cisc3665
game design
fall 2011
lecture # III.2
perception

topics:

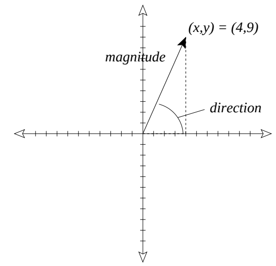
- game physics
- perception
- chasing and evading

references:

- notes on physics from: *Programming Game AI by Example*, by Mat Buckland. Worldware Publishing (2005), chapter 1.
- notes on chasing and evading from: *AI for Game Developers*, by David M. Bourg and Glenn Seemann. O'Reilly Media (2004), chapter 2.

vectors

- a *vector* represents two quantities: a *magnitude* and a *direction*
- a vector is expressed as: $\vec{v} = (x, y)$



  - to determine the magnitude and direction of a vector, imagine a right triangle where the vector is the hypotenuse
  - the length of one side is the x-value and the length of the other side is the y-value
  - so you can use the Pythagorean Theorem to determine the *magnitude*, $|\vec{v}|$, of the vector:

$$h^2 = x^2 + y^2$$
$$h = \sqrt{(x^2 + y^2)}$$

- and some trigonometry to determine the *direction* (θ) of the vector:

$$sin(\theta) = opposite/hypotenuse$$
$$sin(\theta) = y/|\vec{v}|$$
$$\theta = asin(y/|\vec{v}|)$$

normalized or unit vector

- it is common to *normalize* a vector, or find the *unit* vector
- the unit vector is a vector whose magnitude is equal to 1
- a unit vector is primarily used for its directionality
- so if we go back to our example on the previous page, $\vec{v} = (4, 9)$, we can compute the magnitude as:

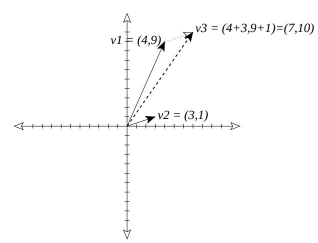
$$|\vec{v}| = \sqrt{(4^2 + 9^2)} = \sqrt{(16 + 81)} = \sqrt{97} = 9.85$$

- the unit vector is computed by dividing the x and y components of the vector by its magnitude:

$$(x, y) = (4, 9) \Rightarrow (4/9.85, 9/9.85) = (0.406, 0.914)$$

using vectors

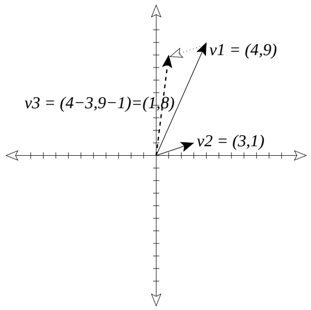
- often, you want to perform operations on vectors, such as adding them together or subtracting them from each other or finding the angle between them
- each of these operations is performed by applying the operation to the x and y components individually
- **adding two vectors:**
  what happens when you add two vectors together?



$$\begin{aligned} \vec{v1} &= (x1, y1) &&= (4, 9) \\ \vec{v2} &= (x2, y2) &&= (3, 1) \\ \vec{v3} &= \vec{v1} + \vec{v2} && \\ &= (x1 + x2, y1 + y2) &&= (4 + 3, 9 + 1) \\ & &&= (7, 10) \end{aligned}$$

- notice that this is like starting $\vec{v2}$ at the endpoint of $\vec{v1}$, to arrive at $\vec{v3}$

- **subtracting two vectors:**
  what happens when you add subtract two vectors from each other?



$$\begin{aligned} \vec{v1} &= (x1, y1) && = (4, 9) \\ \vec{v2} &= (x2, y2) && = (3, 1) \\ \vec{v3} &= \vec{v1} - \vec{v2} \\ &= (x1 - x2, y1 - y2) && = (4 - 3, 9 - 1) \\ & && = (1, 8) \end{aligned}$$

