# Basic path finding and movement

# Readings

## Readings from the Bourg and Seemann text will begin with Chapter 6 and then move onto chapters 7, 2, 3, 4 and 5 in that order. The material toward the end of chapter 4 and all of chapter 5 will be treated later.

# Pathfinding

## This is the process of moving the game character from its initial location to a desired location. There are many different sub-problems and many different solutions. Note that we will separate the notion of path finding from the notion of movement. The basic notion is that there should be some path to follow before moving.

# Simple approach

## If the starting x position is greater than the destination x position, then decrement the x position. If the starting x position is less than the destination x position, then increment x

## If the starting y position is greater than the destination y position, then decrement the y position. If the starting y position is less than the destination y position, then increment y

## Goto the new x,y position

## Wrap the if\_then logic in a function or method. Call the function or method on a schedule

## Each time the function or method is called the game character moves closer to the destination.

# Example

## // 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

# Avoid the obstacle

## If the character encounters an obstacle that prevents the move to the new x,y position, randomly move to the nearest open x,y position and try again. This may be useful in some cases but in general it is not a good approach.

## Problems:

### How to detect the obstacle?

### How to find the open position?

### How to determine ‘nearest’?

### How to avoid returning to the same blocked position?

# Example

## 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,

## 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

# Tracing the obstacle

## Establish two states: findPath and trace. Start in findPath state.

## Continue in the simple path finding algorithm until an obstacle is detected

## When obstacle is detected enter the trace state.

## When in the trace state follow the edge of the obstacle.

## Continue tracing until ‘free’. So what is it to be ‘free’?

## Simple approach

### When the obstacle is detected, determine the straight line path to the destination.

### When the character crosses the straight line path, enter the findPath state

# Breadcrumbs

## The game character adds a third state, followCrumb

## The player character creates a path by drooping breadcrumbs. This could be further modified by having the breadcrumb marker decay over time.

## Each time the player character takes a step, it leaves a mark on the game world. Player is unaware of this!

## The game character moves by using the steps while in findPath.

## When the breadcrumb is encounter the NPIC enters the followCrumb state.

## While in the followCrumb state, examine the neighbors.

## Go to the first neighbor position that contains the player character breadcrumb.

## Determine whether the new position is the destination position.

### If it is stop

### If it is not the destination and there is neighbor with a breadcrumb, go there; otherwise enter findPath

# Generalizing the problem

## The intelligent agent (NPIC) should distinguish between open cells and closed cells.

### Closed cells are the cells that are either barriers or have already been occupied.

### Opens cells are the neighbors that are not closed.

## Keep open and closed lists. Latter you might want to subdivide the closed list into barrier and explored cells.

# Breadcrumb problem again

### Go to the first neighbor position that contains the player character breadcrumb.

## This needs to be modified since it may happen that none of the neighbors on the open list contain a breadcrumb. What to do?

### The character remembers the direction of its previous movement and continues in the same direction if the cell is open; otherwise randomly pick an open cell.

### The character determines which neighbor cells are open and randomly picks one.

# The intelligent agent expands

## The requirements for the non-player intelligent charter (NPIC) are expanding.

### The NPIC must sense characteristics about neighbors

#### At this point only sense barrier and breadcrumb.

### The NPIC must remember previous movement direction.

### The NPIC must remember its open and closed lists.

Keep a list of the things required for the NPIC. These will be used to define the character. You should include the fields or properties that character needs to have position, motion and so on as well as any other internal states. You should also keep a listing of the different kinds of behaviors that might be legal for the NPIC. Note not every NPIC will have the same set of properties and behaviors.

# Path-following

## “Pathfinding is often thought of solely as a problem of moving from a starting point to a desired destination. Many times, however, it is necessary to move computer-controlled characters in a game environment in a realistic way even though they might not have an ultimate destination. For example, a car-racing game would require the computer-controlled cars to navigate a roadway. Likewise, a strategy or role-playing game might require troops to patrol the roads between towns.”

## Not all cells are part of the path. The NPIC can leave the path but generally something bad happens. So the bounds of the path are not barriers.

## At this point it is not so much that the NPIC has found a path to follow as that there is some predefined path that must be followed.

## Add path sensing to the NPIC

### Depending on what you are doing you might sense the dropped breadcrumbs.

## Remember the previous direction.

### Ok the text uses a numbering scheme; directions 1 through 8. I would suggest that constants with nice names be used

## Assign weights to the different directions where the highest weight goes to continuing in the same direction and the lowest weight to reversing direction.

## Instead of the closed list containing the barrier and explored cells, we will need to take care of several things.

## If there are no barriers and the path is closed, then create a bounds list to act as the closed list. If there are barriers and the path is closed, then create a closed list and a bounds list.

## If the path is not closed, follow the previous path finding procedure.

## If a neighbor is not a bound or a barrier, place it on the open list.

## Order the open list according to the weight scheme

## Select the movement with the greatest weight.

## This can be modified by allowing the cells on the bounds list to be included on the open list but this would ordinarily also require some sort of penalty for leaving the path and we have not examined penalties!

