CMSC 425: Lecture 17
Artificial Intelligence for Games: Planning Motion
Tuesday, Apr 9, 2013

Reading: Some of the material from today's lecture comes from the book "AI Game Programming Wisdom 2" by S. Rabin and "Planning Algorithms" by S. M. LaValle (Chapts. 4 and 5).

Motion For the next couple of lectures, we will discuss one of the major elements of the use of AI in games, planning motion for non-player characters (NPCs). Motion is a remarkably complex topic, which can range from trivial (computing a straight-line path between two points in the plane) to tremendously complex, 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 through a tight passageway
- planning the motion of a character from one location to another who is moving in a dense crowd of other moving people (who have their own destinations)
- planning complex ad hoc motion, like a mountain climber jumping over boulders or climbing up the side of walls

Game designers have some advantages in solving these problems, since the environment in which the NPCs move is under the control of the game designers. Nonetheless, the techniques that we will present for doing motion planning are broadly applicable, even though the may not need to applied in their full generality. (For example, we can place our obstacles farther apart to make it easier for characters to fit between them.)

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* to reduce the problem of moving a complex object (or assembly of objects) to one of moving a single point through space space

Discretization: methods such as *waypoints*, *roadmaps*, and *navigation meshes* to reduce the problem of moving a point in space to computing a path in a graph

Shortest paths: practical and efficient algorithms for computing and representing shortest paths

Multiple-object motion:

Flocking: methods such as *boids* 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* for navigating a single agent from an initial start point to a desired goal point through an environment of moving agents

Configuration Spaces: Let us consider the problem of planning the motion of a single agent among a collection of obstacles. Since the technique that I will be presenting arises from the field of robotics, we'll refer to the agent henceforth as a *robot*. The environment in which the agent operates is called its *workspace*, which consists of a collection of geometric objects, called *obstacles*, which the robot must avoid. We assume that the workspace is static, that is, the obstacles do not move. We also assume that a complete geometric description of the workspace is available to us.

For our purposes, a *robot* will be modeled by two main elements. The first element is the robot's geometric model, say with respect to its reference pose (e.g., positioned at the origin). The second is its *configuration*, by which we mean a finite sequence of numeric values that fully specifies the position of the robot. Combined, these two elements fully define the robot's exact shape and position of the robot in space.

For example, suppose that the robot is a 2-dimensional polygon that can translate and rotate in the plane. Its geometric representation might be given as a sequence of vertices, relative to its reference position. Let us assume that this reference pose overlaps the origin. We refer to the location of the origin as the robot's *reference point*. The robot's configuration may be described by its translation, which we can take to be the (x, y) coordinates of its reference point after translation and an angle θ that represents the counterclockwise angle of rotation about its reference point (see Fig. 1(a)). Thus, the configuration is given by a triple (x, y, θ). We define the space of all valid configurations to be the robot's *configuration space*. For any point $p = (x, y, \theta)$ in this space, we define $\mathcal{R}(p)$ to be the corresponding *placement* of the robot in the workspace.

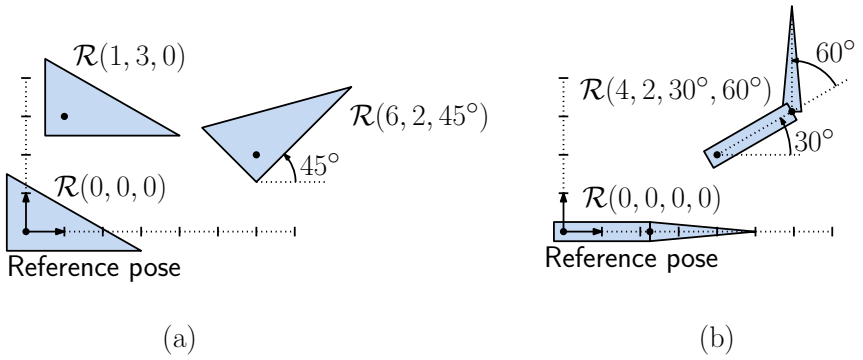

Fig. 1: Configurations of: (a) translating and rotating robot and (b) a translating and rotating robot with a revolute joints.