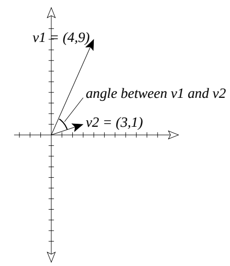
- notice that this is also like starting $\vec{v2}$ at the endpoint of $\vec{v1}$, but since we are using subtraction, we *reverse* the direction of $\vec{v2}$ to arrive at $\vec{v3}$

- **finding the angle between two vectors:**
  how do you find the angle between two vectors?
  you use something called the *dot product*, which is computed as:

$$\vec{v1} \cdot \vec{v2} \;=\; x1 \times x2 + y1 \times y2 \;=\; |\vec{v1}||\vec{v2}|cos(\theta)$$

where θ is the angle between $\vec{v1}$ and $\vec{v2}$, which means that we want to solve for θ.
so we can rearrange the above as:

$$\begin{aligned} |\vec{v1}||\vec{v2}|cos(\theta) &= x1 \times x2 + y1 \times y2 \\ cos(\theta) &= (x1 \times x2 + y1 \times y2)/|\vec{v1}||\vec{v2}| \\ \theta &= acos((x1 \times x2 + y1 \times y2)/|\vec{v1}||\vec{v2}|) \end{aligned}$$



$$\begin{aligned} |\vec{v1}| &= \sqrt{(x1^2 + y1^2)} && = \sqrt{(4^2 + 9^2)} \\ &= \sqrt{(16 + 81)} && = \sqrt{97} = 9.85 \\ |\vec{v2}| &= \sqrt{(x2^2 + y2^2)} && = \sqrt{(3^2 + 1^2)} \\ &= \sqrt{(9 + 1)} && = \sqrt{10} = 3.16 \\ \theta &= acos((x1 \times x2 + y1 \times y2) && / \; |\vec{v1}||\vec{v2}|) \\ &= acos((4 \times 3 + 9 \times 1) && / \; 9.85 \times 3.16) \\ &= acos((12 + 9)/31.126) && = acos(21/31.126) \\ &= acos(0.67) = 47.57^\circ \end{aligned}$$

- **scaling a vector:**
  sometimes you want to scale the magnitude of a vector. use *multiplication* to proportionally increase the magnitude of the vector, and *division* to proportionally decrease the magnitude of the vector.

- for example, if you want to have a vector, $\vec{v4}$ that is half the magnitude of our vector $\vec{v1} = (4, 9)$, there are two ways to compute $\vec{v4}$
- one way is to divide each component in $\vec{v1}$ by 2:

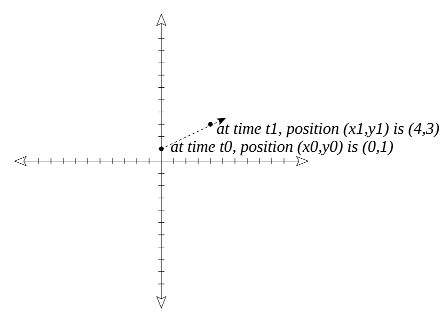
$$\vec{v4} = (x1/2, y1/2) = (4/2, 9/2) = (2, 4.5)$$

- another way is to find the unit vector for $\vec{v1}$ (as we did previously): $(0.406, 0.914)$, and then multiply that by half the magnitude of the $\vec{v1}$, i.e., $|\vec{v1}|/2 = 9.85/2 = 4.925$ (using our earlier calculations for $|\vec{v1}|$:

$$\vec{v4} = (0.406 \times 4.925, 0.914 \times 4.925) = (2, 4.5)$$

motion

- in games, you typically want to make something move
- so we need to know about motion
- motion involves *velocity* (i.e., speed), which is defined as "change in position over time"

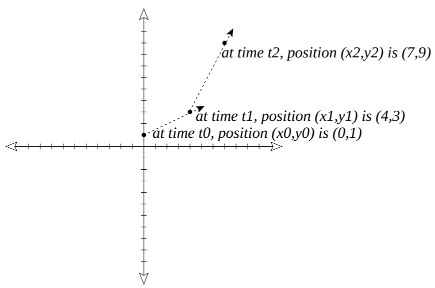


velocity
is a vector, $\vec{v}$, representing the change in x and the change in y over the same time period

in our example:

$$\begin{aligned} (x0, y0) &= (0, 1) \\ (x1, y1) &= (4, 3) \\ \vec{v}_0 &= (x1 - x0, y1 - y0) \\ &= (4 - 0, 3 - 1) \\ &= (4, 2) \end{aligned}$$