## “the troll continuously circles the road. In a real game, you could make the computer-controlled adversaries continuously patrol the roadways, until they encounter a player. At that point the computer-controlled character's state could switch to an attack mode.”

## “In this example, we used the adjacent tiles to make a weighted decision about direction to move in next. You can increase the robustness of this technique by examining more than just the adjacent tiles. You can weight the directions not just on the adjacent tiles, but also on the tiles adjacent to them. This could make the movement look even more natural and intelligent.”

# Wall tracing

## “Another method of pathfinding that is very useful in game development is wall tracing. Like path-following, this method doesn't calculate a path from a starting point to an ending point. Wall tracing is more of an exploration technique. It's most useful in game environments made of many small rooms, although you can use it in maze-like game environments as well. You also can use the basic algorithm for tracing around obstacles, as we described in the previous section on obstacle tracing. Games rarely have every computer-controlled adversary simultaneously plotting a path to the player. Sometimes it's desirable for the computer-controlled characters to explore the environment in search of the player, weapons, power-ups, treasure, or anything else a game character can interact with.”

## Wall tracing is a good exploratory technique. This is especially so for ‘room’ based games.

## Use the left-handed approach

## Determine orientation issues: Where is left?

## Change the weighting scheme of the path-following technique as follows

### Always prefer a left turn over going straight ahead or taking a right turn.

### Always prefer going straight over going right.

# Waypoints

## “Pathfinding can be a very time-consuming and CPU-intensive operation. One way to reduce this problem is to precalculate paths whenever possible. *Waypoint navigation* reduces this problem by carefully placing nodes in the game environment and then using precalculated paths or inexpensive pathfinding methods to move between each node. Figure 6-19 illustrates how to place nodes on a simple map consisting of seven rooms.”

## In Figure 6-19, you'll notice that every point on the map is in the line of sight of at least one node. Also, every node is in the line of sight of at least one other node. With a game environment constructed in this way, a game-controlled character always will be able to reach any position on the map using a simple line-of-sight algorithm.”

## The game AI simply needs to know how the nodes connect to one another. New opportunities to try out those search algorithms!

## “Each method we discussed here has its advantages and disadvantages, and it's clear that no single method is best suited for all possible pathfinding problems. Another method we mentioned at the beginning of this chapter, the A\* algorithm, is applicable to a wide range of pathfinding problems. The A\* algorithm is an extremely popular pathfinding algorithm used in games, and we devote the entire next chapter to the method.”

# Generalized sketch for path finding

## This is a general path finding technique (procedure/method/algorithm)

## To find a path from a starting node to an ending node

## find\_path (start\_node, end\_node){

## Create a list for holding the reachable nodes and another list for the explored nodes

## The reachable nodes are in the open\_list and the explored nodes are in the closed\_list

## Put the starting node into the open list

## open\_list = [start\_node]

## closed\_list = []

## Loop through the open list while it is not empty

## while open\_list is not empty{

## Select a node that is in the open list, a reachable node.

## current\_node = choose\_node(open\_list)

## See the following discussion for how to **choose a node**

## If the current node is the goal node, build and return the path.

## The method to build a path is listed latter.

## if node == goal\_node{

## return build\_path(goal\_node)}

## Remove the current node from the open list and add it to the closed list. This moves the node from the list of reachable nodes and places it in the list of explored nodes. In brief, don't repeat what you have done.

## open\_list.remove(current\_node)

## closed\_list.add(current\_node)

## Ok now what? Find more reachable nodes. Take the nodes that are adjacent to the current node and are not in the closed list and place them in a new open adjacent

## adjacent\_open\_list = get\_adjacent\_nodes(current\_node) - closed\_list)

## Now loop through the nodes in the adjacent open list

## for adjacent\_node in adjacent\_open\_list{

## If the adjacent node is not already in the open list

## if adjacent\_node not in open\_list{

## then for the previous property of the current adjacent node set it to the current node being examined. That is remember how we got to this node.

## adjacent\_node.previous = current\_node

## and then add the current adjacent node to the open list.

## open\_list.add(adjacent\_node)}}}

## If there is nothing in the open list, then there is no path

## return None}

## Note that are three methods to consider: choose\_node, build\_path, and get\_adjacent\_nodes

# Discussion of build\_path

## Build a path from the current path

## build\_path (to\_node){

## The path is a list of nodes. Create the path.

## path = []

## Loop into the current node

## while to\_node != None{

## Add the current node to the path

## path.add(to\_node)

## Make the node stored in the previous property of the current node the current node.

## to\_node = to\_node.previous }

## If there is a non-None node in the previous property the loop will continue and add more nodes to the path; otherwise path building is done, so return the path

## return path}

# See the discussion of get\_adjacent\_nodesandchoose\_node

## OK so far we have a general strategy for path finding. See strategy pattern for other specifics: https://sourcemaking.com/design\_patterns/strategy or the https://sourcemaking.com/design\_patterns/template\_method

## We need to provide ‘plug-ins’ for the two remaining functions or methods.

## This approach allows you to customize find\_path and tailor it to your needs

## You might want to do this a scratch work from your notebook or build it in appropriate code.

# Discussion of get\_adjacent\_nodes

## There are several considerations in building this.

## The first and most important is determining what it is to be ‘adjacent’. Consider what happens if movement is restricted to four directions. There are four adjacent cells. For eight-way movement, there are eight adjacent cells.