A more complex example would be an *articulated arm* consisting of a set of links, connected to one another by a set of *revolute joints*. The configuration of such a robot would consist of a vector of joint angles (see Fig. 1(b)). The geometric description would probably consist of a geometric representation of the links. Given a sequence of joint angles, the exact shape of the robot could be derived by combining this configuration information with its geometric description.

Free Space: Because of limitations on the robot's physical structure and the obstacles, not every point in configuration space corresponds to a legal placement of the robot. Some configurations may be illegal because:

- the joint angle is outside the joint's operating range (e.g., you can bend your knee backwards,

but not forwards . . . ouch!)

- the placement associated with this configuration intersects some obstacle

Such a configuration is called a *forbidden configuration*. The set of all forbidden configurations is denoted $C_{\text{forb}}(\mathcal{R}, S)$, and all other placements are called *free configurations*, and the set of these configurations is denoted $C_{\text{free}}(\mathcal{R}, S)$, or *free space*. These two sets partition configuration space into two distinct regions. (Note that the regions do not need to be connected. That is, there may be two free configurations, but there is no path between them that lies entirely within the free space.)

C-Obstacles and Paths in Configuration Space: *Motion planning* is the following problem: Given a workspace S, a robot $\mathcal{R}$, an initial configuration s, and a final configuration t (both points in the robot's free configuration space), determine whether it is possible to move the robot from one configuration by a path $\mathcal{R}(s) \rightarrow \mathcal{R}(t)$consisting entirely of free configurations (see Fig. 2(a)).

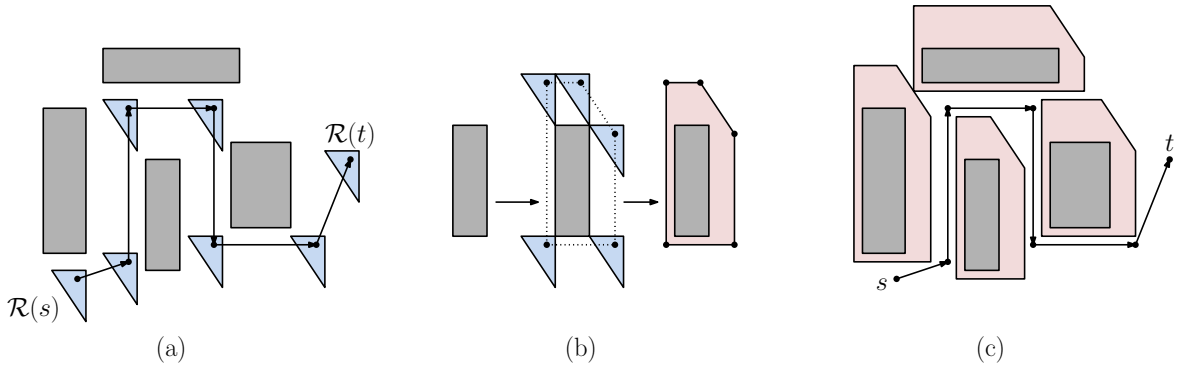

Fig. 2: Motion planning: (a) workspace with obstacles, (b) converting obstacles into C-obstacles, and (c) configuration space and C-obstacles.

Based on the definition of configuration space, it is easy to see that the motion planning problem reduces to the problem of determining whether there is a path from s to t in configuration space (as opposed to the robot's workspace) that lies entirely within the robot's free configuration subspace. Thus, we have reduced the task of planning the motion of a robot in its workspace to the problem of finding a path for a single point through free configuration space.

Since these are rather hard to illustrate, let's consider instead perhaps the simplest nontrivial case, which is translating a convex polygonal robot in the plane amidst a collection of polygonal obstacles. In this cased both the workspace and configuration space are two dimensional. We claim that, for each obstacle in the workspace, there is a corresponding *configuration obstacle* (or *C-obstacle*) that corresponds to it in the sense that if $\mathcal{R}(p)$ does not intersect the obstacle in the workspace, then p does not intersect the corresponding C-obstacle.

