ARACHNOPHOBIA

Documento di Intelligenza artificiale

Nicholas Aspes 871095

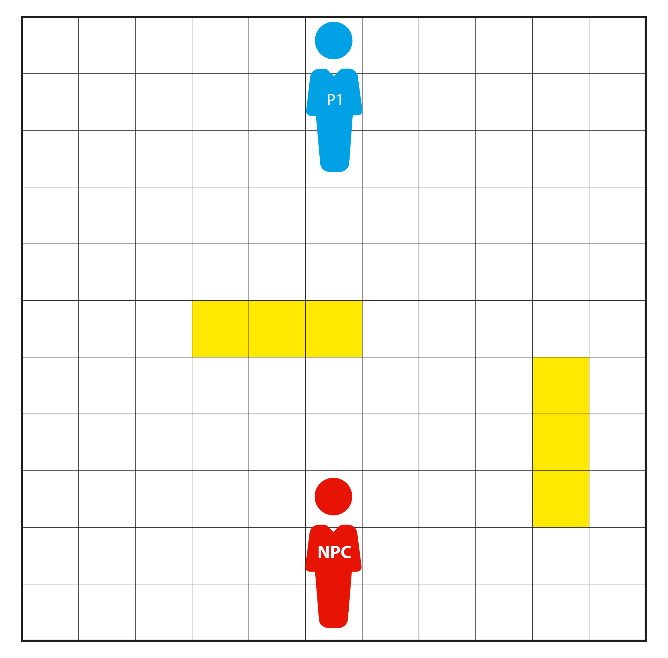
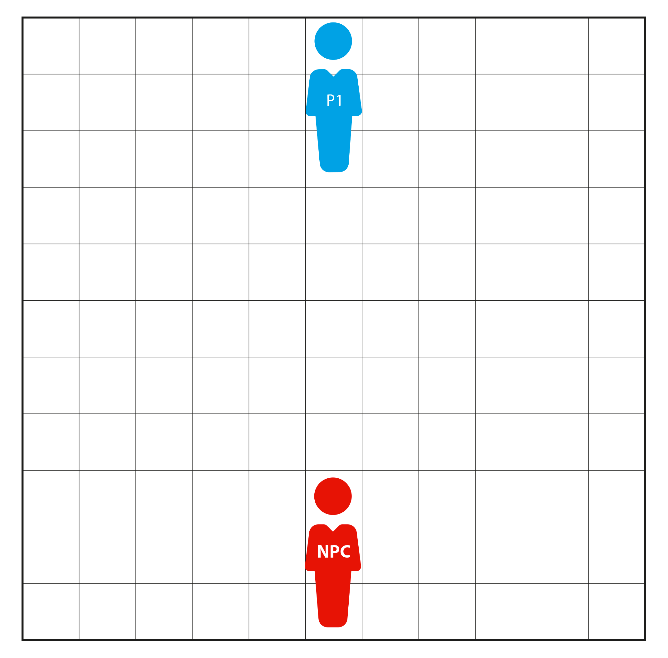
1 – INTRODUZIONE: IL GIOCO

Arachnophobia è un videogioco di tipo survival first person shooter in cui il giocatore deve riuscire a evitare e a tenersi a distanza dal gruppo di NPC avversario, composto da temibili ragni, per il maggior numero di livelli successivi. Ad ogni livello aumenta il tempo in cui il giocatore deve sopravvivere e soprattutto aumenta l’abilità dei ragni nell’inseguire il giocatore. L’unica azione disponibile al giocatore è sparare in direzione dei ragni che, una volta colpiti, generano un muro in fronte a se stessi. In questo modo si creano delle barriere utili per tentare di fuggire. Le barriere hanno però una durata temporale dopo la quale svaniscono. Il giocatore è dotato di una pistola a ripetizione con un numero limitato di colpi, che si rigenerano ogni secondo fino al massimo preimpostato.

2 - STRUTTURA MAPPA

Il gioco si svolge su una mappa logicamente suddivisa in quadrati e rappresentata da una matrice di oggetti di tipo Node. Ogni Node può avere il valore “wall” = true oppure false. Nel primo caso rappresentano un ostacolo e non sono percorribili.

La matrice è costruita prendendo il mondo di gioco e suddividendolo per il numero di quadrati impostato. Dopodiché l’algoritmo scorre su tutti i quadrati e attraverso uno SphereCast controlla che non siano presenti elementi con Mask settata a “Wall”, generando un elemento di tipo Node da aggiungere alla matrice: con valore wall = true se rileva la presenza di un elemento Wall oppure wall = false altrimenti.