## Note also that this restricts how far out the ‘reach’ is. How to determine whether a node is reachable? We have assumed only one cell at a time. Might this be two cells? Might there be a maximum distance if line-of-sight is used? Might there be an angle distance to consider?

## Each consideration would provide a different function, method or algorithm for producing a list of adjacent nodes. (Yes, ‘adjacent’ has an odd meaning here.)

## These different definitions would produce different behaviors.

## However, the basic signature remains constant. Given some node, produce a list of adjacent nodes.

## Start building these!

# Discussion of choose\_node

## A similar approach will be used for choose\_node

## There are many ways to implement this function or method. What we know is the constant of the signature. Given a list of nodes, select one.

## You might pick the first node in the list, a random node, one that maximizes or minimizes some value or one with a particular characteristic. Different behaviors would result from your selection. Internally you might need to sort the list in some way or you might need to do various computations to get a minimum, maximum or specific feature.

## You may need a multi-step approach. Try approach A. If it fails, then try B. And so on. This can provide some graceful degradation.

## Go and build

# Discussion of open\_list.add(node)

## There is an issue here. How should the node be added?

## There is a coupling of open\_list.add and choose\_node. The open\_list.add function or method might do some of the work required for choose\_node

## Consider whether you add nodes at the beginning or end of the open list. Placing the node at the beginning will produce a depth-first behavior while placing them at the end will produce a breadth-first behavior. If this is the only relevant distinction, then choose\_node is simply “select the first node on the list”.

## Note that once again the signature remains the same, but the behavior might be different.

## Go and build!

## As you go along you will find various customizations for path finding. Keep track of these.

## You can then make these plug-ins for the generalized path-finding strategy.

## These should at least go in your notebook which will become the final report for the course. Notebook? Yes the notebook will provide the materials for the report part of the grade.

# A path or the best path?

## Let’s begin by remembering this part of the generalized strategy:

## if adjacent\_node not in open\_list{

## adjacent\_node.previous = current\_node

## open\_list.add(adjacent\_node)}

## There is an issue here. Does the current node provide a better path? In the previous general form we simply remembered how we got to the node and did not have any way to determine if it was the best way.

## Let’s assume for the moment that the best path is the shortest path.

## Need to add some book-keeping (memory). We need to know the length of the path from the start node to any reachable node. Call this the cost of the path. We can assume for the moment that the cost of moving from one node to another adjacent nodes has a fixed cost of 1.

## At the start of the search, set the cost of each node to the max value of infinity; this makes *any* path shorter than that. Also set the cost of the start\_node to 0.

## In implementation there will need to be new properties or states for all nodes.

## We can either change the main generalized code or add the appropriate methods or functions. Yes that is a design decision and it has consequences. This sort of design decision work should be reflected in your notebook.

## In general form need something like the following

## if adjacent\_node not in open\_list{ open\_list.add(adjacent\_node)

## If this is a new path, or a shorter path than what we have, keep it.

## if current\_node.cost + 1 < adjacent\_node.cost{ adjacent\_node.previous = current\_node

## adjacent\_node.cost = current\_node.cost + 1}}

# Uniform cost

## Now consider choose\_node.

## A good idea is to choose the reachable node on the open list that has the shortest path from the start\_node. This will generally choose shorter paths over longer ones even though longer paths may be considered. Shorter paths will be considered first. Since our general strategy stops as soon as a valid path is found, this will get short paths.

## Define the choose\_node method or function so that it checks costs and returns the node with the best (least) cost.

## choose\_node (open\_list){

## best\_node = None

## for a\_node in open\_list{

## if best\_node == None or (best\_node.cost > a\_node.cost){

## best\_node = a\_node

## return best\_node}}}

# You and the crow

## Now we need just a bit more. You walk and the crow flies. You will have a path length and the crow will have a distance. That is the *shortest path* and the *minimum distance* are different: the minimum distance assumes there are no obstacles between the current node and the goal node.

