UNIVERSITATEA TEHNICĂ „Gheorghe Asachi” din IAȘI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DOMENIUL: Calculatoare
SPECIALIZAREA: Sisteme Distribuite și Tehnologii Web

Simulator de evacuare dintr-o clădire în situații de urgență

Coordonator științific
Prof. Dr. Mitică Craus

Absolvent
Ungureanu Ionuț-Leonaș

Iași, 2022

DECLARAȚIE DE ASUMARE A AUTENTICITĂȚII
LUCRĂRII DE DIPLOMĂ

Subsemnatul(a) UNGUREANU IONUȚ-LEONAȘ,
legitimat(ă) cu CI seria VS nr. 911350, CNP 1970218373466
autorul lucrării SIMULATOR DE EVACUARE DINTR-O CLĂDIRE ÎN SITUAȚII DE URGENȚĂ

elaborată în vederea susținerii examenului de finalizare a studiilor de licență organizat de către Facultatea de Automatică și Calculatoare din cadrul Universității Tehnice „Gheorghe Asachi" din Iași, sesiunea IUNIE a anului universitar 2022, luând în considerare conținutul Art. 34 din Codul de etică universitară al Universității Tehnice „Gheorghe Asachi" din Iași (Manualul Procedurilor, UTI.POM.02 – Funcționarea Comisiei de etică universitară), declar pe proprie răspundere, că această lucrare este rezultatul propriei activități intelectuale, nu conține porțiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislației române (legea 8/1996) și a convențiilor internaționale privind drepturile de autor.

Data
29.06.2022

Semnătura

Cuprins

Simulator de evacuare dintr-o clădire în situații de urgență

Ionuț-Leonaș Ungureanu

Rezumat

Trăim într-o vreme în care suntem înconjurați, și asistați, de tehnologie, la tot pasul. Fie că vorbim de smartphone-uri, laptop-uri, roboți de aspirare, mașini, toate pun la dispoziția utilizatorului tehnologie software de ultimă generație. Cu trecerea timpului se observă o utilizare mai accentuată a inteligenței artificiale, în diverse domenii: industria auto, platforme e-comerce, smartphone-uri, securitate, roboți casnici, și așa mai departe.

Având în vedere acestea, se propune realizarea unui simulator de evacuare în caz de dezastre. În cazul particular al acestui proiect s-a decis ca dezastrul abordat să fie reprezentat de realitatea și posibilitatea unui incendiu. Incendiul propriu-zis este mai puțin important, vorbind strict în contextul acestui proiect. Pe ce se dorește a se axa, este evacuarea. Vorbim despre evacuarea indivizilor din clădire, cu accent pe pozițiile în care sunt surprinși de către incendiu, distanța parcursă și timpul pentru a ajunge la o anumită destinație în care pot fi considerați în siguranță, și nu în ultimul rând, starea în care se află indivizii la finalul simulării.

Pentru realizarea acestui simulator, s-ar putea calcula cea mai scurtă distanță și s-ar urma dintr-un punct de start până într-un punct final, acesta din urmă reprezentând o zonă de siguranță. Problema unei astfel de abordări, în cazul acesta, cât și în cazul altor simulări ce presupun atingerea unei destinații într-un context relativ real, este faptul că reduc probabilitatea, sau cauzalitatea, la o constantă. Prin acest lucru, se face referire la faptul că, rezultatele vor fi constante, indivizii urmând aceeași cale pentru punctul de start, și singura variabilă ar fi randomizarea punctelor de start a incendiului.

Pentru introducerea unei variabile în plus, se propune utilizarea învățării cu întărire în speranța că această practică ar introduce un anumit procent de haos, similar contextului realității, care s-ar reflecta în rezultate. S-a ales algoritmul Q, pentru a învăța indivizii să ajungă la destinație, deci să evacueze clădirea, pe baza recompenselor în urma antrenării într-un sistem complex, și anume clădirea care va fi folosită ca referință în acest proiect. Datorită spațiului complex al simulării, algoritmul Q nu este de ajuns, s-au mai exact, nu este alegerea potrivită. Pentru a fi alegerea potrivită, pentru spațiul vast al simulării, se utilizează Deep Q-Learning pentru a putea învăța individul să ia deciziile corecte într-un spațiu enorm, constituit din nenumărate posibile stări.

Sistemul este construit în Unity 3D, clădirea supusă simulării având șase etaje, 3 uși de ieșire, o scară principală care conectează toate etajele, și altele 2 mai mici, intermediare. Clădirea fiind de anvergură mare, cu multiple camere și obstacole, face dificilă evacuarea numai cu ajutorul unei rețele neuronale simple. În acest context, se introduce algoritmul A* pentru găsirea unui căi de ieșire, nu neapărat cea mai scurtă, care v-a fi urmată de către individ, a căror decizii vor fi luate de către rețeaua neuronală.

Introducere

În prezent, în numeroase domenii, există o multitudine de aplicații sau sisteme, atât software, cât și hardware sau fizice, care se ocupă cu studiul unor anumite situații, bine definite, pentru a realiza predicții, a detecta cazuri/stări cheie, care ori pot fi fructificate, ori evitate, în funcție de necesitate. Printre acestea, ne amintim procesul de data mining, care determină pattern-uri frecvente cu scopul de a putea face predicții și a lua acțiuni concrete în fructificarea acestora – de exemplu ne imaginăm problema coșului de cumpărături: în urma predicțiilor se grupează produsele care sunt cumpărate frecvent împreună pentru creșterea câștigului și satisfacerea nevoii clientului. Totodată, luăm în considerare și testele Euro NCAP în cadrul căruia mașinile sunt testate pentru o multitudine de situații pentru a stabili siguranța oferită de acestea și punctele cheie care nu îndeplinesc, sau care îndeplinesc, cerințele de siguranță. Acestea sunt câteva exemple bine cunoscute de sisteme de test, simulare, care oferă diferite perspective asupra obiectului analizat.

Lucrarea, în consecință, își propune realizarea unui sistem de simulare, care supune o clădire, de dimensiuni relativ medii, unei situații dezastruoase, care din nefericire este posibilă, și anume eventualitatea unui incendiu. În ziua de astăzi, sistemele electrice, calculatoarele, sunt pretutindeni, și totodată și posibilitatea declanșării unui incendiu. În astfel de circumstanțe, este de dorit, a ști eventuale probleme care pot apărea în timpul unei evacuări, deoarece informația este putere și ne poate oferi șansa luării de măsuri pentru evitarea situațiilor nedorite, sau îmbunătățirea acestora.

Proiectul practic, surprinde un sistem construit în Unity 3D a căror principale componente sunt:

- o clădire de dimensiuni medii, cu șase etaje;
- flăcările, realizate cu sistemul de particule din Unity;
- punctele de siguranță, care se regăsesc în afara clădirii;
- indivizii.

Modul abordat, în gestionarea acestor componente, este explicat în detaliu, în capitolele următoare.

Capitolul **Fundamente teoretice**, va aborda cunoștințele teoretice care au necesitat în construcția sistemului și funcționarea corectă, cât și a tehnologiilor necesare puse în practică.

Proiectarea, va pune în perspectivă modalitatea de funcționare a componentelor simulării între ele, cât și la nivel individual. Detaliile de finețe vor fi prezentate într-o formă ușoară de înțeles pentru a putea prezenta viziunea autorului legat de componentele puse în contrast.

Implementarea, va prezenta modul în care au fost construite componentele software necesare punerii în funcțiune a sistemului de simulare. Va fi prezentat cum componentele din Unity 3D sunt coordonate de obiecte construite în C# .Net, natura și necesitatea acestor obiecte, și elemente de inteligență artificială necesare în construirea sistemului de simulare. Cel mai important aspect este faptul că indivizii sunt coordonați de o rețea neuronală, antrenată cu ajutorul algoritmului Q, cunoscută sub numele de Deep Q Network.

Capitolul de **testare** va sublinia modul în care a fost antrenată rețeaua neuronală, aspecte de funcționare, limitări și eventuale probleme care pot apărtea.

Capitolul de **concluzii**, pe lângă ideile observate în urma realizării acestui proiect, va îngloba și o secțiune de discuții menită să ofere idei de viitor care ar putea să reducă sau sa elimine unele limitări.

Capitolul 1. Fundamentul teoretic

1.1. Algoritmul Q-Learning

Conform [1], învățarea cu întărire este o subclasă de algoritmi ai inteligenței artificiale, care se ocupă cu dezvoltarea de agenți inteligenți care acționează într-un mediu cu scopul de a maximiza un reward cumulativ. Învățarea cu întărire implică un agent, un set de stări și un set de acțiuni. O acțiune **a**, duce agentul din starea **s1** în starea **s2**, în urma acesteia primind o recompensă, care în funcție de valoarea ei poate fi considerată și pedeapsă, dacă este o valoare negativă.

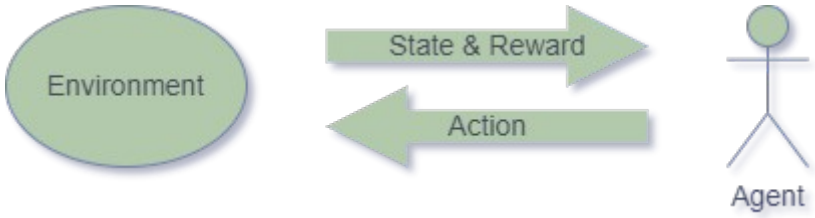

Figura 1.1. Modelul de învățare cu întărire.

În Figura 1.1, se observă faptul că algoritmul realizează următorii pași:

1. Agentul observă starea mediului;
2. Ia o acțiune în domeniul de acțiuni al mediului;
3. Primește o recompensă pe baza acțiunii;
4. Învață din experiență pentru viitoarele decizii;
5. Repetă pașii până la găsirea unei strategii optime.

Q-Learning este unul din cei mai simpli algoritmi de învățare cu întărire. Scopul acestuia este ca agentul să ia cele mai bune decizii în funcție de starea în care se află, astfel încât la final, recompensa totală să fie cât mai mare. Q vine de la cuvântul quality din engleză, care înseamnă calitate. Practic algoritmul determină cât de calitativă este acțiunea luată de agent într-o anumită stare, pentru maximizarea recompensei totale. Algoritmul folosește un tabel, numit Q-Table, care stochează valorile Q pentru toate acțiunile pentru fiecare stare, ca în Figura 1.2.

Action	↑	↓	→	←
Start	0	0	1	0
Idle	0	0	0	0
Correct Path	0	50	22	0
Wrong Path	15	0	18	0
End	0	0	1	0

Figura 1.2. Q-Table.

