (spatially separate) agents, such as a group of people walking abreast of each other. (Note that a waypoint system would need to plan their motion in a single-file line.)
- It is easier to incorporate changes to the environment (such as the insertion, removal, or modification of obstacles).
- A wide variety of pathfinding algorithms can be modified and optimized for using navigation meshes.

Of course, because they are more complicated than waypoint-based methods, there are also disadvantages to the use of navigation meshes:

- They are not as easy to generate as waypoint-based methods.
- Because navigation meshes require a more complex representation of geometry of the scene, the associated pathfinding algorithms are more complex and may take longer to execute.
- They are difficult to generate by hand, and automated generation systems are relatively complex.

Automatic Generation of Navigation Meshes: If the environment is simple, the navigation mesh can be added by the artist who generated the level. Of course, we cannot do this for environments that imported from other sources. If the level is quite large, it is often possible to generate a navigation mesh fairly easily. (Consider for example the sidewalks and roads of an urban scene.) In less structured settings, it is often desirable to generate the navigation mesh automatically. How is this done?

There are many possible approaches to building navigation meshes. We will discuss (a simplified version of) a method due to Mikko Mononen. Let's begin with a few assumptions. First, we assume that the moving agents will be walking along a 2-dimensional surface. This surface need not be flat, and it may contain architectural elements such as ramps, tunnels, and stairways. We will assume that the input is expressed as a polygonal mesh of the world. We will also assume that our moving agent is a walking/running humanoid, and hence can be coarsely modeled as a vertical line segment or a thin cylinder with a vertical axis that translates along this surface.

Find the walkable surfaces: Since we assume that our agent is walking, a polygon is suitable for walking on if (1) the polygon is roughly parallel to the ground, and (2) there is sufficient headroom about this polygon for our agent to walk. Such a polygon is said to be *walkable*. We can identify the polygons that satisfy the first condition by computing the angle between the polygon's (outward pointing) normal vector and the vertical unit vector (see Fig. 2(a)). This angle can be computed through the use of the dot-product operator, as described in earlier lectures.

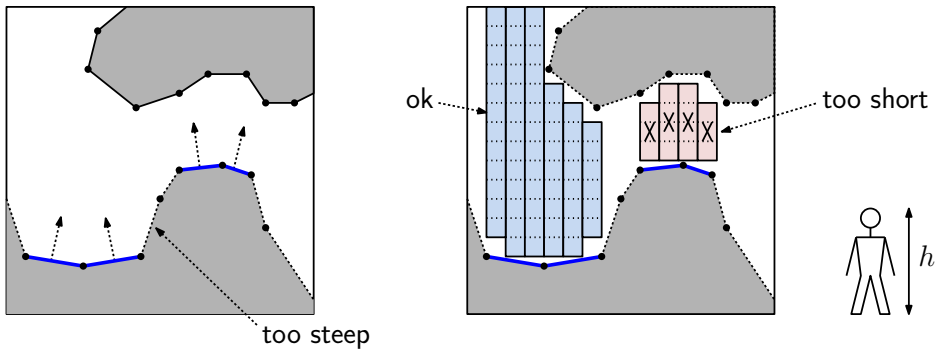


Fig. 2: Walkable surface (side view): (a) Identifying "flat" polygons and (b) voxel method for determining sufficient headroom.

In order to test the the second property, let h denote the height of the agent. Mononen suggests the follows very fast and simple approach. First, voxelize the 3-dimensional space using a grid of sufficient resolution. (For example, the width of the grid should be proportional to the narrowest gap the agent can slip through.) For each polygon that passes condition (1), we determine how many voxels lie immediately above this triangle until hitting the next obstacle (see Fig. 2(b)). (Note that this includes *all* the obstacle polygons, not just the ones that are nearly level.)

Simplify the Polygon Boundaries: Consider the boundary of the walkable surface. (This the boundary between walkable polygons and polygons that are not walkable.) This boundary may generally consist of a very complex polygonal curve with many vertices. We next approximate this curve by a one have much fewer vertices (see Fig. 3(a)).

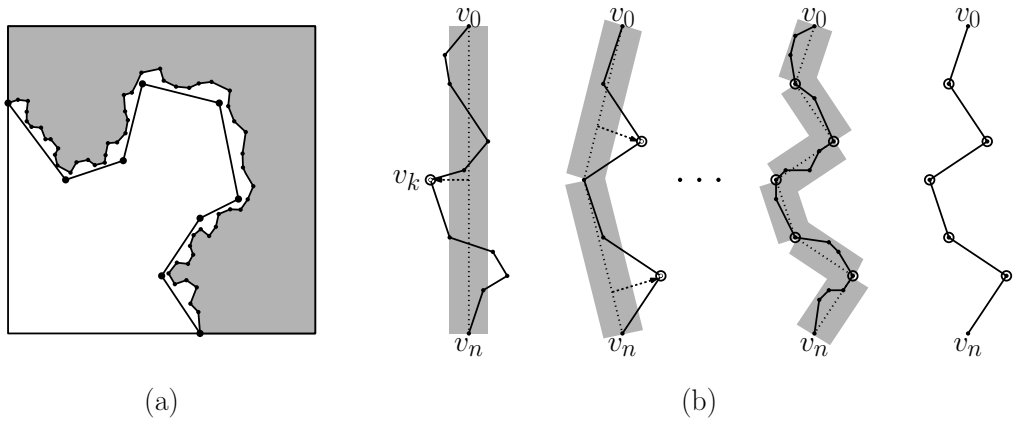


Fig. 3: The Ramer-Douglas-Peucker Algorithm.