## Now we already know some things about distance so we can add these to our choose\_node method or function to produce A\*

# A\* : choose\_node

## Here is A\* where we would add the appropriate distance measure for estimate\_distance

## choose\_node (open\_list){

## min\_cost = infinity

## best\_node = None

## for a\_node in open\_list{

## cost\_start\_to\_node = a\_node.cost

## cost\_node\_to\_goal =

## estimate\_distance(a\_node,goal\_node)

## total\_cost = cost\_start\_to\_node + cost\_node\_to\_goal

## if min\_cost > total\_cost{

## min\_cost = total\_cost

## best\_node = a\_node} }

## return best\_node}

# Movement

# A big picture: Starting with a classic

## **Steering Behaviors For Autonomous Characters** byCraig W. Reynolds, Sony Computer Entertainment America (1999, approximately 1300 citations) Now see: http://www.red3d.com/cwr/ Craig W. Reynolds (born March 15, 1953), is an artificial life and computer graphics expert, who created the Boids artificial life simulation in 1986. Reynolds worked on the film *Tron* (1982) as a scene programmer, and on *Batman Returns* (1992) as part of the video image crew. Reynolds won the 1998 Academy Scientific and Technical Award in recognition of “ his pioneering contributions to the development of three-dimensional computer animation for motion picture production." He is the author of the *Open Steer* library. Wikipedia. See http://opensteer.sourceforge.net/

# A starting point

## *Autonomous characters* are a type of *autonomous agent* intended for use in computer animation and interactive media such as games and virtual reality. These agents represent a character in a story or game and have some ability to improvise their actions. This stands in contrast both to a character in an animated film, whose actions are scripted in advance, and to an avatar in a game or virtual reality, whose actions are directed in real time by a human player or participant. In games, autonomous characters are sometimes called *non-player characters*.

## A simple locomotion model will be presented in order to make the discussion of steering behaviors more concrete. This locomotion model will be based on a simple idealized *vehicle*.

## This vehicle model is based on a point mass approximation.

## A point mass is defined by a *position* property and a *mass* property. In addition, the simple vehicle model includes a *velocity* property. The velocity is modified by applying forces. Because this is a vehicle, these forces are generally self-applied, and hence limited. For example, a typical force which adjusts a vehicle’s velocity is *thrust*, generated by the vehicle’s own power plant, and hence limited in magnitude by the capacity of the power plant. For the simple vehicle model, this notion is summarized by a single ‘maximum force’ parameter (*max\_force*). Most vehicles are characterized by a top speed. Typically this limitation is due to the interaction between acceleration due to their finite thrust and the deceleration due to viscous drag, friction, or (in legged systems) the momentum of reciprocating parts. As an alternative to realistic simulation of all these limiting forces, the simple vehicle model includes a ‘maximum speed’ parameter (*max\_speed*). This speed limit is enforced by a kinematic truncation of the vehicle’s velocity vector. Finally, the simple vehicle model includes an *orientation*, which taken together with the vehicle’s position form a velocity-aligned local coordinate space to which a geometric model of the vehicle can be attached. (The terms *localize* and *globalize* will be used in this paper to connote transforming vectors into and out of this local space.)

## Simple Vehicle Model:

The properties or fields

### mass scalar

### position vector

### velocity vector

### max\_force scalar

### max\_speed scalar

### orientation *N* basis vectors

# The behaviors

## seek and flee

## pursue and evade

## offset and arrive

## obstacle avoidance

## wander

## follow path

## arrive

# The behaviors: seek and flee

## **Seek** (or pursuit of a static target) acts to steer the character towards a specified position in global space. This behavior adjusts the character so that its velocity is radially aligned towards the target.

## **Flee** is simply the inverse of **seek** and acts to steer the character so that its velocity is radially aligned away from the target.

# The behaviors: pursue and evade

## **Pursuit** is similar to **seek** except that the quarry (target) is another moving character. Effective pursuit requires a prediction of the target’s future position. The approach taken here is to use a simple predictor and to reevaluate it each simulation step. Steering for **pursuit** is then simply the result of applying the **seek** steering behavior to the predicted target location.

## **Evasion** is analogous to **pursuit**, except that **flee** is used to steer away from the predicted future position of the target character.

# The behaviors: offset and arrive

## **Offset pursuit** refers to steering a path which passes near, but not directly into a moving target. The basic idea is to dynamically compute a target point which is offset by a given radius R from the predicted future position of the quarry, and to then use **seek** behavior to approach that offset point.

## **Arrival** behavior is identical to **seek** while the character is far from its target. But instead of moving through the target at full speed, this behavior causes the character to slow down as it approaches the target, eventually slowing to a stop coincident with the target.

# The behaviors: obstacle avoidance

## **Obstacle avoidance** behavior gives a character the ability to maneuver in a cluttered environment by dodging around obstacles. There is an important distinction between **obstacle avoidance** and **flee** behavior. **Flee** will always cause a character to steer away from a given location, whereas **obstacle avoidance** takes action only when a nearby obstacle lies directly in front of the character.

# The behaviors: wander

## **Wander** is a type of random steering. One easy implementation would be to generate a random steering force each frame, but this produces rather uninteresting motion. It is “twitchy” and produces no sustained turns. A more interesting approach is to retain steering direction state and make small random displacements to it each frame. Thus at one frame the character may be turning up and to the right, and on the next frame will still be turning in almost the same direction.