In order to see how to define the C-obstacles in this simple example, consider the path traced out by the robot's reference point as we "scrape" it along the boundary of the obstacle (see Fig. 2(b)). The associated C-obstacle is simply the collection of points traced out by this path (or generally a surface in higher dimensions) (see Fig. 2(b)). Given the collection of C-obstacles in configuration space, the motion planning problem reduces to determining whether there exists a *path* from s to t that avoids all the C-obstacles. It is easy to prove that there exists a valid motion in the workspace if and only if there exists a path in the free space.

When rotation is involved, this scraping process must consider not only translation, but all rotations that cause the robot's boundary to touch the obstacle's boundary. (One way to visualize this is to fix the value of θ, rotate the robot by this angle, and then compute the translational C-obstacle with the robot rotated at this angle. Then, stack the resulting C-obstacles on top of one another, as θ varies through one complete revolution. The resulting "twisted column" is the C-obstacle in 3-dimensional space.) Note that because the configuration space encodes not only translation, but the joint angles as well. Thus, a path in configuration space generally characterizes both the translation and the individual joint rotations. (This is insanely hard to illustrate, so I hope you can visualize this on your own!)

When dealing with polyhedral robots and polyhedral obstacles models under translation, the C-obstacles are all polyhedra as well. However, when revolute joints are involved, the boundaries of the C-obstacles are curved surfaces, which require more effort to process than simply polyhedral models. Complex configuration spaces are not typically used in games, due to the complexity of processing them. Game designers often resort to more ad hoc tricks to avoid this complexity, and the expense of accuracy.

Discretizing Configuration Space: As mentioned above, we can reduce the motion planning problem (for a single moving agent) to the problem of computing a path in the free configuration space. Unfortunately, computing such a path for even a single point in space is a nontrivial problem. Typically, we are interested not merely in the existence of a path, but rather finding a "good" path, where goodness connotes different things in different contexts (e.g., shortness, low-energy, natural looking, etc.)

There are two common techniques for computing such paths. The first is based on a sort of physical simulation of *repulsive potential fields* and the second is based on *graph search algorithms*. Let's consider each of these.

Potential-Field Navigation: The analogy to understand this approach is to imagine a smoothly varying terrain with hills and valleys. Suppose you place a marble on top of one of the hills of the terrain and let it go. It will naturally slide down until reaching the lowest point.

How can we base a path finding algorithm on this idea? Suppose that your obstacles reside in 2-dimensional space (see Fig. 3(a)). The idea is to model this as a terrain in 3-dimensional space. The start point s will lie on top of a hill, the goal point t as lying in the bottom of a deep basin, and all the obstacles are modeled as vertical cylinders whose 2-dimensional projections are the obstacles, but which have steep vertical sides (see Fig. 3(b)). Think of the obstacles like the plateaus you find out in the American deserts, where the cross section of the plateau is the obstacle.

Now, when you start the marble rolling at point s, the force of gravity will naturally draw it down towards the destination t. Because the obstacles form high vertical "plateaus", the marble naturally roll around and avoid them. With a bit of luck, the marble will eventually roll around on this terrain and find its way from s to t. By projecting the resulting path down into the plane, we obtain the desired path (see Fig. 3(c)).

More formally, the process is modeled as follows. First, we set up a repulsive field around obstacles (and generally any forbidden regions of the configuration space). Next, we set up an attractive field centered at the desired destination point t. Finally, with the aid of a physics simulator, we let our robotic marble flow "downhill" along the line of steepest descent in the resulting potential field.