(matrice all’inizio del gioco) (matrice con 6 Node con wall = true)

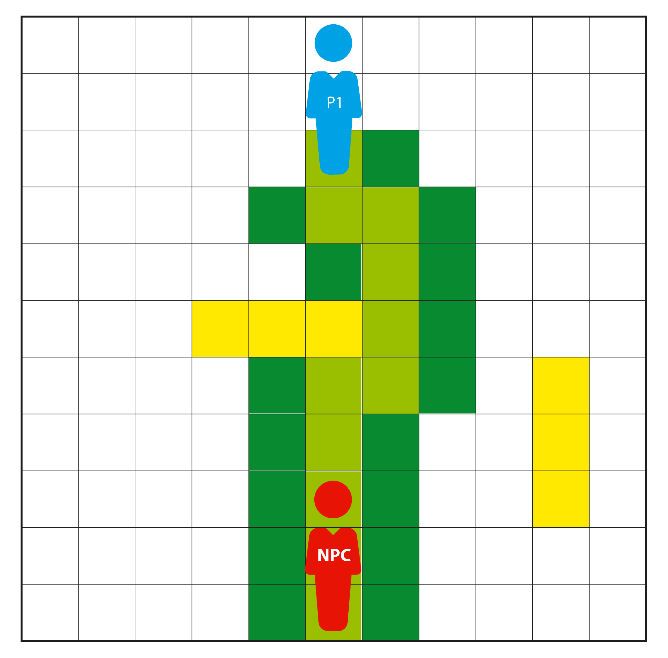
3 – ALGORITMO PATHFINDING BASE

L’algoritmo di pathfinding implementato è una evoluzione dell’algoritmo A\* modificato per sfruttare le caratteristiche del progetto.

Viene indicato con NPC il soggetto che effettua il pathfinding, con target il soggetto ricercato, con startNode il nodo della matrice di Node che rappresenta il la mappa di gioco e in cui si trova l’NPC e con goalNode il nodo che allo stesso modo rappresenta la posizione del target nella matrice.

L’algoritmo A\* prende la posizione nel mondo reale di NPC e di target e ne ricava i corrispettivi nodi dalla matrice dei nodi: startNode e goalNode. Vengono considerate due liste di nodi: i nodi candidati, openList, e i nodi già visitati, closedList. Viene aggiunto il nodo di partenza, startNode, all’openList e comincia la ricerca del path:

1. Viene selezionato da openList il nodo a costo minore. Ogni Node ha 3 parametri di costo: gCost, rappresenta il costo dal nodo di partenza al nodo in esame, hCost, rappresenta il costo stimato dal nodo corrente fino alla destinazione, e fCost, rappresenta la somma di gCost e hCost. Il nodo selezionato viene salvato nella variabile currentNode e viene rimosso da openList e aggiunto a closedList.
2. Se currentNode è uguale a goalNode, l’algoritmo termina. Viene creata una lista finalPath di nodi partendo da currentNode e aggiungendo tutti i nodi parent, fino a raggiungere il nodo di partenza. La lista viene quindi restituita all’NPC che, dirigendosi verso i nodi indicati in finalPath, raggiunge il target.
3. Altrimenti si esplorano i nodi vicini a currentNode. Per ogni vicino verifica di non averlo già esplorato, controllando la non presenza in closedList, e che non sia un nodo rappresentante un Wall, cioè che abbia valore wall = false. Per ciascuno stima gCost e hCost e setta currentNode come parent del nodo. Il nodo vicino viene quindi aggiunto a openList. Si ricomincia quindi dal punto 1.