# The behaviors: follow path

## **Path following** behavior enables a character to steer along a predetermined path, such as a roadway, corridor or tunnel. This is distinct from constraining a vehicle rigidly to a path like a train rolling along a track. Rather **path following** behavior is intended to produce motion such as people moving down a corridor: the individual paths remain near, and often parallel to, the centerline of the corridor, but are free to deviate from it.

# The behaviors: arrive

## **Arrival** behavior is identical to **seek** while the character is far from its target. But instead of moving through the target at full speed, this behavior causes the character to slow down as it approaches the target, eventually slowing to a stop coincident with the target. The distance at which slowing begins is a parameter of the behavior. This implementation is similar to **seek**: a desired velocity is determined pointing from the character towards the target. Outside the stopping radius this desired velocity is clipped to *max\_speed*, inside the stopping radius, desired velocity is ramped down (e.g. linearly) to zero.

# Medium size picture

# Kinematic

## Does not consider acceleration

## Is essentially one-speed

## Agent starts at full speed, moves to desired location, and stops

# Steering behavior

## The behavior is dynamic since it changes with time. Physics-based

## Called steering behavior because it controls direction of agent

## Algorithm requires agent's current position, velocity, and forces acting on the agent

## Output is resultant force or acceleration acting on agent

## The output is used to modify the agent's current velocity

## The overall behavior for an agent starting at rest:

### The agent accelerates to speed

### Cruises until it nears destination (target)

### Decelerates to a stop

# Simple chase and evade

## Chase

## if (predatorX > preyX)

## predatorX--;

## else if (predatorX < preyX)

## predatorX++;

## if (predatorY > preyY)

## predatorY--;

## else if (predatorY < preyY)

## predatorY++;

## Evade

## if (preyX > predatorX)

## preyX++;

## else if (preyX < predatorX)

## preyX--?>;

## if (preyY > predatorY)

## preyY++;

## else if (preyY < predatorY)

## preyY--;

# It works but…!

## In *tile-based games* the game domain is divided into discrete tiles squares, hexagons, etc. and the player's position is fixed to a discrete tile. Movement goes tile by tile, and the number of directions in which the player can move is limited. In a *continuous environment*, position is represented by floating-point coordinates, which can represent any location in the game domain. The player also is free to head in any direction.

## The simple approach can be applied to both tile-based or continuous environment games. In tile-based games, the *x*s and *y*s can represent columns and rows in a grid that encompasses the game domain. In this case, the *x*s and *y*s would be integers. In a continuous environment, the *x*s and *y*s and *z*s if yours is a 3D game would be real numbers representing the coordinates in a Cartesian coordinate system encompassing the game domain.

# Line-of-sight with tiled world

## Even with 8 way movement one cannot actually produce a correct line of sight movement, but you can come close!

## Even though you can’t draw a straight line, you can avoid going to two adjacent cells along the shortest axis

# Bresenham’s algorithm

## The Bresenham algorithm is used to calculate the direction of the predator's movement given the starting point, which is the row and column of the predator's position, and the ending point, which is the row and column of the prey's position, and calculates a series of steps the predator will have to take so that it will walk in a straight line to the prey.

## This is a static path from predator to prey. This path would need to be called each time the predator's target, the prey, changes position. Once the target moves, the precalculated path becomes obsolete, and therefore it must be calculated again. This can provide for seek behavior.

# The Characters

## Let’s begin by assuming that we have character structures (objects) that can store their row and column locations. There will be one for the predator and one for the prey.

## In terms of our general discussion on NPIC this means that we would need field or states in a programming structure for these and, of course, some accessor methods.

# The Technique

## BuildPathToTarget (){

## // set up the basic variables and initialize

## int nextCol=col; // col stored in NPIC

## int nextRow=row; // row stored in NPIC

## int deltaRow=endRow-row; // endRow stored in target

## int deltaCol=endCol-col; //endCol store in target

## int stepCol, stepRow;

## int currentStep, fraction;

## // initialize the path in terms of rows and columns. So what should the value for kMaxPathLength be?

## for (currentStep=0;currentStep<kMaxPathLength; currentStep++){

### pathRow[currentStep]=-1;

### pathCol[currentStep]=-1;

### }

## currentStep=0;

## pathRowTarget=endRow;

## pathColTarget=endCol;

## //determines the direction of the path by using deltaRow and deltaCol values.

## if (deltaRow < 0) stepRow=-1; else stepRow=1;

## if (deltaCol < 0) stepCol=-1; else stepCol=1;

## deltaRow=abs(deltaRow\*2);

## deltaCol=abs(deltaCol\*2);

## pathRow[currentStep]=nextRow;

## pathCol[currentStep]=nextCol;

## currentStep++;

## //uses the values in deltaCol and deltaRow to determine which axis is the longest

### if (deltaCol >deltaRow){

### fraction = deltaRow \*2-deltaCol;

#### while (nextCol != endCol){

##### if (fraction >=0){

##### nextRow =nextRow +stepRow;

##### fraction =fraction -deltaCol;}

#### nextCol=nextCol+stepCol;

#### fraction=fraction +deltaRow;

#### pathRow[currentStep]=nextRow;

#### pathCol[currentStep]=nextCol;

#### currentStep++;}}

## else{

### fraction =deltaCol \*2-deltaRow;

### while (nextRow !=endRow){

#### if (fraction >=0){

#### nextCol=nextCol+stepCol;

#### fraction=fraction -deltaRow;}

### nextRow =nextRow +stepRow;

### fraction=fraction +deltaCol;

### pathRow[currentStep]=nextRow;

### pathCol[currentStep]=nextCol;

### currentStep++;}}

## }

## Review

## All game objects can be defined as having a position and an orientation.

## In some game types a movement algorithm can directly update the position/orientation (e.g. tile-based). However, this will look unrealistic in other types of game (e.g. driving).