Acesta este inițializat cu 0, și este actualizat la fiecare pas, cu ajutorul ecuației Bellman prezentată în Figura 1.3. Practic, la fiecare pas, valoarea Q pentru starea **s** și acțiunea **a**, este actualizată conform ecuației, bazându-se pe recompensa primită în urma acțiunii care a dus agentul în următoarea stare.

$$Q(state, action) \leftarrow (1-\alpha)Q(state, action) + \alpha\left(reward + \gamma \max_a Q(next\ state, all\ actions)\right)$$

Figura 1.3. Ecuația Bellman.

Termenii ecuației Bellman, sunt:

- Q(state, action) – valoarea q a intersecției dintre starea „state” și acțiunea „action”. Parctic, pentru Figura 1.2, pentru starea „Start” și acțiunea „dreapta”, valoarea q este 1.
- α (alpha) – este rata de învățare.
- Reward (recompensa) – este recompensa pe care agentul o primește pentru o acțiune luată plecând dintr-o anumită stare.
- γ (gamma) – factorul de discount, folosit pentru a regla importanța recompensei, pe termen lung sau scurt.
- Max Q(next state, all actions) – este cea mai mare valoare q, dintre toate valorile, pentru următoarea stare.

1.1.1. Rata de învățare

Rata de învățare determină cât de mult informația nou acumulată influențează informația veche, sau experiența câștigată la pașii anteriori. O valoare a ratei de învățare egală cu 0, practic va inhiba capacitatea de învățare a agentului, iar o valoarea egală cu 1 va face agentul să considere doar cea mai recentă informație acumulată.

1.1.2. Factorul de discount

Factorul de discount determină importanța recompensei primite. O valoare egală cu 0 va influența agentul să considere cea mai recentă recompensă, iar o valoare egală cu 1 va influența agentul să considere recompense pe termen lung.

1.2. Politica Epsilon-greedy

Politica epsilon-greedy stabilește echilibrul între explorare și exploatare pentru un agent care învață folosind un algoritm de învățare cu întărire.

Explorarea permite unui agent să își îmbunătățească cunoștințele despre fiecare acțiune luată într-o anumită stare prin expunerea agentului unei anumite situații, indiferent de cunoștințele anterioare, deci practic fără neapărat voia acestuia.

Exploatarea reprezintă utilizarea cunoștințelor anterioare pentru a decide ce acțiune să fie luată de către agent în starea curentă. Dacă agentul doar exploatează, atunci este o posibilitate mare ca acesta să învețe să se descurce doar în anumite situații, care poate nu sunt neapărat cele mai bune, ducând astfel la un comportament per total nu tocmai bun pentru maximizarea recompensei totale.

Mai exact, politica epsilon-greedy, stabilește posibilitatea explorării de către agent, alegând între exploatare și explorare într-o metodă aleatorie, conform Figura 1.4.

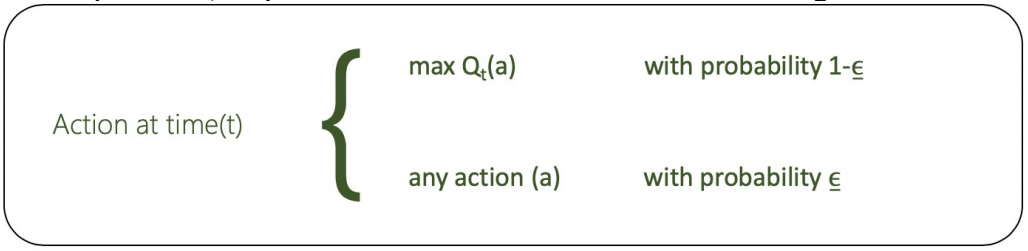

Figura 1.4. Politica Epsilon-greedy [0]

*1.3. Algoritmul A**

A* este un algoritm de găsire a unei căi, dintr-un punct de start într-un punct final, care este frecvent utilizat în informatică datorită robusteței și a eficienței optime, după [2]. O deficiență a algoritmului este faptul că dispune de o complexitate al spațiului mare, deoarece reține toate nodurile generate în memorie.

Algoritmul poate fi văzut ca o extensie a algoritmului lui Dijkstra, cu mențiunea că A* nu găsește cele mai scurte căi între nodul de start și toate celelalte noduri, ci găsește o cale între nodul de start și un nod final specificat. De ce o cale și nu cea mai scurtă cale? Asta se datorează euristicii pe care se bazează algoritmul, care în funcție de cum este configurată va găsi cea mai scurtă cale sau o cale influențată de euristică. Este folosită o politică „primul cel mai bun” (Best-First), acest lucru fiind realizat prin definirea funcției de evaluare:

$$f(n)=g(n)+h(n), n \in Noduri \quad (1)$$

Unde:

- f(n) – costul estimat al celei mai bune soluții care trece prin n;
- g(n) – costul căii din nodul de start la n;
- h(n) – o euristică prin care se estimează cel mai ieftin cost de la n la punctul final.

După cum este menționat și în [3], A* evaluează nodurile combinând distanța deja parcursă până la nod cu distanța estimată până la nodul final. Acesta, întoarce mereu soluția optimă dacă o soluție există atât timp cât distanța până la soluție nu este supraestimată. Dacă euristica h(n) nu este admisibilă, o soluție tot va fi găsită, cu mențiunea că optimalitatea nu este garantată.

Algoritmul urmează pașii în felul următor: inițial se introduce în mulțimea „open”, care este organizată ca o coadă de priorități după f(n), nodul de start, corespunzător stării inițiale. La fiecare pas se extrage din mulțimea „open” nodul cu f(n) minim. Dacă nodul extras este chiar nodul scop, nodul final, atunci se întoarce calea de la nodul de start la cel final. Altfel, dacă nodul nu a fost explorat deja se expandează. Pentru fiecare nod vecin nou găsit, dacă nu există deja în „open” sau în „closed”, se introduce în „open”. Dacă nodul deja există în cele două mulțimi, se verifică dacă nodul curent produce o cale mai scurtă. Dacă acest lucru este valabil, atunci se setează nodul curent ca părinte al nodului vecin și se corectează costul g. Această corectare implică reevaluarea tuturor căilor care trec prin nodul vecin, deci acesta va trebui reintrodus în mulțimea „open”.

Pseudocodul pentru A* este următorul:

```
function reconstruct_path(cameFrom, current)
    total_path := {current}
    while current in cameFrom.Keys:
        current := cameFrom[current]
        total_path.prepend(current)
    return total_path

// A* finds a path from start to goal.
// h is the heuristic function. h(n) estimatesthe cost to reach the goal from node
n.
function A_Star(start, goal, h)
    // The set of discovered nodes that may need to be (re-)expanded.
    // Initially, only the start node is known.
    // This is usually implemented as a min-heap or priority queue rather than a
```

```
hash-set.
    openSet := {start}

    // For node n, cameFrom[n] is the node immediately preceding it on the
cheapest path from start
    // to n currently known.
    cameFrom := an empty map

    // For node n, gScore[n] is the cost of the cheapest path from start to n
currently known.
    gScore := map with default value of Infinity
    gScore[start] := 0

    // For node n, fScore[n] := gScore[n] + h(n). fScore[n] represents our
current best guess as to
    // how cheap a path could be from start to finish if it goes through n.
    fScore := map with default value of Infinity
    fScore[start] := h(start)

    while openSet is not empty
        // This operation can occur in O(Log(N)) time if openSet is a min-heap or
a priority queue
        current := the node in openSet having the lowest fScore[] value
        if current = goal
            return reconstruct_path(cameFrom, current)

        openSet.Remove(current)
        for each neighbor of current
            // d(current,neighbor) is the weight of the edge from current to
neighbor
            // tentative_gScore is the distance from start to the neighbor through
current
            tentative_gScore := gScore[current] + d(current, neighbor)
            if tentative_gScore < gScore[neighbor]
                // This path to neighbor is better than any previous one. Record
it!
                cameFrom[neighbor] := current
                gScore[neighbor] := tentative_gScore
                fScore[neighbor] := tentative_gScore + h(neighbor)
                if neighbor not in openSet
                    openSet.add(neighbor)

    // Open set is empty but goal was never reached
    return failure
```

Pentru o mai bună eficiență, se poate utiliza pentru:

- open – o structură de tip heap;
- closed – un hash set.

Algoritmul va întoarce calea optimă către soluție, dacă o soluție există. Inconvenientul care se observă, este necesitatea reevaluării nodurilor din closed.

1.4. Rețele neuronale

1.4.1. Neuronul

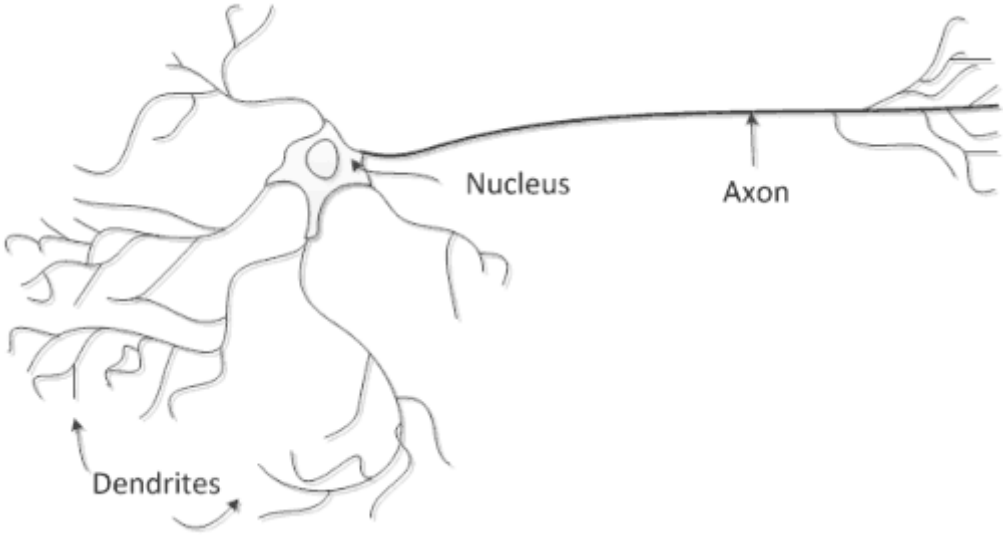

Figura 1.5. Neuron biologic.

În biologie, neuronul este o celulă adaptată la recepționarea și transmiterea informației [4]. Un număr suficient de mare de astfel de celule pot crea o structură deosebit de complexă, drept exemplu fiind creierul uman. Rețelele neuronale artificiale se bazează pe această idee, imitând neuronul biologic și creând neuronul artificial. Cu ajutorul neuronului artificial se creează structuri mai complexe, dar nu la fel de complexe ca rețelele neuronale biologice, care oferă rezultate remarcabile.

