cisc3665
game design
fall 2011
lecture # III.3
path planning

topics:

- pathfinding
- waypoints
- A*

references:

- notes on pathfinding and waypoints from: *AI for Game Developers*, by David M. Bourg and Glenn Seemann. O'Reilly Media (2004), chapters 6 and 7.

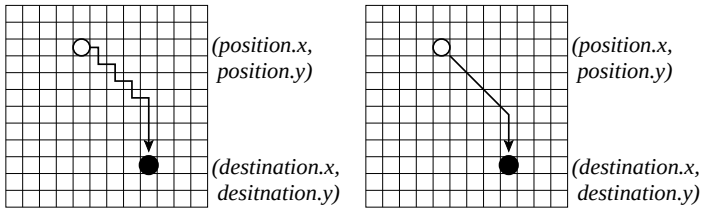
basic pathfinding

- pathfinding is simply defined as figuring out how to move an agent from one location to another
- the simplest algorithm is similar to the "chasing" algorithm from chapter 2 of Bourg and Seemann:

```
PVector position, destination;
// update x position
if ( position.x > destination.x )
  position.x--;
else if ( position.x < destination.x )
  position.x++;
// update y position
if ( position.y > destination.y )
  position.y--;
else if ( position.y < destination.y )
  position.y++;
```

- this works fine if we don't have to worry about the edges of the display window

- the shape of the agent's path, or *trajectory*, could be either:

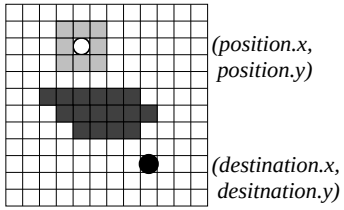


(a) (b)

- the trajectory on the left (a) results if the agent cannot move diagonally, so two drawing updates occur, one after the "update x position" clause and one after the "update y position" clauses in the algorithm on the previous page
- the trajectory on the right (b) results if the agent can move diagonally, so one drawing update occurs after both the "update x position" and "update y position" clauses in the algorithm on the previous page
- it is also possible to use the Bresenham algorithm (or some other scan-conversion algorithm) to produce a smoother line (see Bourg and Seemann ch 2 for details)

obstacle avoidance

- The simple pathfinding algorithm is insufficient if there are obstacles in between the agent's position and its destination. So then you have to apply an obstacle avoidance technique.
- Let's assume that our agent does not know about any obstacles in its path until it is right next to one. The figure below shows the agent's current position with its adjacent cells colored light grey, to indicate that these are the cells that the agent can sense with its perception capabilities. There is an obstacle between the agent's current position and its destination, but the agent does not know about it because the obstacle is beyond the range of the agent's sensors.



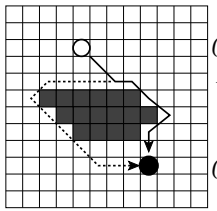
- The simplest obstacle avoidance algorithm would be something like this:

```
// determine desired location to try
if ( position.x > destination.x )
  try.x = position.x - 1;
else if ( position.x < destination.x )
  try.x = position.x + 1;
if ( position.y > destination.y )
  try.y = position.y - 1;
else if ( position.y < destination.y )
  try.y = position.y + 1;
if the cell at (try.x,try.y) does not have an obstacle in it,
  then move to (try.x,try.y)
  else try the other cells adjacent to (position.x,position.y)
       until an empty cell is found
```

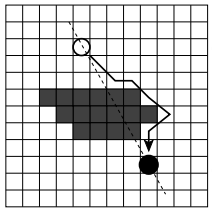
- The "try other cells" portion of the code could either randomly pick adjacent cells until an empty one is found, or could look more methodically, for example moving clockwise (or counterclockwise) around the adjacent cells until an empty one is found.
- The more methodical approach is generally better and more efficient.

obstacle tracing

- If your agent has found an obstacle in its path, it needs to find some way of going around it.
- The figure below shows two alternatives for *obstacle tracing*—moving around the edge of an obstacle. At first, the agent moves towards its destination using the simple pathfinding algorithm. But when it encounters an obstacle and can no longer follow that algorithm, the agent then invokes an obstacle-tracing algorithm. The agent chooses between initially moving left or right. This can an arbitrary decision, since the agent does not know the extent of the obstacle until it has completed tracing its perimeter.