## Kinematic movement algorithms operate using positions and orientations. The output is a target velocity (speed + orientation).

## The speed may simply vary between full speed and stationary, i.e. kinematic algorithms do not use acceleration.

# Kimematic

## Need

### Vector position

### float orientation;

### Vector velocity;

### float rotation;

## Updating (using Newton Euler equations)

### velocity += acceleration \* time\_delta

### rotation += angular\_acc \* time\_delta

### position += velocity \* time\_delta

### orientation += rotation \* time\_delta

## Assume that a suitable mathematics module/library is available

# Kinematic seek

## Seek

### Seek ( Vector source, Vector target,float maxSpeed )

### {Vector velocity = (target – source).normalise()\* maxSpeed;

### return velocity;

### }

## *Orientation*

### DetermineOrientation(Vector velocity, float currentOrientation ) {

## if( velocity.length() == 0 )

### return currentOrientation;

### else

### return Math.atan2( -velocity.x, velocity.y)

### }

### The function **atan2** is the arctangent function with two arguments. The purpose of using two arguments instead of one is to gather information on the signs of the inputs in order to return the appropriate quadrant of the computed angle. **atan2** computes the arctangent of y/x in a range of (−π, π), i.e. it determines the counter clockwise angle (radians) between the x-axis and the vector <x,y> in 2D Euclidean space.

# Kinematic flee

## Flee( Vector source, Vector target, float maxSpeed ) {

## Vector velocity

## = (source – target).normalise() \* maxSpeed;

## return velocity;

## }

# Kinematic arrive

## Arrive ( Vector source, Vector target,

## float maxSpeed, float nearRadius ) {

## float slowingFactor = 0.2;

## Vector velocity = [0,0,...];

## Vector separation = (target – source);

## if**(** separation.length**() <** nearRadius)

## return velocity;

## //Determine velocity, and cap at max speed if needed

## velocity **=** separation **/** slowingFactor;

## if( velocity.length**() >** maxSpeed)

## velocity = velocity.normalise() \* maxSpeed;

## return velocity;

## }

# Steering movements

## Steering behaviours extend the kinematic movement algorithms by determining acceleration (both forward movement and rotation)

## In many game types (e.g. driving games) steering algorithms are often used. In other games, they may not be useful.

## We will consider some of the following forms of steering behaviour:

## Seek(), Flee(), Arrive(), Wander(), Pursue(), Evade(), Interpose(), Align(), Face(), Separate(), PathFollow(), AvoidObstacle(), Jump()

# Flee and Seek

## Basic steering algorithms operate by trying to match some kinematic property of the target to the source, e.g. this might be the target’s position, velocity, orientation, etc. Matching steering algorithms take source and target kinematic properties as input.

## More advanced steering behaviours try to match a combination of properties, potentially with additional constraints.

## Typically for each matching behaviour there is a readily defined opposite behaviour (e.g. Seek vs. Flee, etc.).

# Steering seek

## Seek will try to match the source position to a target location.

## As with the kinematic seek the direction to the target is determined, and a corresponding maximum acceleration set towards the target.

## Seek ( Vector source, Vector target, float maxAcc ) {

## Vector acceleration = (target – source).normalise() \* maxAcc;

## return acceleration;

## }

## Seek will accelerate as fast as possible towards the target. Some means of dampening/reducing velocity will be needed as the target is approached – this is offered by other forms of steering behavior.

## The output of Seek is used to update the position and velocity as follows:

## UpdatePosition() {

## //timeDelta amount of time since last update

## velocity += acceleration \* timeDelta;

## rotation += angular\_acc \* timeDelta;

## if( velocity.length() > maxSpeed )

## velocity = velocity.normalise()\*maxSpeed;

## position += velocity \* timeDelta;

## orientation += rotation \* timeDelta;

## }

## The Seek algorithm does not change the orientation, i.e. other forms of steering algorithm can be used to align orientation with direction of movement

## Flee is simply the opposite of Seek, with a maximum acceleration away from the target output.

# Steering Arrive

## Arrive will control acceleration so that the velocity is zero once the target has been reached.

## To do this, two radii are used: an arrival radius when the target can be considered as reached, and a larger slow radius used to control when the velocity should be reduced from the maximum.