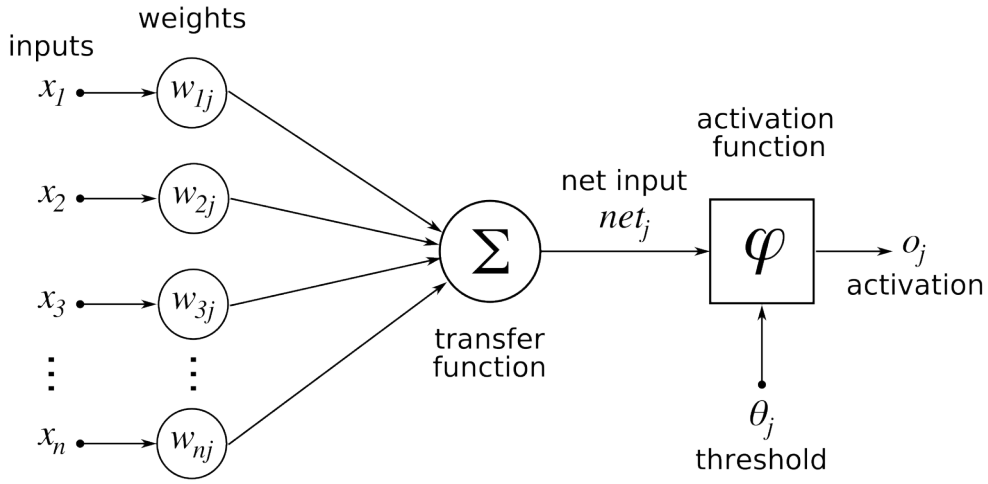

Figura 1.6. Neuron artificial.

Asocierea între neuronul biologic, Figura 1.5, și cel artificial, Figura 1.6, se realizează în următorul mod:

- Sinapsele pe care neuronul biologic le făcea prin dendrite devin intrările neuronului artificial;
- Corpul neuronului devine un sumator plus o funcție de activare (transfer);
- Axonul devine ieșirea neuronului artificial;
- Bias-ul este o intrare care surprinde caracteristicile neliniare simple, cu alte cuvinte biasul determină subiectivitatea neuronului.

1.4.2. Definirea unei rețele neuronale

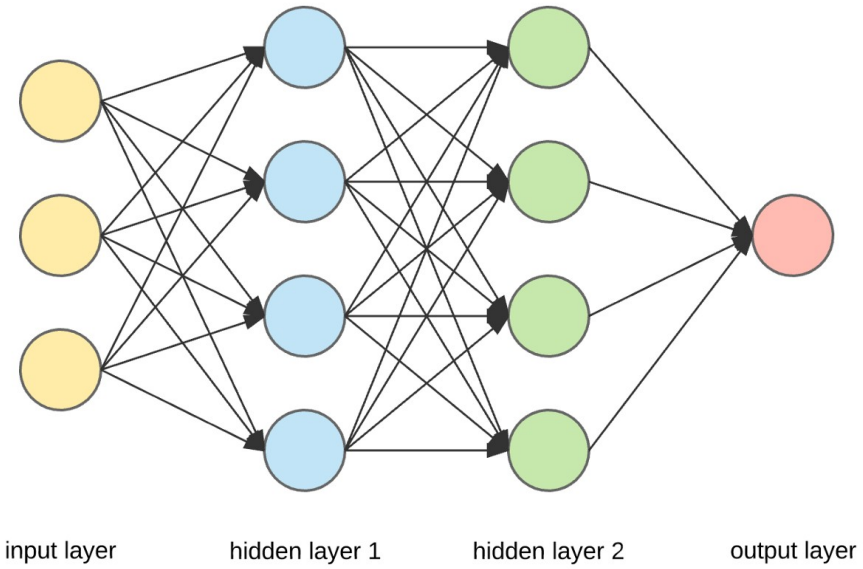

Figura 1.7. Exemplu de rețea neuronală.

O rețea neuronală este un sistem format dintr-un număr de elemente interconectate, neuroni, ce conlucrează pentru rezolvarea unei probleme. Cu alte cuvinte, rețelele neuronale artificiale caracterizează ansambluri de elemente de procesare simple, puternic interconectate și posibil operând în paralel, care urmăresc să interacționeze cu mediul înconjurător într-un mod similar al creierelor biologice, și care prezintă capacitatea de a învăța [5]. Deși se aseamănă în funcționare cu creierul uman, rețelele neuronale au o structură diferită de cea a creierului, fiind mult mai simplă decât corespondentul său uman, dar similar, este compusă din unități puternice cu capacitate de calcul, inferioare însă neuronului uman.

Deosebirea dintre rețelele neuronale și alte sisteme de prelucrare a informațiilor, este capacitatea de învățare în urma interacțiunii cu mediul înconjurător și îmbunătățirea performanțelor. Mediul înconjurător este perceput prin informațiile venite prin intermediul intrărilor neuronilor.

1.4.3. Tipuri

După tipul conexiunilor făcute de neuroni, avem:

- Monolayer – rețele formate doar din start de intrări și strat de ieșiri;
- Multilayer – rețele formate din strat de intrări, start de ieșiri, și unul sau mai multe straturi ascunse.

După direcția semnalului, avem:

- Rețele feed-forward – rețele multilayer în care informația este transferată și prelucrată doar de la intrări către ieșiri;
- Rețele feedback – rețele în care există legături între starturile superioare către cele inferioare.

1.4.4. Modalități de învățare

Învățarea cu supervizare este tipul de învățare inductivă ce pleacă de la un set de exemple de instanțe ale problemei și formează o funcție de evaluare care să permită clasificarea (rezolvarea) unor instanțe noi. Învățarea este supervizată în sensul că setul de exemple este dat

împreună cu clasificarea lor corectă. Deci cu alte cuvinte, presupune existența în orice moment a unei valori dorite a fiecărui neuron din stratul de ieșire.

Învățarea fără supervizare presupune ca setul de antrenare să conțină doar datele de intrare. Rețeaua extrage singură anumite caracteristici importante datelor de ieșire în urma unei competiții dintre neuroni.

1.4.5. Funcții de activare

1.4.5.1. Lineară

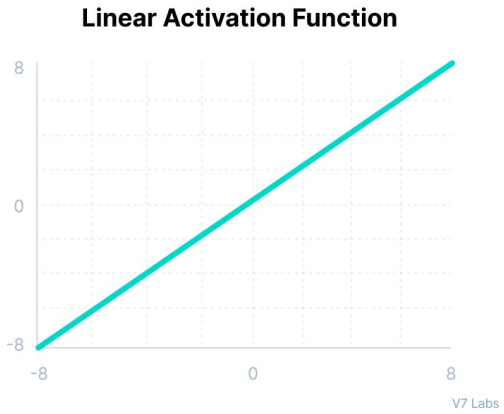

Figura 1.8. Funcția de activare liniară.

$$f(x)=x \tag{2}$$

1.4.5.2. SELU – Scaled Exponential Linear Unit

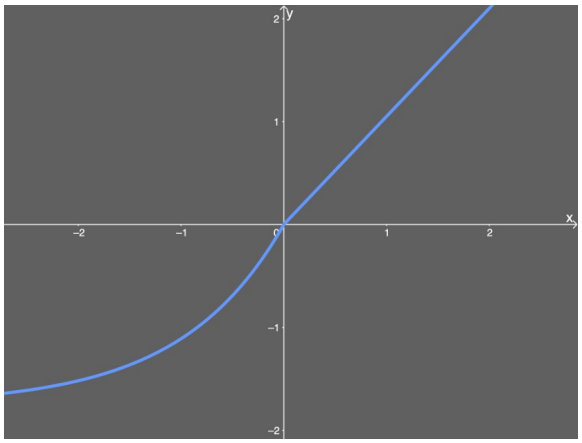

Figura 1.9. Selu.

$$f(x)=\lambda\begin{cases} x, x>0 \\ \alpha e^{x}-\alpha, x\leq 0 \end{cases} \tag{3}$$

Unde:

- $\alpha = 1.6732632423543772848170429916717$;
- $\lambda = 1.0507009873554804934193349852946$;

1.5. Deep Learning

Deep Learning este un subset al machine learning-ului, care în esență este o rețea neuronală cu trei sau mai multe straturi [6]. Rețelele neuronale încearcă să simuleze comportamentul creierului uman, făcând posibilă învățarea din set-uri mari de date. Straturile adiționale pot ajuta în optimizarea și rafinarea acurateței predicțiilor.

Deep Learning este prezentă în numeroase aplicații și servicii inteligente care îmbunătățesc automatizarea, realizează task-uri analitice și fizice, fără intervenția umană. Domenii în care este întâlnită Deep Learning, sau învățarea adâncă:

- experiența clientului – utilizată în chatbots pentru îmbunătățirea satisfacției clientului;
- generarea de text;
- aviatică și armată – în detecția obiectelor de interes din satelit;
- cercetare medicală – în cercetarea cancerului, pentru detecția automată a celulelor canceroase;
- computer vision – detecția obiectelor, clasificarea imaginilor, restaurare și segmentare.

1.6. Deep Q-Learning

Q-Learning este un algoritm simplu, și totuși puternic, care ajută un agent să ia decizia corectă în funcție de starea în care se află. Problema apare atunci când mulțimea stărilor este de o anvergură mult prea mare. Considerând o mulțime a stărilor cu 10000 elemente și o mulțime a acțiunilor cu 1000 elemente, ar rezulta necesitatea unui Q-Table cu 10 milioane de celule, ceea ce este enorm de mult. Cu creșterea numărului de stări și acțiuni, direct proporțional crește și spațiul de memorie necesar stocării datelor, în special pentru Q-Table. Deci, putem trage concluzia că algoritmul Q nu este scalabil cu mediul supus analizei, atât din punct de vedere al memoriei, cât și al timpului necesar explorării pentru crearea tabelului [7]. Astfel s-a ajuns la ideea ca valorile Q să fie aproximate cu ajutorul rețelelor neuronale.