- One method for determining how the agent should move when it encounters an obstacle is as follows. Project an imaginary line-of-sight from the agent's position to its destination. Then when the obstacle is encountered, select the initial move of the obstacle-tracing algorithm by moving in the direction that brings the agent closest to the line-of-sight. This is shown in the figure below:



- This technique is also handy for knowing when to stop executing the obstacle-tracing algorithm and return to executing the simple pathfinding algorithm. When the agent moves around the obstacle, it should choose its moves to stick as close to the line-of-sight as possible (given the constraints of the obstacle). As soon as the agent can move back onto the line-of-site, unobstructed, then the agent should switch back to the simple-pathfinding algorithm.

using the line-of-sight

- Here's how to use the line-of-sight. First, assume that you know the location of the agent's current position $(pos.x, pos.y)$ and its destination $(dest.x, dest.y)$.
- Then you can use the point-slope formula to determine if any point (x, y) is on that line.

$$y - pos.y = m(x - pos.x)$$

where m is the slope of the line between the agent's current position and its destination:

$$y - pos.y = \frac{(dest.y - pos.y)}{(dest.x - pos.x)}(x - pos.x)$$

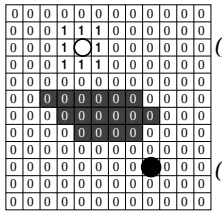
- Since you know $(pos.x, pos.y)$ and $(dest.x, dest.y)$, you can easily determine if any given (x, y) works in the above equation by doing the math: plug in the values for each variable and determine if the left and right sides of the equation are equal (or not). If they are not, then (x, y) is not on the agent's line-of-sight path.

breadcrumbs

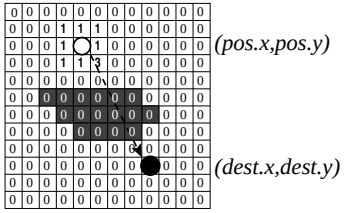
- Some games have players leave *breadcrumbs* in the environment, like a "trail" showing where the agent has been.
- This is handy in environments where the obstacles are static (or slow-moving).
- Every time an agent successfully moves through a cell, it leaves a "breadcrumb". Other agents can later sense these breadcrumbs and "pick up the scent" of a viable trail—by following the breadcrumbs from one cell to another.
- Agents do this by sensing breadcrumbs in adjacent cells and using these to prioritize which cell to visit next.
- The lead (first) agent still has to perform pathfinding (as on earlier slides), but subsequent agents that pick up the trail can avoid this computation and follow the old path.
- Of course, this only works when the subsequent agents have a destination that is the same as (or along the way to) the lead agent's destination.
- In dynamic environments, the breadcrumbs can "decay" over time, fading from one timestep in the game to another, until they can no longer be sensed if too much time has passed since they were "dropped".

path following using terrain analysis

- One method of path following is for the agent to perform *terrain analysis*.
- With this method, the agent maintains a map of the space in its memory. This map is a 2-dimensional grid, where each entry in the grid corresponds to a cell in the agent's world that the agent can move to. Initially, all the entries in the grid are set to 0 ("unsensed").
- As the agent senses its environment and moves around, it fills in the grid with information about the cells it senses. The agent might store 1 in empty cells and -1 in cells that contain obstacles. In the figure below, the agent starts in the location shown and senses its immediate neighbors.



