Dept. of Computer Science & Software Eng., Concordia University
COMP 476/6331 --- Winter 2020

Advanced Game Development

Assignment 2 --- Due March 2, 2020

PURPOSE: This assignment will give you the opportunity to learn more about programming AI behaviour. This assignment contains both **theoretical** questions, which you don't need to implement, and a **practical** question, which requires you to write a C#/Unity program.

SUBMISSION: The assignments must be done individually. On-line submission is required (see details at end) – a hard copy will not be read or evaluated.

PREPARATION: Parts of this assignment require knowledge of basic concepts from parts of the material covered in the course slides (the course schedule lists which exact sections of which books the material is based on).

Question #1: (10%) [Theoretical Question]

We can represent a weighted directed graph like as a set of nodes and edges (with assigned weights indicated):

nodes = {S, A, B, C, D, E, G}
edges = { SA: 3, SB: 10, AB: 5, AD: 6, AC: 9, BC: 3, BG: 15, DE: 6,
CD: 9, CE: 7, CG: 6, EG: 3}.

In the graph, assume that *S* is the start node and *G* is the goal node.

a) (5%) If we use Dijkstra's algorithm to find the minimum cost path from S to G, then the following table shows the contents of the open and closed lists for the first 2 steps of the algorithm. Fill in the remaining lines. Each entry in the lists is of the following format: (Node, Cost-So-Far, Connection). Clearly indicated updated entries. Stop when the *guaranteed* shortest path has been found.

Current Node	Open list	Closed list
–	(S,0,–)	
S	(A,3,SA), (B,10,SB)	(S,0,–)

b) (5%) For a node *A*, we can specify that the value of the heuristic function at the node is 5 by writing *A* : 5. Update the graph for part a) with the following heuristic values:

nodes = {S : 11, A : 12, B : 4, C : 11, D : 4, E : 9, G : 0}

If the A* algorithm is applied to the same graph as in question 2, the following table shows the contents of the open and closed lists for the first 2 steps. Fill in the remaining lines. Each entry in the open list is now of the following format: (Node, Cost-So-Far, Connection, Estimated-Total-Cost). The entries for the closed list are the same as before. Stop when the *guaranteed* shortest path has been found.

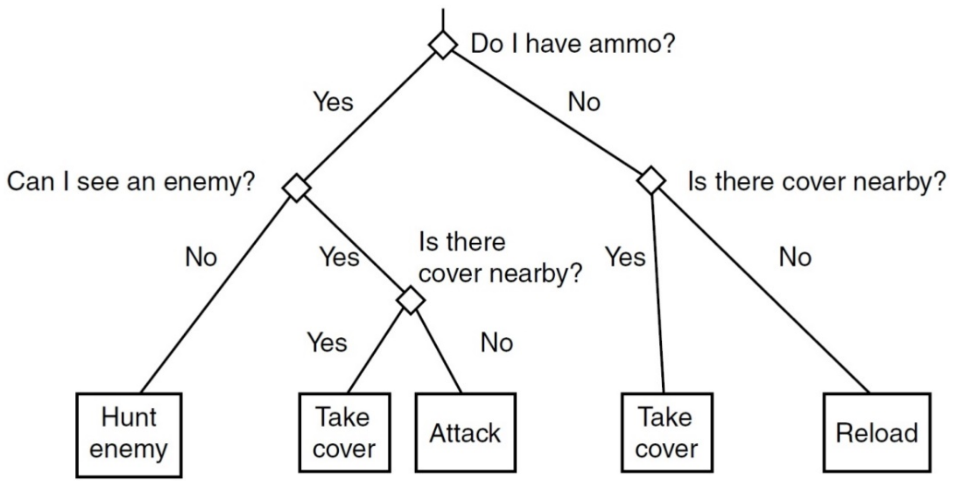
Current Node	Open list	Closed list
–	(S,0,–,11)	
S	(A,3,SA,15), (B,10,SB,14)	(S,0,–)