(algoritmo A\* base)

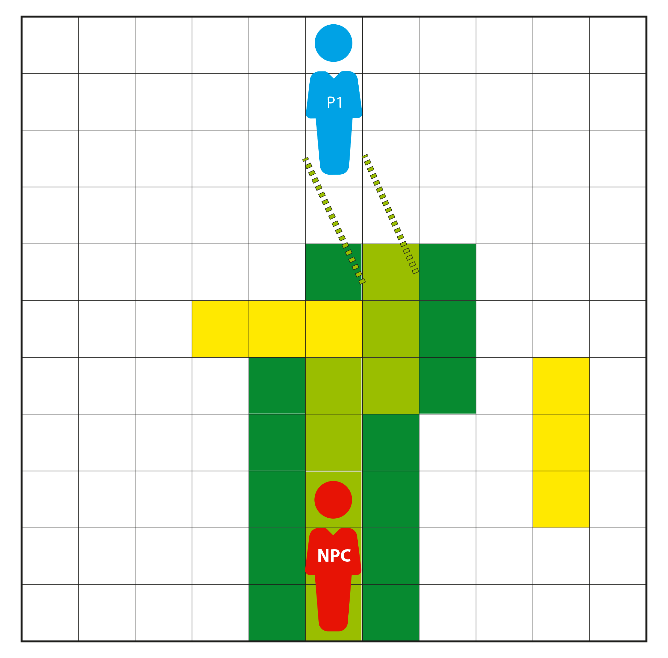
L’algoritmo viene ripetuto ad ogni Update dell’NPC per seguire gli spostamenti del Player e per adattarsi alle modifiche della mappa.

3.1 MODIFICHE GREEDY

Considerando le caratteristiche del progetto l’algoritmo ha subito le seguenti modifiche.

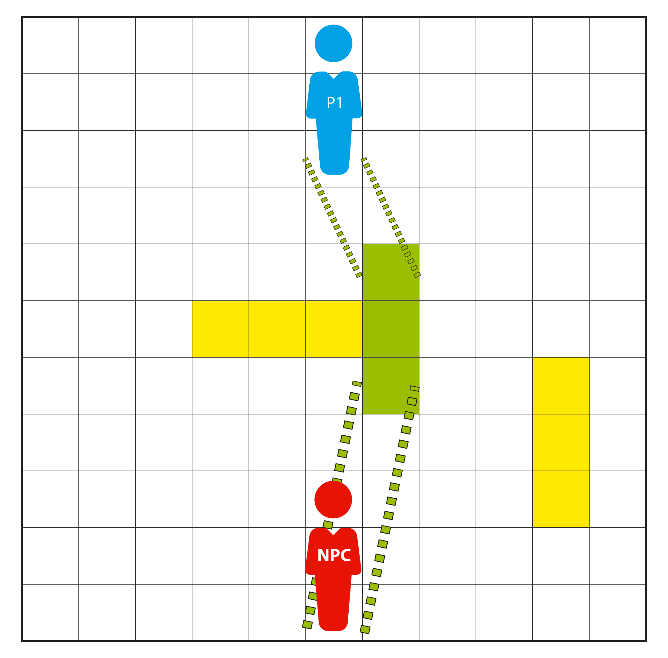
Lo stato iniziale della griglia non presenta nessun nodo con valore wall = true. Prima di avviare l’algoritmo di ricerca, utilizzando due Linecast che partono dai lati del nodo di partenza e arrivano ai lati del nodo target, viene verificata la presenza di ostacoli tra NPC e target corrente. Gli ostacoli possono essere oggetti di tipo Wall o NPC. Se non ci sono ostacoli, allora viene settata come destinazione direttamente la posizione del target.

Quando è rilevata la presenza di ostacoli, viene chiamato l’algoritmo di pathfinding ma cercando di interromperlo, sfruttando i Linecast sopradescritti, per ottenere una soluzione greedy. Prima di arrivare al punto 2 dell’algoritmo, si effettua un controllo da currentNode verso il goalNode e, se non si verifica la presenza di ostacoli, l’algoritmo è interrotto, impostando currentNode come parent diretto di goalNode. Chiamo quanto descritto come punto 1.1 dell’algoritmo. In questo modo la lista finalPath non conterrà i nodi da percorrere da startNode fino a goalNode ma conterrà i nodi a partire da startNode fino al primo nodo da cui goalNode è visibile.



(interruzione algoritmo quando Player visibile)

Si applica poi lo stesso ragionamento all’interno del punto 2: mentre l’algoritmo ripercorre i nodi in ordine inverso, seguendo il parent di ciascuno, utilizza i Linecast per controllare la presenza di ostacoli tra currentNode e startNode. Se non rileva la presenza di ostacoli interrompe la creazione della lista finalPath. In questo modo il primo nodo all’interno della lista sarà il primo nodo che raggiungibile direttamente e che permette di superare il primo ostacolo presente nel path.