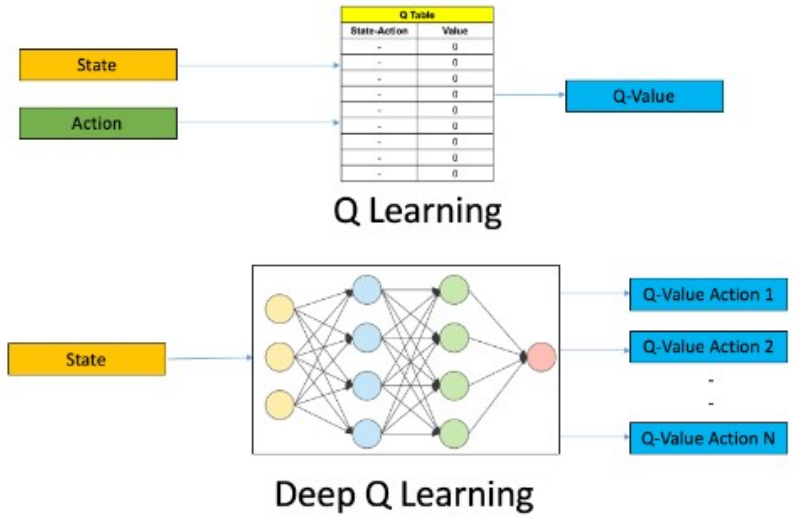

Figura 1.10. Modelul Q-Learning și modelul Deep Q-Learning.

În Figura 1.10, se observă faptul că valorile Q sunt aproximate cu ajutorul rețelei neuronale, care ia ca intrări starea mediului. Ieșirea rețelei neuronale sunt valorile Q al tuturor acțiunilor posibile pentru starea de intrare.

În continuare, sunt prezentați pașii implicați în Deep Q-Learning:

1. Toată experiența trecută este salvată în memorie;

2. Următoarea acțiune este determinată de valoarea Q maximă din ieșirile rețelei neuronale;
3. Ajustarea ponderilor se realizează cu eroarea medie pătratică dintre valoarea Q prezisă și valoarea Q țintă. Din Figura 1.3, calculul valorii Q țintă este extras ca:

$$Q^*(s_t, a_t) \approx r_t + \gamma\, Q(s_{t+1}, \text{argmax}_{a'} Q'(s_t, a_t))$$

Figura 1.11. Calculul valorii Q țintă în Deep Q-Learning.

Având în vedere faptul că aceeași rețea neuronală calculează valoarea prezisă și valoarea țintă, pot apărea divergențe între cele două valori. Pentru rezolvarea acestei probleme se utilizează două rețele neuronale, una pentru realizarea predicției și una pentru calculul valorii țintă.

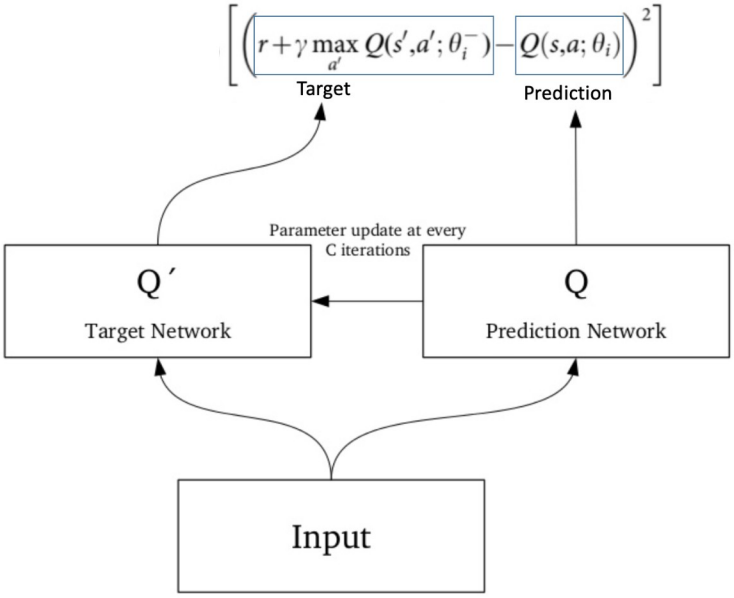

Figura 1.12. Modelul Double Deep Q Network.

Rețeaua neuronală care se ocupă cu calculul valorilor țintă are aceeași arhitectură cu rețeaua neuronală care se ocupă cu calculul predicțiilor, cu mențiunea că ponderile sunt staționare. După un anumit număr de pași, parametrii rețelei neuronale care se ocupă cu predicția, sunt copiați în rețeaua neuronală care calculează țintele. Faptul că țintele sunt ținute fixe, pentru o perioadă de vreme, face antrenarea mult mai stabilă.

Algorithm 1 : Double Q-learning (Hasselt et al., 2015)

Initialize primary network Q_θ, target network $Q_{\theta'}$, replay buffer $\mathcal{D}$, $\tau << 1$
for each iteration **do**
  for each environment step **do**
    Observe state s_t and select $a_t \sim \pi(a_t, s_t)$
    Execute a_t and observe next state s_{t+1} and reward $r_t = R(s_t, a_t)$
    Store (s_t, a_t, r_t, s_{t+1}) in replay buffer $\mathcal{D}$
  for each update step **do**
    sample $e_t = (s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$
    Compute target Q value:
      $Q^*(s_t, a_t) \approx r_t + \gamma\, Q_\theta(s_{t+1}, argmax_{a'} Q_{\theta'}(s_{t+1}, a'))$
    Perform gradient descent step on $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$
    Update target network parameters:
      $\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$

Figura 1.13. Pseudocod Deep Q-Learning [0].

Antrenarea rețelei este realizată prin reluarea experienței. Pentru aceasta, experiența agentului este reținută în memorie sub forma [stare, acțiune, recompensă, stare următoare], urmând ca din această memorie să fie extrase aleatoriu un număr de înregistrări cu care să fie realizată antrenarea rețelei neuronale.

Concluzionăm că următorii pași sunt implicați în Deep Q-Learning:

1. Obținerea stării;
2. Selectează o acțiune conform politicii epsilon-greedy;
3. Execută acțiunea și obține recompensa;
4. Reține starea în memorie;
5. Alegerea aleatorie a unui număr de înregistrări din memorie;
6. Antrenează rețeaua neuronală pe baza setului de înregistrări extras;
7. După un număr de iterații copie parametrii din rețeaua neuronală de predicție în rețeaua neuronală de calcul a țintelor;
8. Repetă pașii pentru un anumit număr de episoade.

1.7. Tehnologii

1.7.1. Unity 3D

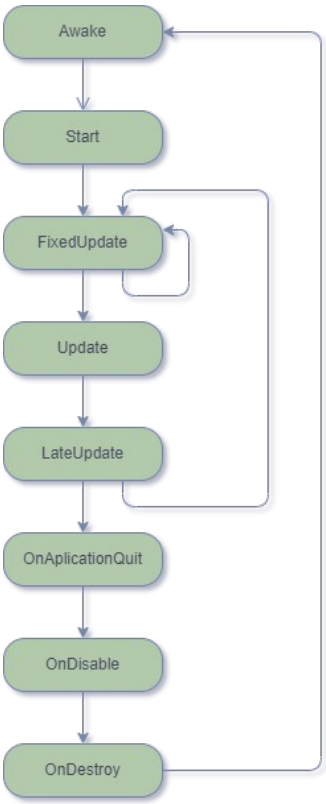

Figura 1.14. Pașii necesari rezumați ai unity 3d.

Unity 3D este un sistem de dezvoltare, pentru multiple platforme, al jocurilor. Cu ajutorul acestuia se pot realiza jocuri trei dimensionale, 3D, sau doi dimensionale, 2D, dar și simulări sau simulatoare interactive. Ce este de reținut, și foarte important de luat în considerare, este modelul de execuție Unity, care influențează realizarea proiectului în consecință. În Figura 1.14, sunt rezumați numai câțiva pași importanți în dezvoltarea proiectului. Diagrama completă se găsește pe pagina oficială pentru documentație a Unity.

Capitolul 2. Proiectarea

2.1. Specificații hardware

În dezvoltarea proiectului, următoarele specificații hardware au fost implicate:

- Processor: Intel(R) Core(TM) i7-9750;
- RAM: 16 GB;
- DISK: SSD M2 500GB;
- GPU:
  - Intel(R) UHD Graphics 630;
  - NVIDIA GeForce GTX 1660 Ti.

2.2. Specificații software

Din punct de vedere al specificațiilor software, proiectul a fost dezvoltat pentru platforme de Windows, sistemul de operare gazdă fiind Windows 11, următoarele software-uri fiind implicate:

- Unity3D 2019.4.24f1 – utilizat pentru realizarea componentei grafice a simulatorului, pentru testare și realizarea build-ului aplicației;
- Visual Studio Community 2019
  - pentru realizarea proiectului în C++, în vederea realizării unei structuri pentru rețeaua neuronală și a wraperului în C# necesar pentru includerea dll-ului, ce conține implementarea rețelei neuronale și a algoritmului de Deep Q-Learning, în Unity;
  - pentru scrierea scripturilor de C# necesare componentelor Unity, în vederea realizării simulatorului și a relațiilor dintre componentele acestuia.

2.3. Sistemul de dispatch

În Figura 1.14, cei câțiva pași rezumați, care intervin în funcționarea sistemului Unity 3D, sunt executați secvențial în ordinea prezentată de către firul de execuție principal, cunoscut ca MainThread. Mai mult, după cum se știe, toate interacțiunile cu interfața grafică, în orice aplicație desktop, trebuie realizate de pe firul de execuție principal, care se ocupă cu controlul detaliilor grafice. În sistemele de construire a aplicațiilor desktop, precum proiectele de tip WPF, adesea întâlnim modalități de a interacționa cu firul principal de execuție prin intermediul unui „dispatcher”. Practic, ce face acest dispatcher este să primească task-uri, în care sunt subliniate interacțiunile cu interfața grafică, și să le dea firului principal de execuție la anumite momente.

La ce ne referim prin „anumite momente”? Ne referim la pasul, pe care îl putem interpreta ca un moment, din multitudinea de pași secvențiali pe care trebuie să îi execute.

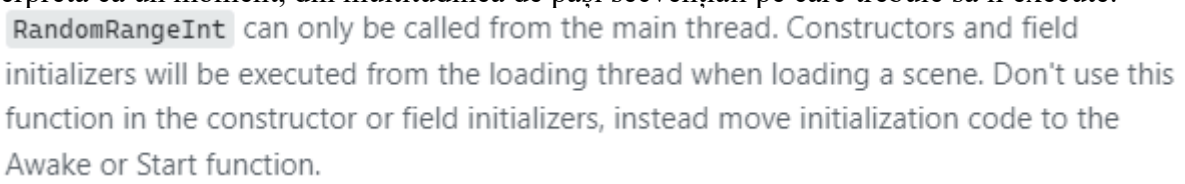

Figura 2.1. Exemplu eroare de main thread în unity.