There is a standard algorithm for simplifying polygonal curves, called the Ramer-Douglas-Peucker[1] Algorithm. Here is how the algorithm works. First, let δ denote the maximum

[1]The algorithm was discovered independently by Urs Ramer in 1972 and by David Douglas and Thomas Peucker

error that we will allow in our approximation. Suppose that the curve runs between two points v_0 and v_n. If the entire curve fits within a pair of parallel lines at distance δ on either side of the line segment $\overline{v_0v_n}$, then we stop. Otherwise, we find the vertex v_k at maximum distance from this line segment. We add a new vertex at v_k, and then we recursively repeat the algorithm on the two sub-curves $\overline{v_0v_k}$ and $\overline{v_kv_n}$ (see Fig. 3(b)).

Notice that the Ramer-Douglas-Peucker algorithm is not quite what we desired, because it generates a curve that lies on both sides of the original curve. Some modifications are necessary in order to produce a curve that has the property of lying entirely on one side of the original curve, but the principle is essentially the same.

Triangulating the Simplified Polygon: After simplification, we have a collection of polygons, each of which may contain some number of holes (see Fig. 4(a)). The final step is to generate a triangulation of this polygon. Ideally, we would like to have a triangulation in which the triangular elements are relatively "fat." Mononen suggests a very simple heuristic for achieving such a triangulation. (Again, I'll present a variant of his approach.)

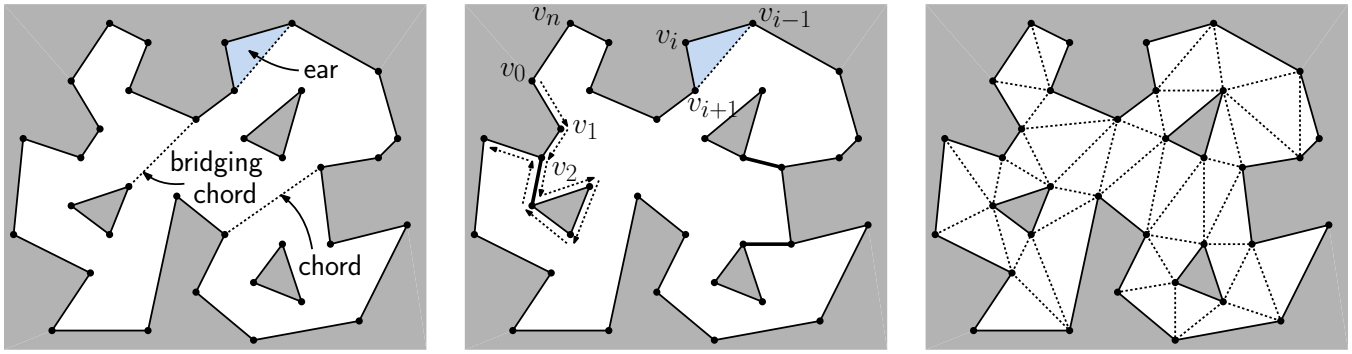


Fig. 4: Triangulating the simplified polygon.

Before presenting the algorithm, let's give a couple of definitions. A line segment that connects two vertices of the polygon and that lies entirely within the interior of the polygon is called a *chord* (see Fig. 4(a)). A chord that connects two holes together or that connects a hole with the outer boundary is called a *bridging chord*. A chord that connects two vertices that share common neighboring vertex cuts of a single triangle from the polygon. This triangle is called an *ear*.

Here is the algorithm:

Bridge the holes: First, connect each hole of the polygon either to another hole or two the boundary of the outer polygon using bridging chords. Repeatedly select the bridging chord of minimum length, until all the holes are connected to the outer boundary. (If there are h holes, this will involve exactly h bridging chords.)

By thinking of each bridging chord a consisting of two edges, one leading into the hole and one leading out, the resulting boundary now consists of one connected

in 1973. Ramer published his result in the computer graphics community and Douglas and Peucker published theirs in the cartography community.

component, which we can treat as a polygon without any holes. Number the vertices $v_0, \ldots, v_n$ in counterclockwise order around this polygon (see Fig. 4(b)).