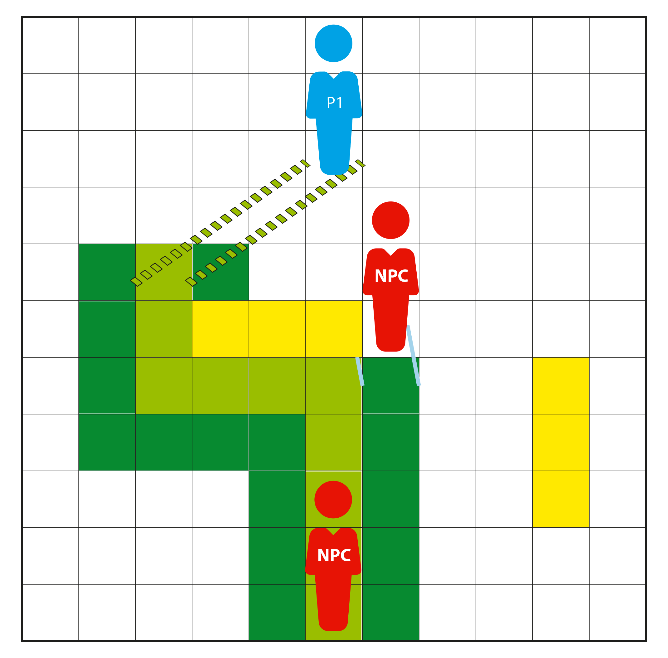


(interruzione generazione finalPath quando NPC visibile)

3.2 MODIFICHE AL COSTO DEI CAMMINI

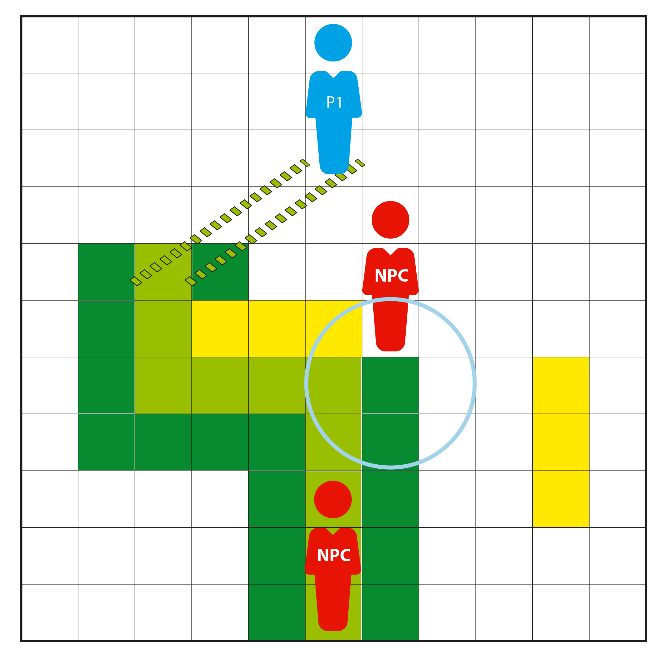
Come detto in precedenza, anche gli altri NPC vengono considerati come ostacoli. Più precisamente, viene utilizzato l’incontro con altri NPC per aumentare il costo di un cammino, in modo tale da guidare il punto 1 dell’algoritmo verso la selezione di un nodo che porti alla generazione di un finalPath diverso da quello in cui sono presenti altri NPC. Il cammino, tuttavia, non viene del tutto bloccato in quanto potrebbe essere l’unico possibile per raggiungere il target.

La modifica al costo di un cammino avviene in due momenti: il primo è durante il punto 1.1, dove utilizzo i Linecast per controllare se il cammino è libero da ostacoli dal currentNode al goalNode. In questo caso infatti i Linecast possono rilevare la presenza di altri NPC e in questo caso viene aumentato il costo del cammino durante l’esplorazione dei vicini di currentNode.



(modifica pathfinding quando incontro altri NPC)

Subito dopo il punto 1.1 si trova un nuovo punto, 1.2, in cui attraverso uno SphereCast controllo se sono presenti altri NPC per un certo raggio intorno a currentNode e, in caso di riscontro, aumento il costo del cammino.