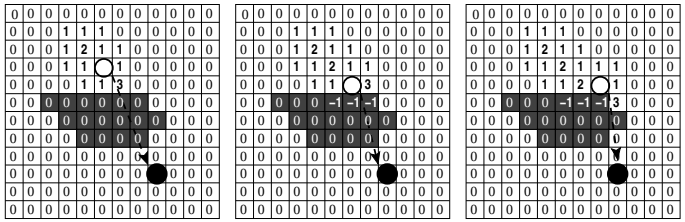
- The agent begins pathfinding by looking at its 2D grid for entries corresponding to its 8 adjacent cells. If the agent reads a 0 from its 2D grid, then it needs to sense the corresponding cell in its environment and update the 2D grid accordingly. When all cells surrounding the current cell have been sensed, then the agent can choose its move—by selecting one of the empty cells (marked by 1).
- An intelligent agent will pick the empty cell that is closest to its line-of-sight, from the current position to the destination. In fact, the agent can "mark" the 2D grid by determining which cells are closest to the line-of-sight and storing higher values in the grid—e.g., putting 3 in cells that are along the line-of-sight (but are not the destination).



Note that the agent does not mark all the cells along its line-of-sight—only the ones that it has sensed and knows that there are no obstacles there.

- Then the agent's choice of move is merely to find the largest value amongst its 8 neighbor cells and move there.

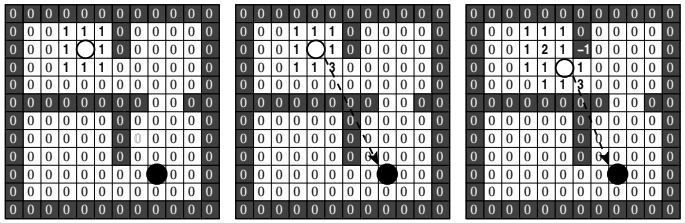
- As the agent moves, it senses its neighboring cells and updates its 2D grid.



- Note that the agents stores a value of 2 in the cells it has already visitied, marking its trail.
- Also notice that "old" values remain in the 2D grid, meaning that the grid acts like the agent's memory of where it has explored in its environment.

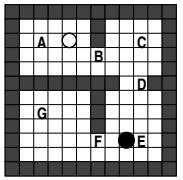
wall tracing

- Another application of this technique is for a task called *wall tracing*.
- Here, instead of using the terrain analysis technique for going around obstacles, the agent uses it for going around the edges of walls. This is a handy method for an agent to find its way out of a room.



waypoint navigation

- One method for navigation environment is to define special locations in the environment called *waypoints* (the book calls them *nodes*). An agent will find a path that goes from one waypoint to another. The waypoints should be defined so that an agent always has a line-of-sight from one to another.
- The figure below shows seven waypoints (marked with letters **A** through **G**), one inside each doorway and one inside each room in the environment.



- Waypoints should be defined so that each waypoint can be seen from at least one other waypoint.

- The agent can then construct an *adjacency matrix* indicating which waypoints are adjacent to each other, which dictates which waytpoint should be visited first when starting from any other waypoint.
- The table below contains the adjacency matrix for the map on the previous slide. The table should be read as follows. Each row is labeled with a starting waypoint, and each column is labeled with an ending waypoint. For example, when starting at waypoint **A** and going to waypoint **E**, the first waypoint that should be visited is **B**.

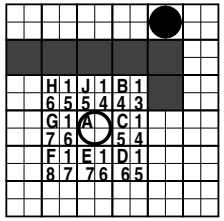
	A	B	C	D	E	F	G
A	-	B	B	B	B	B	B
B	A	-	C	D	D	D	D
C	B	B	-	D	D	D	D
D	B	B	C	-	E	F	F
E	D	D	D	D	-	F	F
F	E	E	E	E	E	-	G
G	F	F	F	F	F	F	-