How do we do this? If we let $\Psi(x, y)$ denote the value of the potential field at any point direction (x, y), then the direction of steepest descent it given by the *gradient vector*, which can be computed

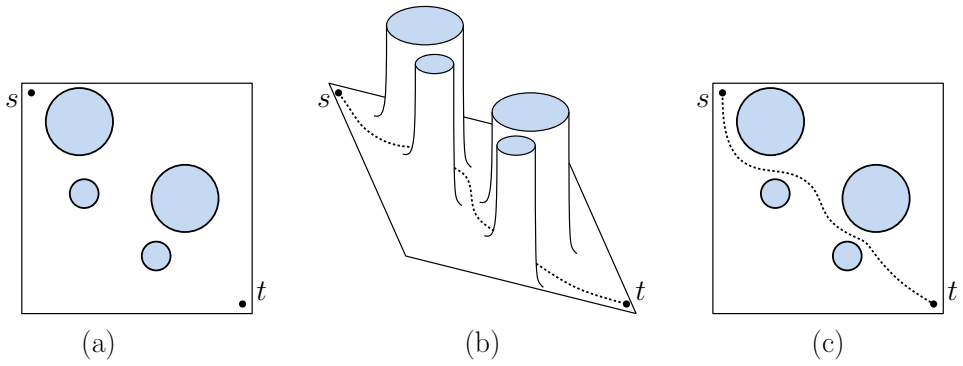

Fig. 3: Computing paths by potential-field navigation.

from the partial derivatives of Ψ. More formally, the gradient is $\nabla\Psi = (\partial\Psi/\partial x, \partial\Psi/\partial y)$. Thus, if you define your potential fields to be differentiable functions, then the gradient is easy to compute at any point. Of course, the gradient changes from one point to the next. The simulator follows the gradient direction downhill for a short distance. It then recomputes the gradient at the new point, and keep continuing.

One of the features of potential-field navigation is that, if the physics is properly modeled, the movement point naturally follows a smooth energy-minimizing path. Thus, it results in smooth, natural motion.

While potential-field navigation has a certain intuitive charm to it, it has a significant downside, which is shared by all gradient-descent methods. Namely, it can get stuck in local minima of the potential field. Once your robotic marble roles into a depression surrounding by mountainous obstacles on all sides, it will come to rest there, without reaching the destination. This suggests the need for more *global* solutions, which will discuss next.

Graph-Based Navigation: The other widely used class of methods are based on first discretizing the configuration free space into a graph model (with nodes and edges), and then applying any standard shortest path algorithm, such as Dijkstra's algorithm or A* search to compute the shortest path. (We'll say more about these algorithms later.) There are many different methods that are used for extracting this graph. We will consider a few common ones for the remainder of this lecture.

Waypoints and Road Maps: The simplest approach for generating a navigation graph, is to scatter a large number of points throughout the configuration free space, called *waypoints*, and then connect nearby waypoints to one another. The edges of this graph can be labeled with the distance between the associated points. The resulting graph is called a *road map*. (Note that these are distances in configuration space, so this is not simply Euclidean distance. For example, you may need to define what is meant by the distance of rotating a joint through an angle of θ.) Given the location of the start point s and the destination point t, we join these with edges to nearby waypoints. Finally, we invoke a shortest path algorithm. The result is a path in configuration space.

How do we construct these waypoints? There is no single preferred method. Here are a few ideas:

Placed by the game designer: The game designer has a notion of where it is natural for the game agents to move, and so places waypoints along routes that he/she deems to be important. For ex-

ample, this would include points near the entrances and exits of rooms in an indoor environment or along the streets or crosswalks in an urban setting. This gives the designer a lot of control of the motion of the game agents, and a lot of flexibility to add more points where motion is highly constrained and fewer where motion is unconstrained.

In a large environment, however, there may be too many waypoints for a single designer to place. Thus, we would like something more automated.

Grid: The simplest way to cover a large area is to overlay a square grid, and generate waypoints along the vertices of the grid (see Fig. 4(b)). This has the advantage of simplicity, but note that it can result in the generation of a very large number of grid points, if the grid resolution is not properly set.