Question #2: (10%) [Theoretical Question]

Consider the decision tree from Figure 6.6 of Artificial Intelligence for Games 2nd Edition, by Millington and Funge, reproduced below.

a) (6%) Design a hierarchical finite state machine that would produce behaviour similar to that of the decision tree.

b) (4%) Add an alarm behaviour/mechanism (see Slide 30 of Week 4). To your hierarchical finite state machine from a), add alarm behaviour such that the NPC, regardless of what is going on, will go to sleep (somewhere relatively safe) if too tired. Add the option of taking a quick nap if no enemies have been visible for a while and the NPC has ammo.



Question #3: (80%) [Practical Question]

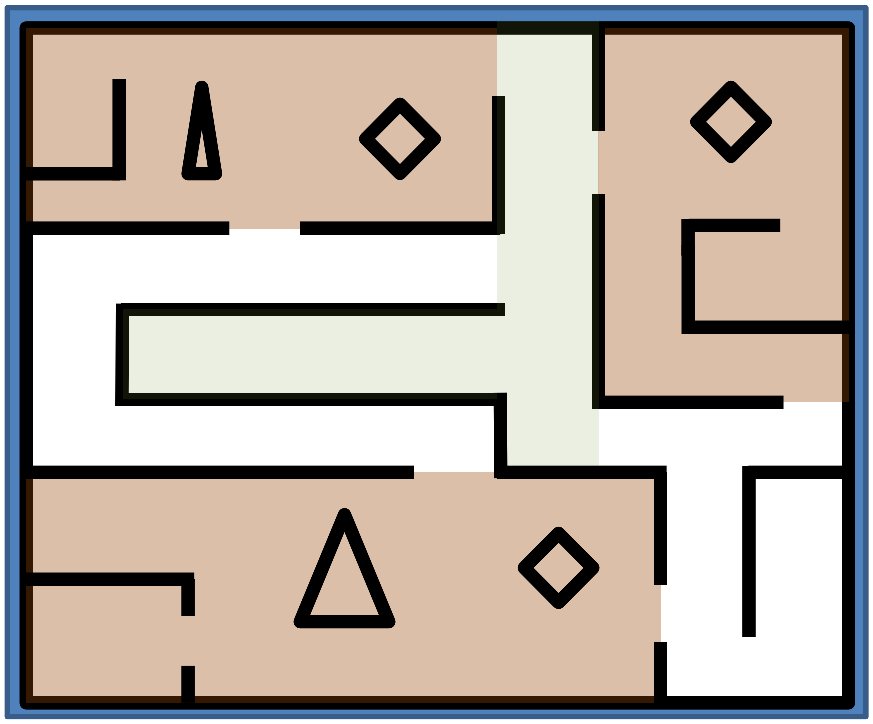
Problem Statement:

For the question you are to write a Unity program in C# to have an NPC perform path following. Your program must comprise of the following:

R1) Level and Pathfinding Graphs

- Using Unity as your game engine, define an environment in the form of a 2½D level which is closed (the movement is only within the geometry defined below). Make sure the (stationary) view point can see the entire level (i.e., don't make the walls of the level too high). Fill the level with the following geometry (*must* have all these, at least):
  - Three completely disjoint large rooms (disjoint meaning they are not touching, not even sharing common walls or doorways). Place these three rooms at opposite corners of the level and have their interiors together occupy about one-third to one-half of the area of the level.
  - Within each one of these three large rooms, connected by an open door, place a small room (like a walk-in closet or slightly larger) that has no other exits.
  - For each of the large rooms, place two open doorways to two different long corridors that connect to the other rooms. Each of the three long corridors must have at least one right-angle turn, may split and/or merge, and may share common junctions with other corridors.
  - Off of one or two of these long corridors place a "dead-end" corridor.
  - Within each large room place one or two relatively large static convex polygonal (not round) obstacles, not so close to a wall that the NPC can't move around it.