- motion also involves *acceleration*, which is defined as "change in velocity over time"



acceleration is also a vector, $\vec{a}$, representing the change in the x and y components of velocity over time

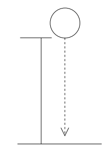
in our example:

$$\begin{aligned}(x0, y0) &= (0,1)\\ (x1, y1) &= (4,3)\\ \vec{v}_0 &= (4,2)\\ (x2, y2) &= (7,9)\\ \vec{v}_1 &= (x2-x1, y2-y1)\\ &= (7-4, 9-3)\\ &= (3,6)\\ \vec{a} &= \vec{v}_1 - \vec{v}_0\\ &= (3-4, 6-2)\\ &= (-1,4)\end{aligned}$$

- if an object in motion has acceleration $= 0$, then it has *constant velocity*

falling objects

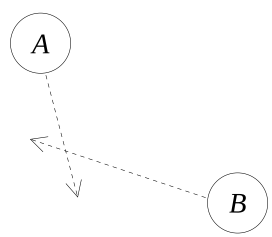
- what happens when an object falls?
- we'll model free-falling objects—objects that do not have a force applied to them—except the force of *gravity*
- the value of *acceleration due to gravity* is 9.8 meters per second squared (m/s/s)



time	*velocity*	*distance travelled* (Δy)
$t0 = 0$	0.0 m/s	0.0 m
$t1 = 1$ sec	9.8 m/s	9.8 m
$t2 = 2$ sec	19.6 m/s	29.4 m
$t3 = 3$ sec	29.4 m/s	58.8 m

colliding objects

- what happens when two objects collide with each other?



- force
- *momentum*: $p = mv$ (mass times velocity)
- change in momentum over time that a force is acting on an object: $dp = F \times dt$
- *impulse* is the measure of the strength and the duration of the force of the collision
- if two objects collide (and there are no external forces acting in the system), then *momentum is conserved*

$p_{total} = p_1 + p_2$

- so change in momentum $= 0$

$\Delta p_1 + \Delta p_2 = 0$

- *elastic* collision $=$ when kinetic energy is conserved (otherwise, collision is *inelastic*)
- Let's assume that the mass of the objects doesn't change when two objects collide. So we have, by conservation of momentum:

$\Delta p_1 = \Delta v_1 \times m_1$

$\Delta p_2 = \Delta v_2 \times m_2$

$\Delta p_1 + \Delta p_2 = 0$

$\Delta v_1 \times m_1 + \Delta v_2 \times m_2 = 0$

$\Delta v_1 = v_{1f} - v_{1i}$

$\Delta v_2 = v_{2f} - v_{2i}$

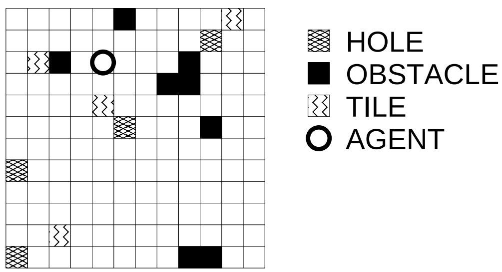
$(v_{1f} - v_{1i}) \times m_1 + (v_{2f} - v_{2i}) \times m_2 = 0$

$v_{1f} m_1 - v_{1i} m_1 + v_{2f} m_2 - v_{2i} m_2 = 0$

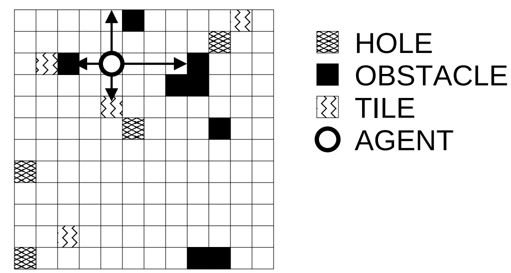
$v_{1f} m_1 + v_{2f} m_2 = v_{1i} m_1 + v_{2i} m_2$

perception

- there are generally two types of perception: *local* and *global*
- with *global* perception, the agent(s) in your game can "sense" everything that is in the environment; i.e., all environmental properties are *accessible*
- so in the example below, the agent knows where all the holes, obstacles and tiles are in its environment:



- with *local* perception, the agent(s) in your game can only "sense" properties using *sensors* that are (virtually) part of the agent
- so in the example below, the agent can only knows about the closest element(s) in each of the four compass directions (assuming that this agent has four sensors, and each sensor is pointed in one compass direction)
- obviously in this case, there are properties that the agent does not see; these properties are *inaccessible*



- the chasing and evading agents in ch 2 of Bourg & Seeman have global perception
- whereas, the TileWorld agent in your unit II assignment has local perception
- in that assignment and in our example on the previous page, the agent has 4 *sensors* that look for non-empty cells in each of the 4 compass directions
- it is common to write a `sense()` function that simulates the sensors
- in our case, i.e., in `tileworld.pde`, we simply find the agent's location in the occupancy grid, and then look at cells above (north), below (south), to the left (west) and right (east) of the agent in order to determine if anything is there
- we can set a *range* for each sensor—the maximum number of cells to look at in any direction, counted from the agent's location
- if we don't set a range, then we assume that the agent can see to the edge of the world in each direction—unless it finds a non-empty cell; in which case, the sensor returns a value indicating what it detected in the non-empty cell (e.g., a tile, a hole or an obstacle)

chasing and evading

- exemplified as two-agent environment, where one agent is the **predator** (and does the *chasing*) and the other agent is the **prey** (and does the *evading*)
- basic chasing algorithm— predator updates its position by moving closer to its prey

```
predator.sense(); // get (x,y) location of prey
if ( predator.x > prey.x ) predator.x--;
else if ( predator.x < prey.x ) predator.x++;
if ( predator.y > prey.y ) predator.y--;
else if ( predator.y < prey.y ) predator.y++;
if (( predator.x == prey.x ) && ( predator.y == prey.y )) {
  GOTCHA!!!
}
```

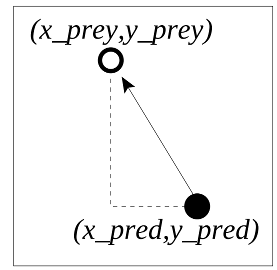
- basic evading algorithm— prey updates its position by moving farther from its prey opposite of the predator algorithm!

```
prey.sense(); // get (x,y) location of predator
if ( prey.x > predator.x ) prey.x++;
else if ( prey.x < predator.x ) prey.x--;
if ( prey.y > predator.y ) prey.y++;
else if ( prey.y < predator.y ) prey.y--;
```

- but this is ineffective if predator and prey are both using these algorithms and their velocities are the same.
- so we should try something smarter...

line-of-site chasing

- compute straight-line path from predator to prey
- in a continuous environment (i.e., where agents locations can be real-valued coordinates), then you can just use math to calculate a vector from predator to prey



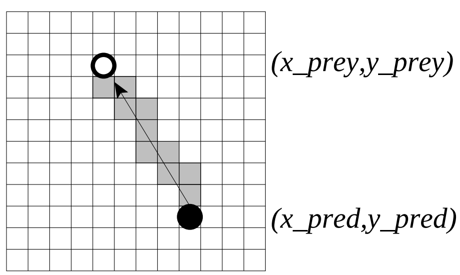
distance to travel from predator to prey = *magnitude* of vector:

$= \sqrt{(pred.x - prey.x)^2 + (pred.y - prey.y)^2}$

angle to travel from predator to prey = *direction* of vector:

$= sin^{-1}(|pred.y - prey.y|)/magnitude$

- in a discrete (e.g., tiled) environment, you need to use a *scan conversion* algorithm to approximate the straight line, but always moving in one of the 4 compass directions



to do

- read ch 2 of Bourg & Seemann book (handout)
- work on assignment for unit II (labII.1), which is now due on OCT 14 *(extended from original deadline)*
- new date for midterm: *WED OCT 19*