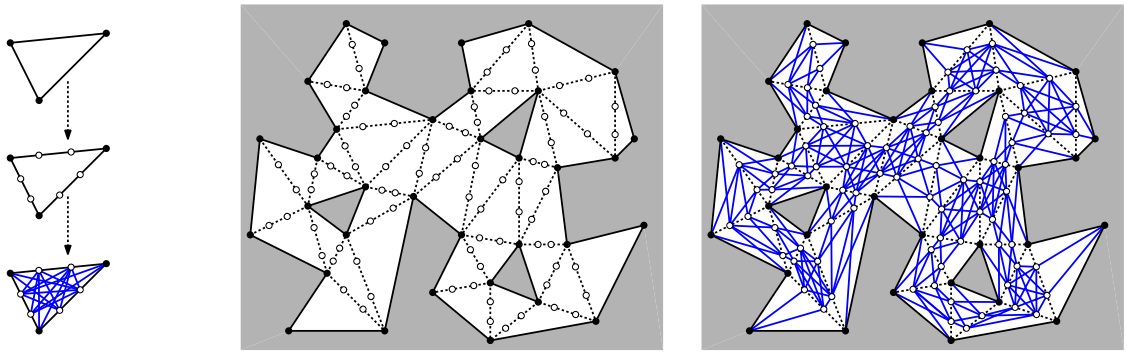
Remove Ears: If the polygon consists of only three vertices, then we are done. Otherwise, find three consecutive vertices v_{i-1}, v_i, v_{i+1} such that $\overline{v_{i-1}v_{i+1}}$ is a chord. The triangle $\triangle v_{i-1}, v_i, v_{i+1}$ is an ear. Among all the possible ears, select the one whose chord is of minimum length. Cut this ear off (ouch!) by adding the chord $\overline{v_{i-1}v_{i+1}}$. The remaining polygon has one fewer vertex. Repeat the process recursively on this polygon, until only three vertices remain. The union of all the removed ears is the final triangulation (see Fig. 4(c)).

By the way, this is but one way to triangulate a polygon with holes. There are many algorithms that are significantly more efficient than this one (from the perspective of worst-case running time). The best such algorithms run in time $O(n \log n)$. If the polygon has no holes, it is possible to triangulate it in $O(n)$ time, but the algorithm is quite complicated, and the $O(n \log n)$ time algorithm is more widely used.

Computing Paths in Polygonal Domains: The final triangulation is the desired navigation mesh. The last detail that remains is how to compute shortest paths in the navigation mesh. In our next lecture we will present algorithms for computing shortest paths in geometric graphs. Assuming that we have such an algorithm, let us next consider how to generalize a graph-based shortest path to a mesh-based shortest path.

Computing a the exact shortest path on a mesh or within a polygonal domain can be solved efficiently in theory, but the algorithm is a bit complicated. We will propose a simpler approach, which produces a good approximate solution.

Discretize: First, distribute a small number of vertices along the each edge of the mesh (see Fig. 5(a) and (b)). Next, for each face of the mesh, form a complete graph by connecting these vertices together (see Fig. 5(a) and (c))).

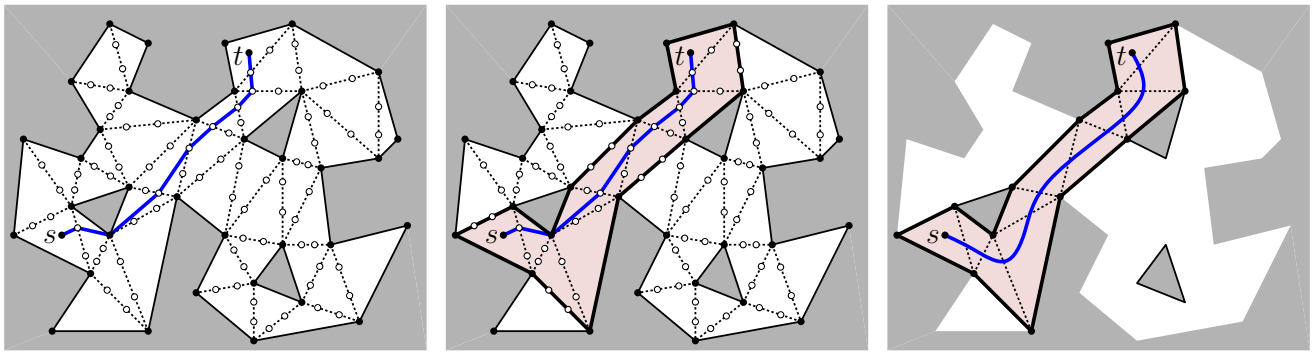


(a) (b) (c)

Fig. 5: Discretizing a polygonal domain for computing shortest paths.

Shorten/Smooth: Compute the shortest path in the resulting graph using any graph-based method (see Fig. 6(a)). Identify a simple polygon (without holes) by combining all the faces of the mesh that this path passes through (see the shaded polygon in Fig. 6(b)).

Finally, apply any further optimizations (such as shortening or smoothing) to the path as desired, subject to the constraint that the path does not leave this shaded polygon (see Fig. 6(c)). Because this polygon has no holes, it is much easier to perform the desired optimizations.



(a) (b) (c)

Fig. 6: (a) The shortest path between s and t, (b) the polygon containing the path (shaded in red), and (c) the final smoothed path.