În Unity, sunt numeroase acțiuni care nu pot fi realizate prin intermediul oricărui thread, și este obligatoriu a fi realizate în thread-ul principal de execuție. Drept exemplu în acest sens, avem eroarea din Figura 2.1, în care este evidențiată problema în cauză, cât și soluția. Pentru simulatorul de evacuare, avem de a face cu sistemul Physic al unity pentru obținerea

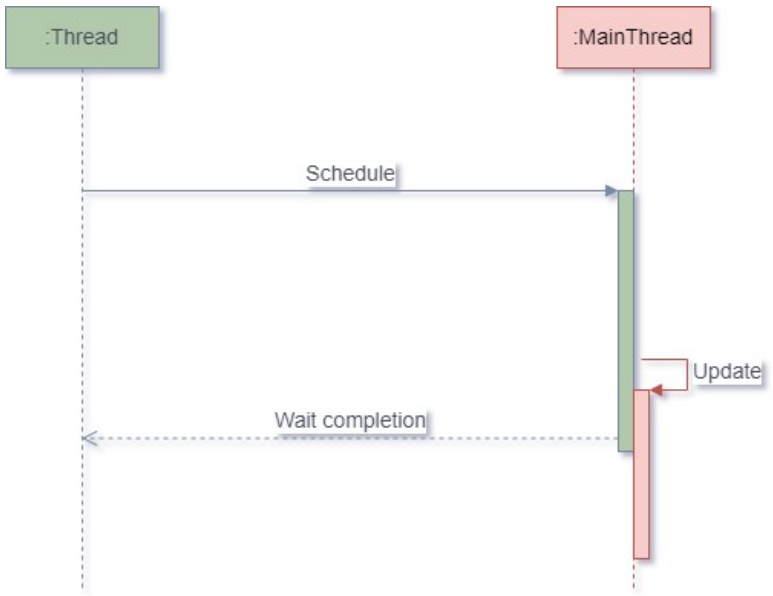

Figura 2.2. Unity dispatcher.

observațiilor din mediu, și totodată este necesară și interacțiunea cu mediul simulării pentru crearea, distrugerea și modificarea obiectelor care iau parte la simulare. Aceste acțiuni pot fi realizate doar de pe thread-ul principal, și cum în unity nu este un dispatcher implicit, s-a propus realizarea unuia în acest sens.

```
_context.Simulator.Dispatcher.Schedule(() =>
{
    _context.Simulator.BotsManager.Reset();
}).WaitOne();
```

Figura 2.3. Exemplu de programare a unui task la execuție pe threadul principal.

Practic, ce face acest dispatcher este să preia task-uri de la thread-uri și să le rețină până thread-ul principal ajunge să le execute. Pentru aceasta, ambele thread-uri trebuie sa poată accesa acest obiect de dispatch. În urma programării la execuție a unui task, este returnat un obiect către thread-ul care realizează programarea, cu ajutorul căruia se poate aștepta completarea task-ului.

2.4. Proiectarea Simulării

2.4.1. Modul general de funcționare

Din punctul de vedere al utilizatorului, deci vorbind prin referire la interfața grafică, modul de funcționare a simulatorului este simplu. Simulatorul pune la dispoziție două scene Unity. Scenele sunt componente în care se lucrează cu conținutul Unity, acesta însemnând obiecte 3D, animații, camere și altele. Aceste două scene sunt, după cum urmează:

1. Scena meniului – în care sunt disponibile câteva configurații importante pentru rularea simulării, precum:
   - Setări pentru indivizi – numărul și viteza de mișcare a acestora. Pentru viteza de mișcare a indivizilor este de menționat faptul că rețeaua neuronală care corectează direcția de mers a indivizilor a fost antrenată cu viteza la unitatea egală cu 1. În configurări aceasta poate fi modificată între 1 și 1.4 inclusiv, asta însemnând că putem accelera indivizii cu până la 40%, dar există posibilitatea ca odată cu creșterea vitezei, în unele condiții, precizia urmăririi căii de ieșire să scadă.

- Setări pentru simulare – numărul de runde de simulare și timpul de afișare a rezultatelor între runde, pentru ultima rundă rezultatele rămânând afișate permanent până la revenirea la meniu. Este de preferat ca pentru antrenare să se seteze această valoare la minimul permis.
- Setări pentru flăcări – acestea constând în dimensiunea flăcărilor și viteza de propagare a lor.
- Setări pentru sunet – în care se pot seta volumele pentru sunetul din scena meniului și alarma din scena de simulare.
- O secțiune de diverse în care se poate bifa căsuța pentru antrenare, care practic pune sistemul în starea de antrenare.

2. Scena simulării – în care este realizată simularea.

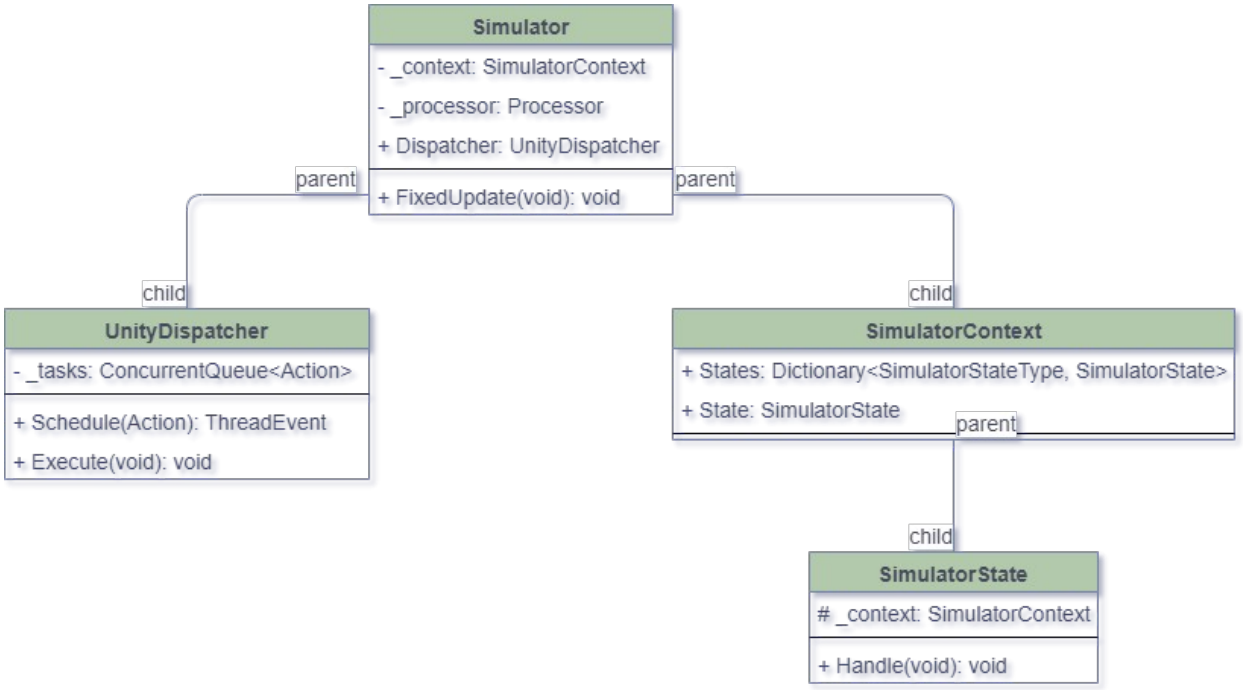

Figura 2.4. Clase principale ale simulării.

Procedeul simulării este realizat prin intermediul clasei „Simulator”, din Figura 2.4, care este atașată în scena simulării. Metodele specifice Unity ale acesteia (Start, Update, FixedUpdate) sunt executate pe thread-ul principal. Metoda „Execute” a clasei UnityDispatcher este executată pe thread-ul principal în cadrul metodei „FixedUpdate”, care practic execută toate task-urile adunate prin metoda „Schedule” până la momentul curent.

2.4.2. Stările simulării

Pentru a separa operațiile care nu necesită execuția pe thread-ul principal, s-a realizat mutarea lor pe un thread separat de execuție. Această mutare a fost făcută prin intermediul design pattern-ului state, a cărei execuție este realizată pe thread-ul secundar, pus la dispoziție prin intermediul clasei Processor. Un nivel mare de procesare pe thread-ul principal va duce la o scădere a performanței atât grafice, cât și logice la nivel de back-end. Acest lucru este vizibil prin apariția fenomenului de „FPS drop”, în care practic scade semnificativ numărul de frame-uri pe secundă. Acest fenomen se prezintă ca o mișcare sacadată, întreruptă și incoerentă, a interfeței grafice. De aceea este necesar executarea tuturor operațiilor logice pe un thread separat, apelându-se la thread-ul principal prin intermediul dispatcher-ului, pentru operații obligatorii thread-ului principal, doar la nevoie.

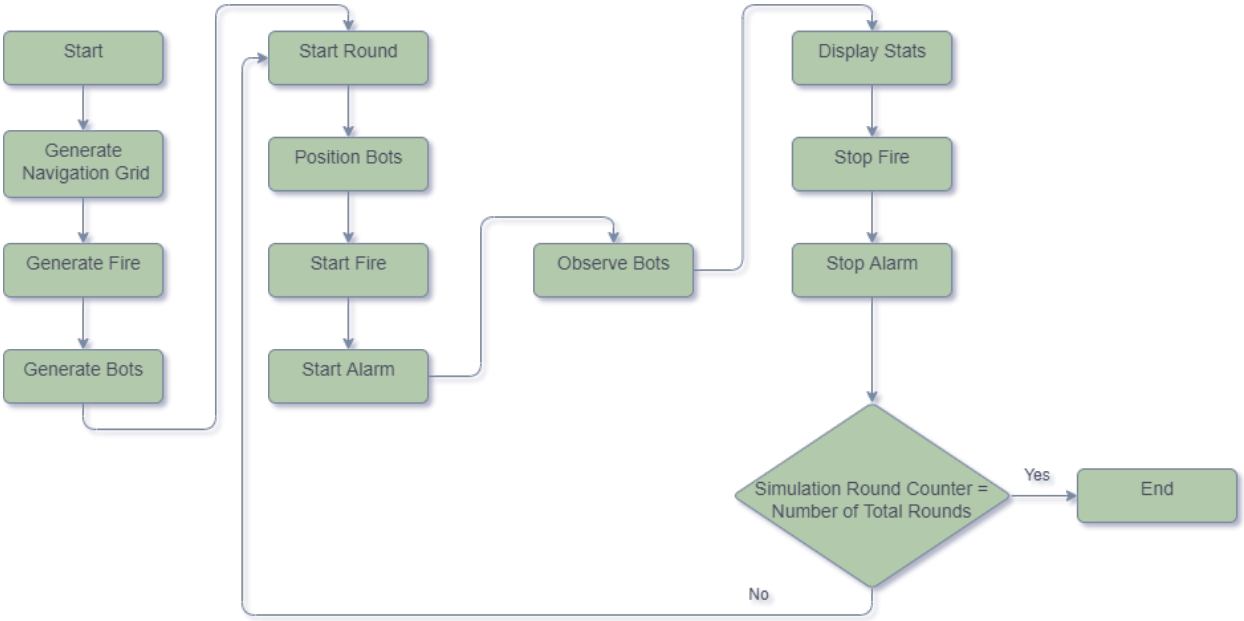

Figura 2.5. Stările simulării.