(modifica pathfinding quando rilevo altri NPC nelle vicinanze)

3.3 RISULTATI

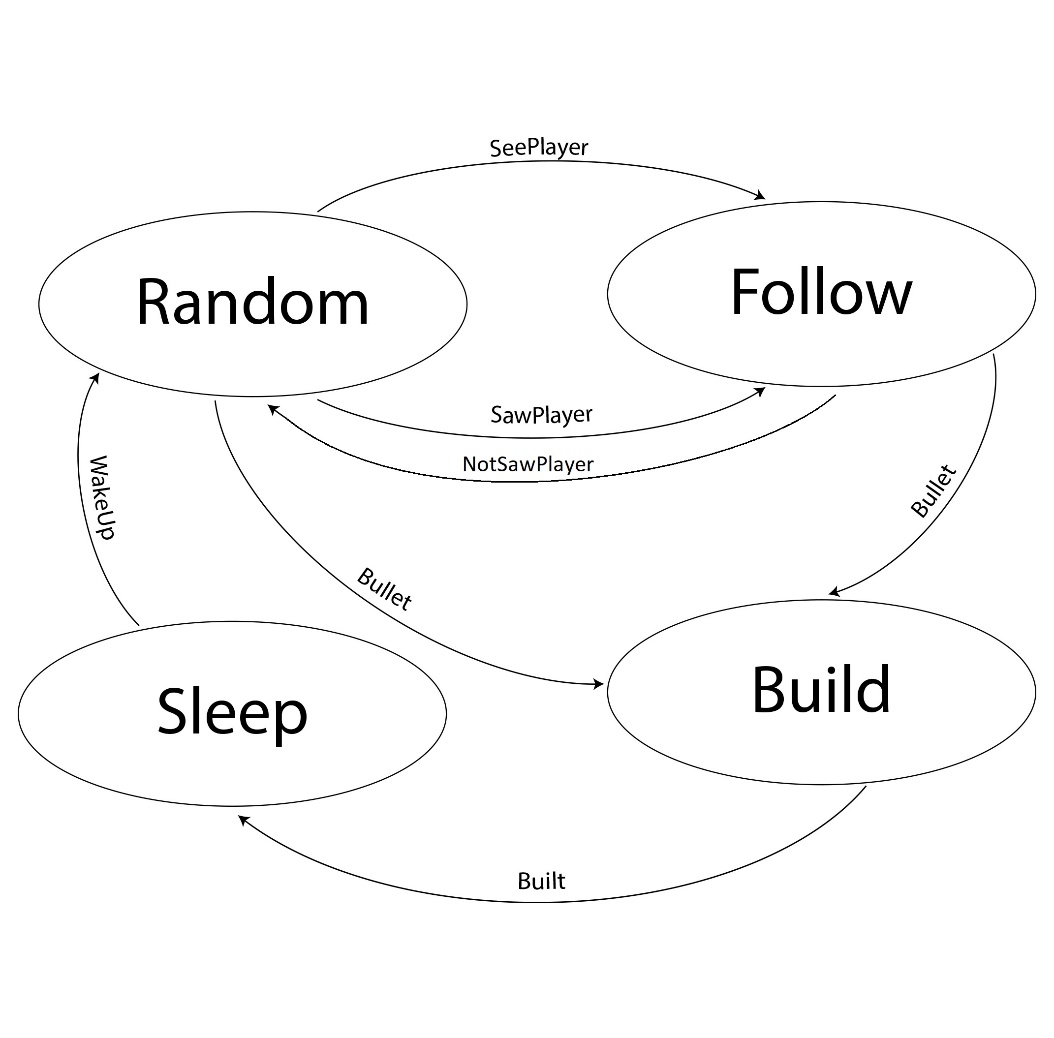
Le modifiche all’algoritmo A\* base hanno portato a un miglioramento delle prestazioni generali del progetto, dovuto principalmente all’interruzione anticipata dell’algoritmo quando non ci sono più ostacoli verso il target e al non utilizzo totale dell’algoritmo quando non ci sono ostacoli tra NPC e target.

Inoltre, utilizzando come destinazione degli NPC la prima posizione che permette di superare un ostacolo, si è ottenuta una maggior naturalezza nel movimenti degli NPC, che si dirigono direttamente nel punto migliore per arrivare all’obiettivo. L’utilizzo di due Linecast ai lati degli NPC inoltre permette in automatico di considerare lo spazio necessario agli stessi per passare lungo il percorso ed evitare lo scontro con gli ostacoli.