## The opposite to Arrive is Leave – although there is little need to have a behaviour that will slowly *accelerate* away from the target. It is more likely to have a behavior that accelerates away with maximum acceleration, i.e. Flee.

## Arrive(Vector source, Vector target,

## Vector currentVelocity,

## float maxAcceleration, float maxSpeed,

## float arriveRadius, float slowRadius ) {

## float slowingFactor = 0.2;

## Vector acceleration = [0,0,.];

## Vector direction = target – source;

## float distance = direction.length();

## if( distance < arriveRadius )

## return acceleration;

## { Determine target velocity }

## { Determine acceleration }

## return acceleration

## }

## //Determine target velocity

## //If outside slow radius, then max speed, else scale speed based on distance

## float targetSpeed;

## if( distance **>** slowRadius)

## targetSpeed = maxSpeed;

## else

## targetSpeed = maxSpeed \* distance / slowRadius;

## Vector targetVelocity =

## direction.normalise() \* targetSpeed;

## //Determine acceleration

## //Calculate acceleration so that it will slow down object near target

## acceleration = targetVelocity – currentVelocity;

## acceleration /= slowingFactor;

## if( acceleration.length() > maxAcceleration )

## acceleration = acceleration.normalise() \* maxAcceleration;

# Steering wander

## Wander will produce a movement that gives the impression of a random walk.

## In order to avoid jittery behaviour a circle can be projected in front of the object with steering towards a target that is constrained to move along the perimeter.

## Each update, the target is displaced by a small random amount (moving back and forth around the perimeter over time).

## By controlling size of circle, distance from object and random displacement, a wide range of motion can be generated.

## Wander( Vector source,

## float sourceOrientation, float maxAcceleration,

## float wanderOffset, float wanderRadius,

## float wanderRate, float wanderOrientation ) {

## //Work out new target orientation

## wanderOrientation += wanderRate\* randomBinomial();

## //Any random function will suffice, binomial is smooth

## targetOrientation = wanderOrientation + sourceOrientation;

## Vector target = source + wanderOffset \* asVector(sourceOrientation);

## target += wanderRadius \* asVector(targetOrientation)

## //Return acceleration based on a Seek towards the determine target location

## //Return a (2D) vector with the same orientation as the specified angle

## Vector2 asVector( float angle ) {

## return new Vector2(

## (float)Math.Cos(angle),

## (float)Math.Sin(angle) ); }

# Steering pursuit

## Seek will head directly towards the target. If the target is moving, it is better to aim towards where the target is likely to be in the future – a behaviour known as Pursuit.

## Different algorithms can be used to predict the likely future position. A simple, yet effective, approach is to assume the target will continue to move with the same current velocity.

## The target’s movement can then be used to calculate it’s future position, which is used as the seek position.

## Pursuit( Vector source, Vector target,

## Vector sourceVel, Vector targetVel ) {

## //Maximum future time in which to predict

## float maxPrediction;

## Vector direction = target – source;

## float distance = direction.length();

## float speed = sourceVel.length();

## //If time to target is too long, then use maximum prediction time, else determine time to contact

## if( speed <= distance / maxPrediction )

## prediction = maxPrediction;

## else prediction = distance / speed;

## //Determine new target location and return Seek acceleration towards it

## target += targetVel \* prediction;

## return Seek( source, target)

## }

## Evade is simply the opposite of Pursuit, i.e. the evaders flees from the projected future direction

# Compound pattern movement

## Assume that the NPIC is to travel a particular path that is prearranged.

## Let the path be in a loop. For example there are four known boundary points. An example would be to patrol an area. Assume also that there is some event that causes the NPIC to leave the loop and enter a different state.

## Each of the boundary points can be considered a target.

## Use one of the previously defined movements to go from the starting point to the next point on the path.

## In the example if there are four points and P0 is the starting point.

### while the state is patrol

### {move from P0 to P1

### move from P1 to P2

### move from P2 to P3

### move from P3 to P0}

## Suppose the path is more complex? That is there are more points.

## Make a list of points so that each item in the list is a pair of startPoint and endPoint. Note that the list of pairs should close the loop.

## Identify the place where the NPIC’s current point is a startPoint in the list. Select movement style and begin looping there.

# Waypoints

## A more general approach is to have the builder of the level provide waypoints for the level.

## A waypoint graph specifies lines/routes that are “safe” for traversing

## Each line (or link) connects exactly two waypoints

## An agent can choose to walk along any of these lines without having to worry about running into major obstacles

## Path is a graph structure

## Vertices are waypoints on path

## Edges between waypoints directly reachable from one another

## Basic idea is to use seek to go from one waypoint to another.

## Can use circles for proximity and speed modifications that allow a bit more realism.

# Flocking boids

## Boids for flocking behavior. Originated with Craig Renolds

## Boid-style flocking is rather straightforward and uses three rules.

# Boids an their rules

## Start by initializing the quantity and positions of the boids.

## If you place the boids outside of the viewable area that will appear to come from far away.

## Each boid has a velocity vector. Since the rules will act independently, each boid will calculate how it should move.

## Each rule produces its own velocity vector. Add those three to the boids current velocity vector to get the voids new velocity vector.