A*

- A* (pronounced "A-star") is one of the most popular methods for pathfinding in games (and in robotics!)
- A* works well with not too, too large a space to traverse.
- Using waypoints helps make a large space tractable.
- Then you can use A* to calculate a path from one waypoint to another.
- The next slide contains the pseudo code for the A* algorithm.
- The "open" list contains the waypoints that need to be searched.
- The "closed" list contains the waypoints that have already been searched.
- The "cost" of going from "this" waypoint to the current waypoint is calculated by adding the cost of getting to the current waypoint (g) to the estimated cost of going to the next waytpoint (h). The estimated cost is determined using a *heuristic* — an educated guess.
- This is the essence of A*: $cost = g + h$

```
add starting waypoint to "open" list
while ( open list is not empty ) {
  current waypoint = lowest cost waypoint from open list
  if ( current waypoint == goal )
    done!
  else {
    remove current waypoint from open list
    add current waypoint to closed list
    for each waypoint that is adjacent to the current waypoint {
      if ( this waypoint is not on the open list ) and
         ( this waypoint is not on the closed list ) and
         ( there is no obstacle between current waypoint and
           this waypoint ) {
        add this waypoint to the open list
        calculate cost of going to this waypoint from current waypoint
      }
    }
  }
}
```


- Here is an example:



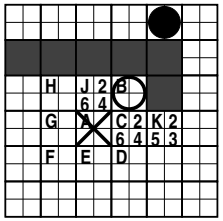
We start at the waypoint marked **A**. The destination is the filled black circle near the upper right corner. Each of the waypoints adjacent to **A** is evaluated.

- To illustrate the computation, each waypoint's upper right corner contains the cost to get from the current waypoint to the waypoint being evaluated. This is g. This is 1 for the waypoints that could be visited on the first step, i.e., one step from A.
- Each waypoint's lower right corner contains the estimated cost to get from the waypoint being evaluated to the destination. This is h.
- Each waypoint's lower left corner contains the total cost, $g + h$.

- So, the first pass through the A* algorithm on this example goes like this:

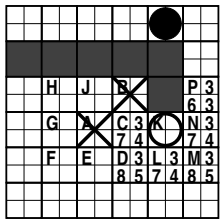
```
add A to the open list
set the current waypoint to A (only waypoint on open list)
A is not the goal so:
  remove A from the open list ()
  add A to the closed list (A)
  add B, C, D, E, F, G, H, and J to the open list
```

- The second pass goes like this:



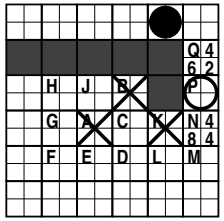

```
set the current waypoint to B
     (lowest cost waypoint on
      open list)
B is not the goal so:
  remove B from the open list
  add B to the closed list (A,B)
  add K to the open list
     (C,D,E,F,G,H,J,K)
```


- The third pass goes like this:



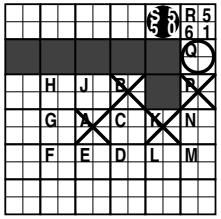

```
set the current waypoint to K
     (lowest cost waypoint on
      open list)
K is not the goal so:
  remove K from the open list
  add K to the closed list (A,B,K)
  add L, M, N and P to the open list
     (C,D,E,F,G,H,J,L,M,N,P)
```

- The fourth pass goes like this:



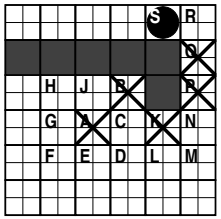

```
set the current waypoint to P
     (lowest cost waypoint on
      open list)
P is not the goal so:
  remove P from the open list
  add P to the closed list (A,B,K,P)
  add Q to the open list
     (C,D,E,F,G,H,J,L,M,N,Q)
```

- The fifth pass goes like this:




```
set the current waypoint to Q
     (lowest cost waypoint on
      open list)
Q is not the goal so:
  remove Q from the open list
  add Q to the closed list (A,B,K,P,Q)
  add R and S to the open list
     (C,D,E,F,G,H,J,L,M,N,R,S)
```

- The sixth pass goes like this:




```
set the current waypoint to S
     (lowest cost waypoint on
      open list)
S is the goal so we're done!
```


to do

- work on homework assignment for unit III, which is due November 7