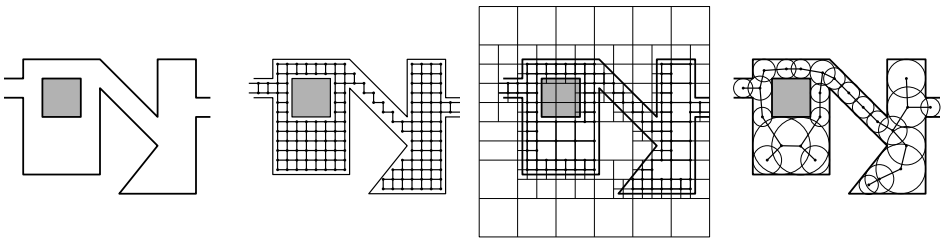

Fig. 4: A road map for a set of obstacles (a) based on waypoints generated on: (b) a grid, (c) a quadtree, (d) the medial axis.

Quadtree: Placing the waypoints at the vertices of a quadtree decomposition has the advantage that the resolution of the waypoints can be adjusted to fit the tightness of the environment (see Fig. 4(c)). Narrow cramped corridors will be decomposed into small regions, and wide open fields into large regions.

Medial-Axis Waypoints: Both grids and quadtrees suffer from the same problems. The first is that motion along the vertices of either structure tends to produce paths that are aligned with the coordinate axes. Thus, motion along a diagonal corridor will tend to zig-zag to the left and right. Additional steps are needed to smooth out such paths.

One idea for fixing this is to situate the waypoints so they correspond to the centers of obstacle-free disks of maximum size. We say that a circular disk D is *maximal* if there is no obstacle-free disk of larger radius that contains D. The union of the centers of all maximal disks naturally defines a set of points that runs along the center of your domain. It is an important object in geometry, called the *medial axis*.

By sampling waypoints on or near the medial axis (see Fig. 4(d)), agents will naturally move along the centers of corridors. (Of course, you can add a bit of random variation, so they merely walk in a more natural manner that is not too close to the walls.) This method of placing waypoints is best for 2-dimensional domains (since it is messier to compute the medial axis of higher dimensional configuration spaces) and where the domain consists mostly of corridor-like structures.

Navigation Meshes: One of the shortcomings of all the aforementioned approaches is that they neglect the essential feature that the free configuration space is a *region*, not a collection of discrete points. If you are a dynamic setting, and an obstacle is placed on a waypoint in the center of a

narrow corridor, it may be impossible to determine how to navigate around the obstacle, since there are no alternate waypoints nearby.

The next idea is based on producing a decomposition of the free configuration space into a collection of convex regions. Such a decomposition is called a *navigation mesh*. There are many ways to build road maps based on such meshes. A very popular way is to shoot a bullet path both up and down from each obstacle vertex. These bullet paths subdivide space into a collection of trapezoid shaped regions (with parallel vertical sides). The decomposition is called a *trapezoidal map*, and it can be computed very efficiently (see Fig. 5(b)).

To generate a road map from a trapezoidal map we create a vertex in the center of each trapezoid and a vertex at the midpoint of each vertical edge. We create edges joining each center vertex to the vertices on its (at most four) edges (see Fig. 5(c)).

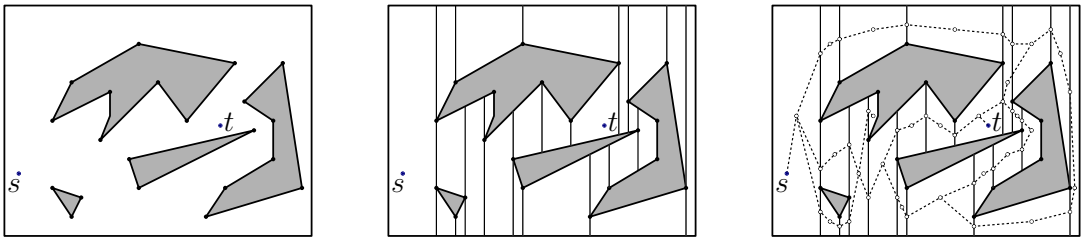

Fig. 5: Trapezoidal-map based navigation mesh: (a) the obstacles, (b) trapezoidal map, (c) road map with connecting links to s and t.

Now to answer the motion planning problem, we assume we are given the start point s and destination point t. We locate the trapezoids containing these two points, and connect them to the corresponding center vertices. We can join them by a straight line segment, because the cells of the subdivision are convex.