- Below is an example level showing the minimal requirements.



- You are to create **two** different pathfinding modes, one is mesh-based using Unity's Navmesh, and one is based on a pathfinding graph using Points of Visibility.
  - For Navmesh pathfinding, simply bake the level as a Navmesh.
  - For the points of visibility (PoV) graph, following the description given in the class slides, place nodes at critical shortest path positions (keeping in mind the size of your NPC): centre of doorways, at the corners of door frames, around objects in rooms, at the corners of rooms, at the corners of corridors, etc. Also, place visibility points in the centres of voids, middle of corridors, etc., so that you end up with at least 30 nodes. Connect each visibility point with directed edges to all the other visibility points it can "see" (so that a straight edge between the nodes does not intersect a wall or obstacle; you can do this by visual inspection). The PoV graph should not use Unity's Navmesh in any way.
- For the PoV pathfinding graph, make sure there are enough nodes such that there are a variety of paths through the room, and around the level. Render these nodes (simply as points or small discs) on the floor of the level. The weights of the edges of the graph will be the Euclidean (shortest) distance between the endpoints.

R2) A* Algorithm:

- Implement the A* algorithm with three different heuristics to run on your pathfinding graph (the lab material will cover similar material). First will be the null heuristic (yes, this is just Dijkstra's algorithm). The second will use the Euclidean Distance heuristic. The third will use the Cluster heuristic.
- As discussed in the course slides, for the Cluster heuristic consider the nodes in each room to be a cluster, and each corridor to contain a cluster (in the figure above, different regions of the level are coloured according to which cluster the pathfinding nodes in that region belong to; there are 6 clusters in my example). Using Dijkstra's algorithm, between each pair of clusters compute the shortest path length between

any two nodes (one from each cluster). Use these results between pairs of clusters to create a lookup table of heuristic values.

- For the PoV pathfinding graph, demonstrate the "fill" for A* for all three heuristics as follows. To do this for both A* algorithm, pick roughly the same start node and goal node that are "far" from each other (in terms of path length) in your level. Compute the shortest path using the A* algorithm with its heuristic. Then on the level indicate ***graphically*** which nodes are included in the fill for that heuristic (i.e., all nodes that were ever placed in the open list). In addition, for each heuristic, ***render the shortest path computed*** by highlighting graphically the edges between consecutive nodes on the path.
- **Submit screenshots of all three "fills" in your write-up as part of your question solution. This demonstration of the fill is only to show that part R2) is implemented properly; for part R3, just use the A* as implemented in this section minus all the additional graphical rendering of paths and fill (leave the background overlay of the pathfinding graph in place for reference).**

R3) NPC Behaviour:

- If you haven't already, add the following *steering* behaviours to your collection from Assignment 1:
  1. *Path Following* – Implement this (defined in Slides 11-12 of Week 2) to delegate to Arrive (rather than Seek since we want the character to stop at the Goal)
  2. *Looking Where You're Going* (delegating to Align)
- You are to implement a simple game of "inverse tag" on your level with at least three NPCs (fewer for a small level). The decision making by the NPCs playing will be implemented using a finite state machine (this can be implemented simply with if-else, case statements, etc.) and a behaviour tree (the implementation is covered in the lab so the expected implementation is the same as shown in the lab). All the NPCs that are not "it" will be trying to tag the NPC that is "it" (i.e., the inverse of the game tag). The "it" NPC will have higher maximum velocity than the other NPCs.
- The decision making will be done as follows:
  1. The decision making of the "it" NPC will be done with a **finite state machine**. This NPC will be fleeing the other NPCs. It will flee by following a path away from, e.g., the centre of mass of other nearby NPCs. This path can be simply chosen locally by looking at the local pathfinding graph, or by incorporating A* in some way. For more interesting behaviour, consider using tactical pathfinding based on the positioning of the positioning of the other NPCs.
  2. The decision making of the other NPCs will be implemented using a **behaviour tree**. One of these NPCs simply wants to get close enough to the "it" NPC to "tag" it. Since the "it" NPC can move faster, simply chasing it isn't the best choice. Instead some of the NPCs should try to flank the "it" NPC so it is partially trapped by the NPCs. Use pathfinding to position the NPCs around the "it" NPC.
- The decision making has to be done locally by each NPC using whatever perception information you think is reasonable (line of sight, etc.). The behaviour can be enrichened by using some random choices, etc. Special situations such as ending up at the end of a dead-end corridor should be handled by the decision making.