4 - MACCHINA A STATI

Per descrivere il comportamento degli NPC è stata implementata una Finite State Machine: ogni NPC si trova in uno stato che ne descrive le azioni e il comportamento che manterrà finché si trova in quel determinato stato. Ogni stato è connesso a un altro attraverso le transizioni. Quando le condizioni per una transizione si verificano, lo stato cambia e si cominciano a eseguire le azioni e il comportamento del nuovo stato.

Gli NPC possono trovarsi in 4 stati: “Random”, “Follow”, “Build” e “Sleep”.



4.1 - RANDOM

Lo stato iniziale di ogni NPC è lo stato Random. In questo stato l’NPC non è a conoscenza della posizione del Player e quindi sceglie casualmente come target un nodo della matrice con wall = false. Inizia quindi la ricerca, attraverso l’algoritmo di pathfinding, verso questa posizione. In questo stato, l’NPC tiene in memoria due variabili “temporali”: randomTimeSaved, che rappresenta l’istante il target viene selezionato, e randomTimeTravel, che rappresenta il numero di secondi dopo il quale un nuovo target verrà scelto. randomTimeTravel rappresenta nella pratica il tempo che l’NPC ha a disposizione per raggiungere il target scelto. Se il punto viene raggiunto prima dello scadere del tempo, l’NPC rimane fermo in attesa e, solo nel momento della scadenza, seleziona un nuovo target. randomTimeTravel viene ridefinito ogni volta che viene scelto un nuovo target con un valore scelto randomicamente, in questo modo si desincronizzano gli NPC che sceglieranno ciascuno il proprio target in tempi differenti.

Dallo stato Random si può passare allo stato Follow attraverso due condizioni: la prima è quando mi avvicino a una certa distanza da Player e non ci sono ostacoli tra NPC e Player, SeePlayer.

Quando questa condizione si verifica la prima operazione che l’NPC effettua è quella di creare un Report all’interno di SharedKnowledge. La SharedKnowledge sono le informazioni sulla posizione di Player condivise da tutti gli NPC. Nella pratica contiene un insieme di Report. Ciascun Report consiste nel DateTime dell’avvistamento di Player, la posizione di Player e la posizione dell’NPC che ha creato il report. I report hanno una durata di validità dopo la quale vengono scartati.

La seconda condizione attraverso la quale è possibile passare dallo stato Random a quello Follow è SawPlayer: prendo da SharedKnowledge tutti i Report validi e per ciascuno valuto se la distanza tra me e l’NPC che ha creato la segnalazione è inferiore a un determinato raggio. Se la condizione viene accettata significa che un NPC vicino ha visto il giocatore e utilizzo la sua informazione per inseguire il Player. In questo modo viene simulato l’avviso di un NPC verso gli NPC vicini di avvistamento di Player.

Dallo stato Random è possibile passare direttamente allo stato Build quando l’NPC viene colpito da un proiettile, Bullet.

4.2 - FOLLOW

Mentre si trova in questo stato, l’NPC cerca di inseguire Player nella mappa. Prendo quindi da SharedKnowledge il report valido più recente tra quelli generati dagli NPC vicini (o da se stesso) e imposto la posizione di Player come target.

Ad ogni passo inoltre chiamo SeePlayer per controllare se vedo Player, in questo caso creo un nuovo Report da aggiungere in SharedKnoledge (se già presente un Report dello stesso NPC viene sostituito).

Dallo stato Follow posso cambiare stato in due condizioni: torno allo stato Random quando non ci sono più Report validi in SharedKnowledge generati da NPC vicini, NotSawPlayer, oppure passo allo stato Build quando vengo colpito da un proiettile, Bullet.

4.3 - BUILD

In questo stato viene effettuata una singola azione non ripetuta.

Si considera la posizione dell’NPC e si divide lo spazio di fronte a questo in 3 parti: left, center e right. Per ciascuna parte si utilizza uno SphereCast: viene generata una colonna se non viene rilevata la presenza in quello spazio di un NPC o di Player. Genero in totale un massimo di 3 colonne che vanno a formare un muro.

Successivamente aggiorno la matrice per identificare i Node con wall = true. I muri inoltre hanno un tempo di vita dopo il quale vengono distrutti ed è quindi necessario aggiornare la matrice anche quando si verifica questo evento.

Al termine di questa azione si attende un periodo di tempo dopo il quale si passa automaticamente allo stato Sleep.

4.4 - SLEEP

Nello stato Sleep, l’NPC rimane addormentato per un numero di secondi per poi tornare allo stato Random.