În Figura 2.5, sunt prezentate stările simulării și relațiile dintre ele. Practic aceste stări de funcționare, introduse prin intermediul design pattern-ului de state, sunt efectiv pașii care intervin în simulare, în ordinea prezentată în diagramă.

- Start – este starea în care se pregătesc detaliile necesare pornirii simulării.
- Generate Navigation Grid – pornește corutina de generare a gridului de navigare și așteaptă până la terminarea acesteia.
- Generate Fire – pornește corutina care se ocupă cu generarea gridului de flăcări, și implicit a acestora, și așteaptă până la terminarea corutinei.
- Generate Bots – crează indivizii implicați în simulare, tot prin intermediul unei corutine.
- Start Round – pregătește detaliile necesare pentru runda ce urmează a fi executată.
- Position Bots – asignează indivizilor noi poziții aleatorii de plecare.
- Start/Stop Fire – la start se pornește expansiunea nodurilor de foc, iar la stop este oprită corutina care se ocupă cu expansiunea.
- Start/Stop Alarm – pornește/oprește alarma de incendiu, a cărei volum poate fi setat din configurări.
- Observe Bots – practic notifică indivizii că trebuie să evacueze clădirea, și așteaptă până ce toți ajung în una din stările Safe/Trapped/Dead.
- Display Stats – afișează pe interfața grafică rezultatele din urma rundei de simulare.
- RoundCheck – reprezentat prin rombul cu condiția de egalitate, verifică dacă mai sunt runde de simulare de executat.
- End – este starea în care este eliberată memoria (nodurile de foc și indivizii sunt distruși).

După cum se poate observa, modelul arhitectural al simulării, care pune în contrast etapele simulării, este de o scalabilitate deosebită, permițând adăugarea unor etape noi cu ușurință, fără a fi necesară modificarea stărilor vechi, ci doar a relațiilor dintre ele. Totodată, acestea fiind executate pe un thread separat față de cel principal, este ușor realizabilă separarea în execuție astfel încât să nu fie puternic influențată execuția thread-ului principal.

Evident, capabilitatea de execuție depinde mult și de specificațiile hardware pe care este executat sistemul de simulare, cât și disponibilitatea sistemului de operare la noi thread-uri date către execuție, dar este foarte importantă construirea unei arhitecturi optime din punct de vedere al procesării, chiar dacă în unele instanțe sunt necesare unele detalii, poate, mai complexe.

2.5. Proiectarea indivizilor

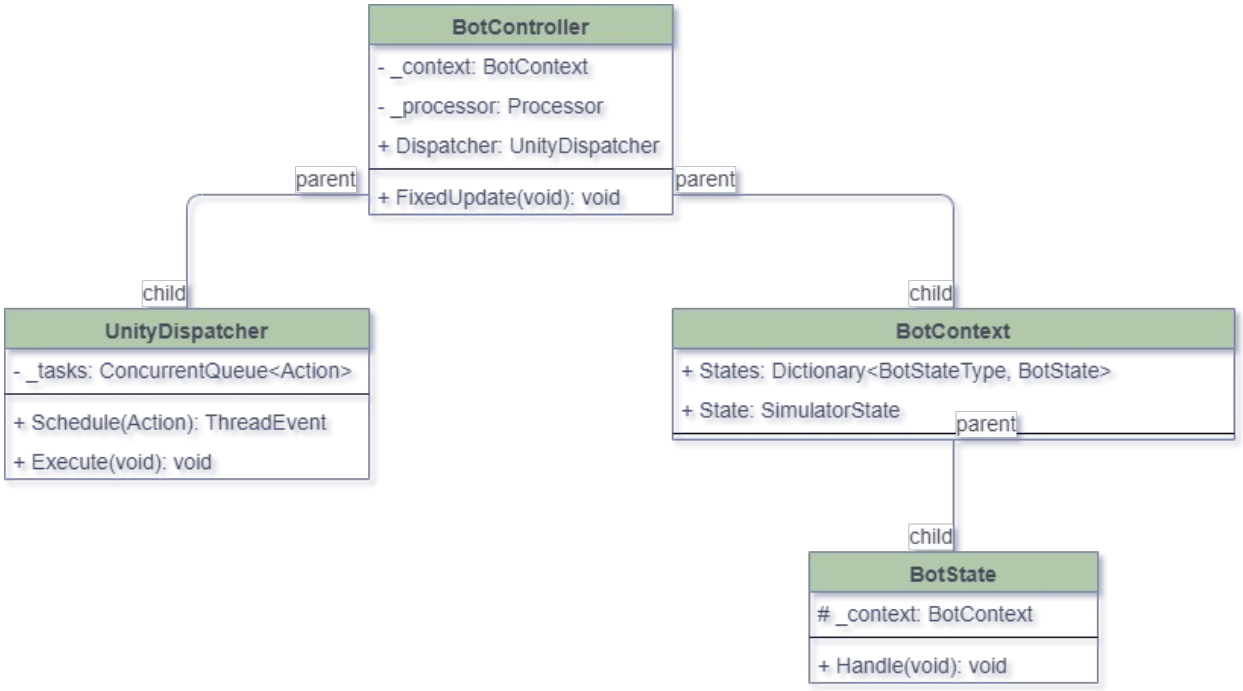

Figura 2.6. Clasele principale ale indivizilor.

Fiecare GameObject, al fiecărui individ, are atașat un script care la load, creează o instanță a clasei „BotController”. Se poate observa faptul că modelul din Figura 2.6 e identic cu cel utilizat pentru simulare, deci putem spune că este reutilizabil. În metoda „FixedUpdate” sunt executate și task-urile stocate în dispatcher, care se ocupă cu preluarea datelor de interacțiune a individului, sau bot-ului, cu mediul, sau altfel spus observațiile legate de starea în care se află acesta în mediul simulării.

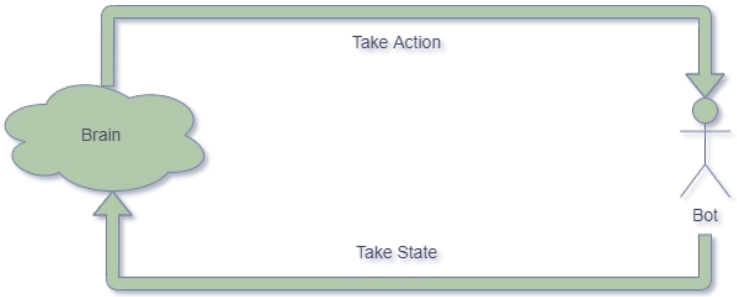

Figura 2.7. Interacțiunea rețelei neuronale cu bot-ul aflat în mediul simulării.

În Figura 2.7, unde „Brain” reprezintă rețeaua neuronală care se ocupă cu corecția direcției bot-ului, iar „Bot” este GameObject-ul individului din spațiul simulării, se poate observa că interacțiunea prezentată este practic interacțiunea din Deep Q-Learning dintre un agent și mediul în care performează.

2.5.1. Stările de funcționare

Similar ca în cazul simulării, și pentru indivizi a fost implementat design pattern-ul de state, cu ajutorul căruia se expun stările de funcționare și relațiile dintre acestea. Principala necesitate a acestuia este de se ocupa de interacțiunea rețelei neuronale cu spațiul simulării în

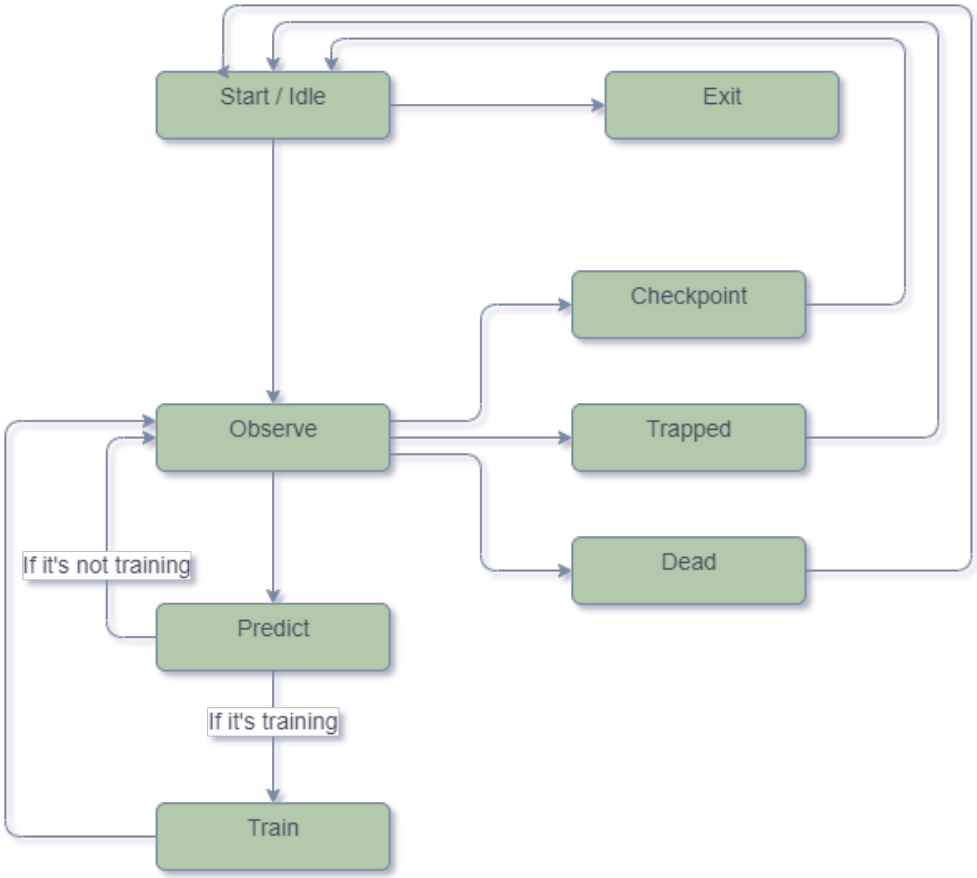

Figura 2.8. Stările de funcționare a individului.

care se află individul.