- For all movement, the NPCs are to use the steering behaviour *Path Following* while using *Looking Where You're Going* for orientation. Aim for movement that is "realistic" and the NPCs do not pass through walls or obstacles too frequently while moving. Do this *without using collision detection* but rather with appropriately placed paths and a well-chosen NPC velocities. E.g., you may want to adjust the positions of the points of visibility for the PoV graph, or add more pathfinding nodes in troublesome areas of the graph. If you are using Rigidbody components, make sure the "isKinematic" flag is checked (then collisions will not affect the Rigidbody).
- During the **demonstration** of your program in the lab, show the above behaviour while using A* using the Cluster heuristic, and again while using Navmesh.
- Any parameters not explicitly indicated are up to you. Make choices that make your program visually and behaviourly interesting. E.g., the number of non-"it" NPCs, the relative size of the NPCs to the environment, maximum velocity of NPCs, etc.
- As in Assignment 1, you don't need a fully animated character as long as you can perform affine transformations (translate and rotate) on the character, and can tell which way it is facing. That will be minimally sufficient for this assignment as well.
- **Submit the diagrams of both the FSM and the behaviour tree in your write-up as part of your question solution.**

EVALUATION CRITERIA For Question 3 of this assignment

1. Only working programs will get credit. The marker must be able to run your program if it works to evaluate it, so you must give in your write-up any instructions necessary to get your program running. If your program does not run, we will not debug it.
2. Breakdown:
   - R1: 20% (equally divided among the requirements listed in item R1 above)
   - R2: 30% (equally divided among the requirements listed in item R2 above)
   - R3: 30% (equally divided among the requirements listed in item R3 above)
   - Appropriateness of level design, how interesting are the behaviours (e.g., tactical pathfinding), and general aesthetics : 10%
   - Write-up: 10%

Note on marking: As designed, there can be a wide variation in fulfilling the requirements of this assignment; just barely meeting the requirements will not result in a full mark. For example, if your program only barely has the features listed in R3, do not expect full marks. Behaviours where there is an obvious extra effort made to make the *interaction* between NPCs more realistic/interesting/appealing may expect full or nearly full marks (this is the purpose of the 10% for "how interesting are the behaviours").

What to hand in for Assignment

Questions 1-2: (20%) (Theoretical Questions)
- Submit your answers to questions 1 and 2 on the Moodle course webpage (as a MS Word or PDF file, for example, called **TheoryYourIDnumber.doc** or **TheoryYourIDnumber.pdf**). Submit this file under "Theory Assignment 2".

Question 3 (80%) (Practical Question)
Submission Deliverables (Only Electronic submissions accepted) :

1. Submit a well-commented C# program for Unity including data files, if any, along auxiliary files needed to quickly get your program running, and any other instructions for compiling/ building/running your program. The source codes (and brief write-up, explained below) must be submitted **electronically** in a **zip format** with all the required files (example: **YourIDnumber.zip**). Submit this zip file as "Programming Assignment 2". ***Please do not e-mail the submission.***
2. Demonstrate your working program from an **exe file dated on or before March 2 at 23h55** to the lab instructor in the lab during the period of March 3-5.
3. Included in your zip file will be a brief write-up about your program explaining any special features or implementation details that you would like us to consider during the evaluation.