# Updating the flock

## update\_boids(){

## Vector v1, v2, v3

## Boid b

## for each boid b{

## v1 = rule1(b)

## v2 = rule2(b)

## v3 = rule3(b)

## b.velocity = b.velocity + v1 + v2 + v3

## b.position = b.position + b.velocity}}

# Rule 1: Cohesion

## Rule 1: Boids try to fly towards the center of mass of neighboring boids

## Typically the ‘center of mass’ is simply the average position of all the boids. Could be modified if boids had different masses.

## Assume there are N boids, called b1, b2, ..., bN. Also, the position of a boid b is denoted b.position. Then the ‘center of mass’ c of all N boids is given by:

## c = (b1.position + b2.position + ... + bN.position) / N

## Note that the positions are vectors, and N is a scalar.

## Now there is room for a bit of modification since the ‘center of mass’ is for the whole flock. It might be reasonable to change the formula to allow a particular boid to perceive the ‘center of mass’ of the rest of the flock.

## For boidJ (1<=J<=N), the perceived center pcJ is

## pcJ = (b1.position + b2.position + ... +

## bJ-1.position + bJ+1.position + ... + bN.position) / (N-1)

## Using the perceived center pick some percentage to move the boid toward the center. A matter of art determining the percentage. Call this value the move\_rate, so that

## rule1(boid bJ){

## Vector pcJ

## for each boid b{

## if b != bJ then

## pcJ = pcJ + b.position

## }

## pcJ = pcJ / N-1

## return (pcJ - bJ.position) \* move\_rate}

# Rule 2: Separation

## Rule 2: Boids try to keep a small distance (boid\_sep)away from other biods (and possibly other objects)

## This rule avoids boids colliding into each other. If a boid is within a defined small distance of another boid move it further away by the same distance as it already is.

## This is done by subtracting from a vector c the displacement of each boid which is nearby. Initialize c to zero since the rule should produce a vector which when added to the current position moves a boid away from those near it.

## rule2(boid bJ){

## Vector c = 0;

## for each boid b{

## if b != bJ then

## if |b.position - bJ.position| < boid\_sep then

## c = c - (b.position - bJ.position)

## }

## return c}

## Doubling the distance for repulsion can be tailored. If two boids are near each other, this rule will be applied to both of them. They will be slightly steered away from each other, and at the next time step if they are still near each other, they will be pushed further apart. This will appear to be a smooth acceleration. It is a good idea to maintain a principle of ensuring smooth motion.

## Doubling the distance is not, of course the only option. This can become more complex, but will require more calculations. Hence this factor will be a matter of art.

# Rule 3: Alignment

## Rule 3: Boids try to match velocity with nearby boids.

## Similar to Rule 1. Instead of averaging the positions of the other boids, average the velocities. As in Rule 1 calculate the ‘perceived velocity’, pvJ. Then add a small part of that value (boid\_vchange) to the boid’s current velocity.

## rule3(boid bJ){

## Vector pvJ

## for each boid b{

## if b != bJ then

## pvJ = pvJ + b.velocity

## }

## pvJ = pvJ / N-1

## return(pvJ - bJ.velocity) \* boid\_vchange

## }

# Rule Summary

## Rule 1 Cohesion

### Every boid attempts to move towards the average position of other nearby boids.

## Rule 2 Separation

### Each boid attempts to maintain a reasonable amount of distance between itself and any nearby boids, to prevent overcrowding

## Rule 3 Alignment

### Boids try to change their position so that it corresponds with the average alignment of other nearby boids

# Tailor the rule set

## Easiest way to do this is to add another rule in update\_boids and add its contribution to the boid’s velocity at each step.

## If there is a constant wind or current, add a rule for this. This is a simple one. Complexity can be added

## wind\_rule(Boid b){

## Vector wind

## return wind)

## Suppose the flock needs to steer to a place. Need a place which is the goal\_place and a rate of change toward that goal (goal\_rate).

## move\_to\_goal\_rule(Boid b){

## Vector goal\_place

## return (goal\_place - b.position) / goal\_rate }

# Question for you!

## Now what is

## in the NPIC?

# Path finding (cont)

# Some points to think about

## The text and class are sources of information. There are more and these often provide the details that you might need to implement some design.

## Amit’s Game Programming Information is a great source for information http://www-cs-students.stanford.edu/~amitp/gameprog.html

## The current relevant section is http://www-cs-students.stanford.edu/~amitp/gameprog.html#tiles

## Also see http://www.redblobgames.com/

# The result is

# For a bit more

## Check the simple code on the canvas site.

## There is a bit of code there as well.

# But wait there is more!

## See: http://www.redblobgames.com/grids/line-drawing.html

### Linear interpolation

### Grid walking

## And remember for path finding

### https://qiao.github.io/PathFinding.js/visual/

### http://www.cokeandcode.com/main/tutorials/path-finding/