În continuare vor fi rezumate caracteristicile stărilor prezente în Figura 2.8:

- Start/Idle – starea în care individul așteaptă să fie notificat de prezența incendiului, și în care este pregătit individul pentru evacuare.
- Observe – starea în care este preluată starea în care se află individul în spațiul simulării cu referire la punctul de referință din calea de navigare către punctul de siguranță, sau altfel spus: observațiile.
- Predict – starea în care este realizată predicția următoarei acțiuni pe baza observațiilor preluate la pasul anterior.
- Train – starea care este realizată numai dacă bot-ul antrenează rețeaua neuronală, și care reprezintă antrenarea rețelei prin reluarea experienței.
- Checkpoint/Trapped/Dead – sunt stările în care ajunge bot-ul în urma interacțiunii cu sistemul de simulare:
  - Checkpoint – dacă a ajuns într-o zonă de siguranță;
  - Trapped – dacă nu există o cale de evacuare între poziția curentă și poziția unei zone de siguranță;
  - Dead – dacă a intrat în contact cu flăcările.
- Exit – este starea care întrerupe ciclul de pași.

Ca și în cazul simulării, și pentru indivizi intervin mai multe thread-uri, din aceleași motive explicate anterior:

- thread-ul principal pe care sunt executate metodele specifice Unity;
- un thread pentru execuția stărilor;
- și un thread care se ocupă cu consumul nodurilor din calea de navigare.

Practic, pentru a ieși din clădire, individul are nevoie de o cale de evacuare, precalculată, deoarece clădirea este complexă, și rețeaua neuronală, în forma sa curentă, simplă, nu este capabilă să evacueze individul din clădire. Și atunci, se calculează o cale de evacuare, utilizând gridul de navigare, la primirea semnalului de a evacua, sau atunci când calea nu mai este validă.

Necesitatea unui thread care să se ocupe special de calea de navigare a apărut deoarece în timp ce sunt executate stările, individul își schimbă starea în spațiul simulării, și dacă calea de navigare era gestionată la nivelul stării Observe, apărea posibilitatea ca uneori individul să treacă de punctul de referință din calea de navigare, și nu ar fi reușit practic să consume acel nod la timp.

2.5.2. Rețeaua neuronală

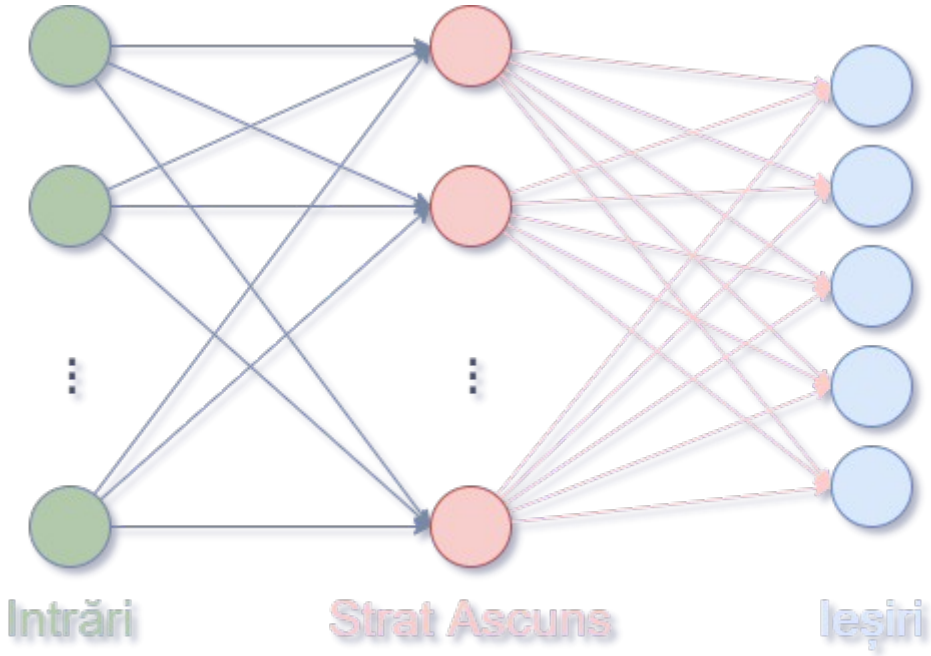

Figura 2.9. Rețeaua neuronală de predicție a acțiunilor bot-ului.

În Figura 2.9 este prezentată arhitectura rețelei neuronale implicate în predicția acțiunilor individului. După cum se poate vedea, rețeaua are un singur strat ascuns, care este de aceeași dimensiune cu stratul intrărilor, aceasta fiind numărul de observații, a căror format va fi prezentat în sub-subcapitolul următor. Stratul ieșirilor înfățișează 5 ieșiri, care reprezintă velocitatea țintă a direcției în care individul se îndreaptă. Odată ce se dă notificarea de evacuare, bot-ul începe să meargă la o viteză constantă, și se oprește doar dacă a ajuns în zona de siguranță sau intră în flăcări. Pe parcursul mersului, direcția țintă este corectată frecvent de către rețeaua neuronală, antrenată prin Deep Q-Learning, pentru a urma calea de navigare.

După cum s-a specificat, ce se corectează este direcția țintă, și nu direcția efectivă. Direcția este actualizată constant la fiecare apel al metodei FixedUpdate către direcția țintă cu un pas fix de 0.1. Limitele direcției sunt [-1, 1]. Ieșirile rețelei neuronale reprezintă 5 valori țintă ale direcției, în intervalul menționat:

- -1;
- -0.7;
- 0;
- 0.7;
- 1.

2.5.3. Formatul observațiilor

Formatul observațiilor este în ordine, după cum urmează:

- 23 de intrări ale distanțelor, preluate de lasere plecând de la nivelul mijlocului bot-ului, paralel cu podeaua, a căror distanță maximă de propagare este de 5 unități, plecând de la un unghi de -110 grade până la 110 grade cu un offset de 10.
- 23 de intrări ale distanțelor, preluate de lasere ce pleacă de la nivelul gleznelor.
- 46 de intrări care detectează tipul de coliziune întâlnită de lasere:
  - 2 - existența flăcărilor la distanța dată de laser;
  - 1 – existența unui bot;
  - 0 – orice altceva.
- o intrare care specifică unghiul dintre direcția înainte a bot-ului și punctul de referință.
- o intrare care specifică valoarea parametrului direcției efective la momentul preluării observației.
- o intrare care specifică distanța între bot și punctul de referință.
- o intrare pentru validarea coliziunii cu un alt bot.
- o intrare pentru validarea coliziunii cu focul.
- o intrare pentru validarea coliziunii cu orice alt obiect (perete, birou, etc.).

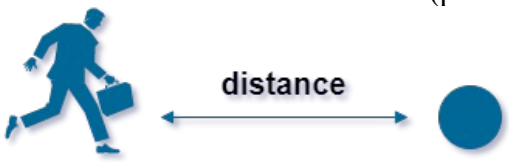

Figura 2.10. Măsurarea distanței între bot și punctul de referință.

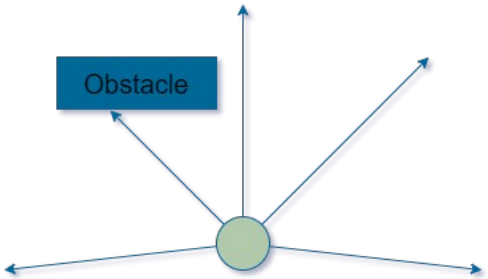

Figura 2.11. Obținerea observațiilor utilizând laserele.

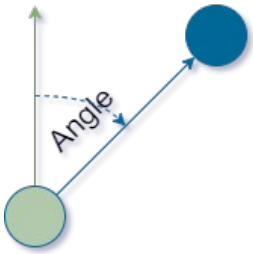

Figura 2.12.
Obținerea ungiului.

În Figura 2.10, Figura 2.11, Figura 2.12, sunt exemplificate modalitățile de obținere a datelor necesare unei observații la un moment dat.

2.6. Proiectarea sistemului de navigare

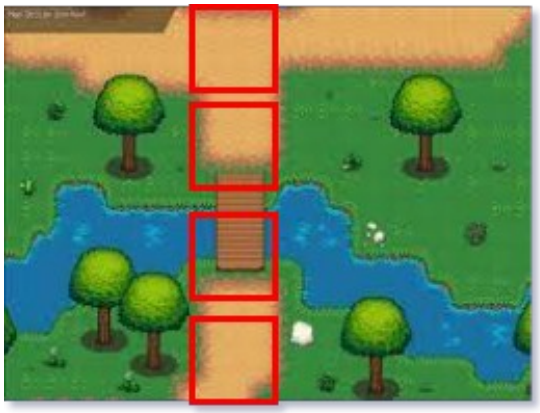

Figura 2.13. Exemplu minimal al logicii de creare a gridului de navigare.

Pentru a ajuta individul să evacueze clădirea trebuie să îi oferim modalitatea de a obține o cale din orice punct al clădirii până într-o zonă de siguranță. Ca analogie, putem considera faptul că orice clădire are asigurate în caz de nevoia unei evacuări, semne de ieșire care să direcționeze ocupanții către ieșire. În aceeași manieră, s-a stabilit utilizarea unei căi de navigare pentru a direcționa indivizii în afara clădirii, cu mențiunea că segmentarea între punctele din calea de navigare este mult mai mare decât în realitate.

Primul lucru care trebuie realizat este crearea unui grid de puncte tri-dimensional, care să acapareze toată suprafața destinată simulării, deci clădirea, în toată anvergura ei, și punctele de siguranță. Acest grid are ca scop împărțirea spațiului, ne putem imagina, într-o multitudine de cuburi, similar Figura 2.13 și Figura 2.14. Centrele acestor cuburi, care sunt puncte de coordonate în spațiul 3D, vor fi reținute într-o matrice tri-dimensională cu dimensiunile necesare. Practic, centrele sunt importante, și distanța între acestea, pentru viitoare calcule.

După ce s-a realizat crearea gridului, este necesară o filtrare a punctelor, după următoarele criterii:

- toate punctele a căror cuburi se suprapun cu obiecte, pereți, podele, sau orice alt obiect care nu poate fi depășit, mers peste, sunt eliminate.
- toate punctele din spațiul gol sunt eliminate, mai puțin cele de pe primele nivele. Pentru a putea fi înțeles acest procedeu, imaginați-vă o cameră goală plină de cuburi, dispuse unul peste altul. Toate cuburile sunt eliminate, mai puțin primul rând de cuburi, cel de pe podea.

Filtrarea, după criteriile de mai sus, ne oferă zonele prin care se poate deplasa individul. Crearea și filtrarea sunt realizate la începutul simulării, o singură dată, și nu la fiecare rundă. Pentru obținerea unei căi de navigare se folosește algoritmul A* prezentat în capitolul de **Fundamente Teoretice**.

Aspecte importante ale navigării acestei căi, sunt:

- calea constă într-o multitudine de noduri, și pe parcursul urmării acesteia, se tot elimină primul nod, în funcție de distanța față de acesta.
- calea este fixă, apariția flăcărilor peste aceasta, face nodurile din zona respectivă să devină de neparcurs.
- la întâlnirea unui nod de neparcurs, indivizii trebuie să recalculeze calea de evacuare.

Un exemplu foarte minimal legat de cum ar fi arăta acest grid de navigare se găsește în Figura 2.13, unde pătratele roșii sunt zonele care pot fi parcurse, restul zonelor fiind inaccesibile.

2.7. Proiectarea sistemului de extindere a flăcărilor

Figura 2.14. Modul de dispunere a flăcărilor.

Pentru extinderea flăcărilor s-a abordat aceeași manieră ca și în cazul gridului de navigare, cu excepția că, flăcările, fiind construite cu sistemul de particule de la Unity, afectează sistemul când numărul lor este prea mare și dimensiunea în scală este prea mică. În acest sens, dimensiunea a fost restricționată între 3 și 6, în funcție de dimensiune, flăcările fiind așezate mai bine sau nu în clădire. De menționat este faptul că gridul flăcărilor cuprinde numai clădirea de evacuat.

Legat de dimensiunea flăcărilor, pentru o dimensiune egală cu 3, sunt 11*11*11 noduri, deci 1331 de flăcări, și implicit sisteme de particule. Considerând o dimensiune de 1, am avea 33*33*33 noduri, deci 35936 flăcări, ceea ce este considerabil foarte mult.

Totodată, nu mai este realizată o filtrare, având în vedere faptul că nodurile de foc sunt prea mari în dimensiune pentru a realiza filtrarea de mai sus cu succes.

2.8. Proiectarea sistemului de generare și poziționare a indivizilor

2.9. Proiectarea DQN

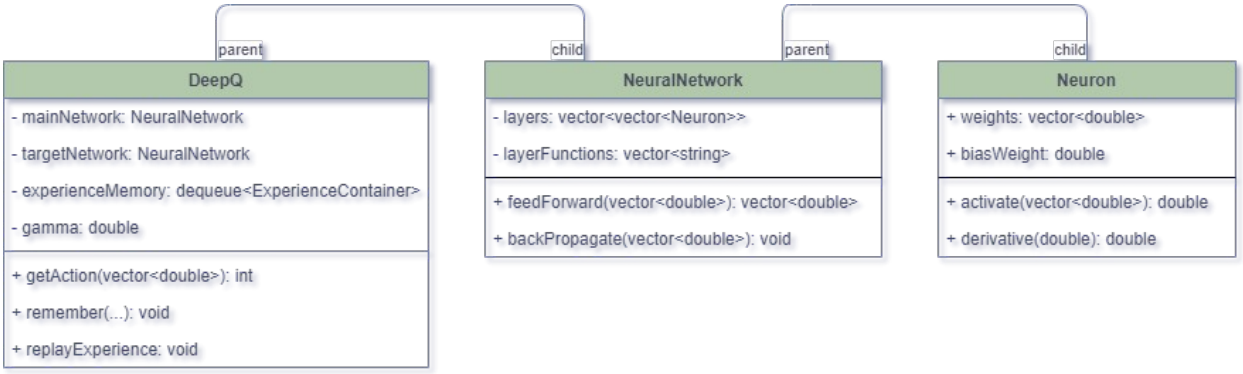

Figura 2.15. Structura de clase pentru DQN.

Componentele necesare construirii rețelei neuronale antrenate prin Deep Q-Learning sunt

prezentate sumar în Figura 2.15. Aceste clase sunt implementate în C++, iar peste ele a fost implementat un wrapper în C# pentru transferul lor în Unity. Metodele clasei DeepQ sunt pași ai Deep Q-Learning:

- getAction – realizează predicția acțiunii care trebuie luată în starea curentă.
- remember – memorează experiența
- replayExperience – reia experiența în vederea antrenării rețelei neuronale.

Se observă faptul că sunt folosite două rețele neuronale, una pentru predicția acțiunii, care este constant antrenată, și una pentru predicția valorilor Q țintă, care este actualizată după un anumit număr de pași, astfel fiind pus în practică modelul Double Deep Q Network.

2.10. Proiectarea recompensei

Recompensa este calculată la fiecare memorare a experienței, fiind inițializată cu 0, după care sunt aplicate următoarele formule:

$$R = R + (-1) * distance * |angle| \tag{4}$$

Unde:

- distance – distanța între bot și punctul de referință;
- angle – unghiul între direcția înainte și punctul de referință.

$$R = R + botCollision * (-300) + fireCollision * (-10000) + otherCollision * (-1000) \tag{5}$$

Unde:

- botCollision – 1 dacă există coliziune cu un alt bot, altfel 0;
- fireCollision – 1 dacă există coliziune cu foc, altfel 0;
- otherCollision – 1 dacă există coliziune cu orice altceva, altfel 0.

Capitolul 3. Implementarea

3.1. Scena simulării

Din punct de vedere al interfeței grafice, scena simulării este realizată din multiple elemente, în Unity numite GameObjects, după cum urmează:

- terenul pe care sunt amplasate obiectele simulării;
- decorul exterior format din arbori;
- clădirea preluată ca asset gratuit de pe UnityStore și care poate fi găsită la adresa https://assetstore.unity.com/packages/3d/environments/urban/polygon-office-building-82282;
- bot-ul, preluată ca asset gratuit de pe UnityStore și care poate fi găsit la adresa https://assetstore.unity.com/packages/3d/characters/humanoids/humans/man-in-a-suit-51662;
- ferestrele de afișaj a mesajelor și statisticilor, și a butoanelor.

3.1.1. Clădirea

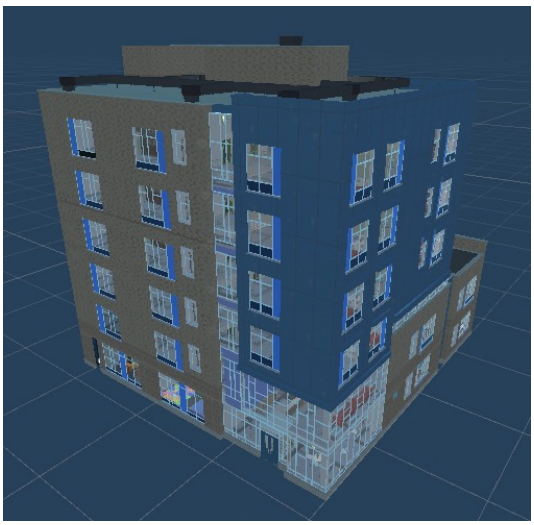

Figura 3.1. Asset-ul clădirii.

Se poate observa, din Figura 3.1, că clădirea simulării este de dimensiuni relativ mari, consistând în șase nivele și trei ieșiri către exterior.

Figura 3.2. Modelul interior al clădirii.

Deoarece individul nu a fost proiectat să își deschidă ușa singur și de obicei, în timpul unei evacuări, ușile sunt lăsate deschise, s-a optat pentru opțiunea de a deschide toate ușile la începutul simulării. Totuși deschiderea ușilor bloca trecerea pe scările secundare, acestea având o lățime destul de mică. În acest sens ușile au fost scoase din simulare prin dezactivarea lor. Se poate observa, din Figura 3.2, complexitatea interiorului clădirii, ceea ce întărește relevanța simulării.

3.1.2. Nodurile de foc

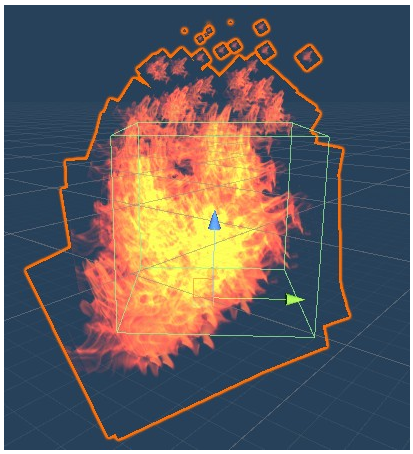

Figura 3.3. Sistemul de particule al nodului de foc.

Figura 3.3 prezintă imaginea sistemului de particule, staționară, care în simulare reprezintă nodul de foc. Se observă un contur verde sub forma unui cub. Acele limite reprezintă limitele collider-ului, BoxCollider-ului, nodului de foc, în contactul cu care, individul este trecut în starea de DEAD. După crearea gridului pentru nodurile de foc, acestea sunt create și așezate în pozițiile asociate, și ținute inactive până la necesitatea activării lor.

Expansiunea incendiului este realizată prin alegerea aleatorie a unui nod și activarea acestuia, după care rând pe rând nodurile din jurul acestuia sunt activate, la o distanță temporală, care poate fi configurată.

3.1.3. Individul

Figura 3.4. Asset-ul bot-ului.

În Figura 3.4 este prezentat GameObject-ul individului. Pentru mișcare acestuia i s-a

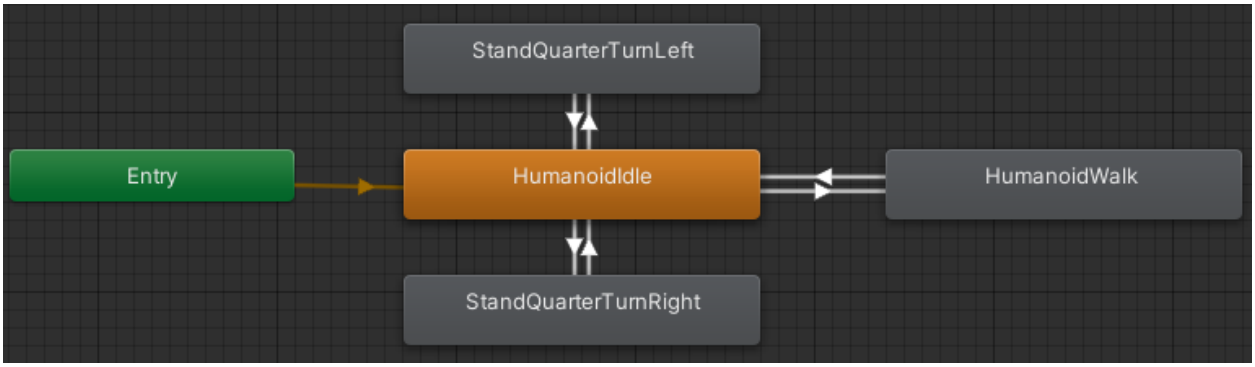

Figura 3.5. Controlerul animațiilor bot-ului.

atașat un component de tipul CharacterController, iar pentru a bloca trecerea prin alte obiecte a fost adăugat un component de tipul RigidBody. Mișcarea este controlată prin intermediul componentului Animator, care are setat drept controller modelul din Figura 3.5.

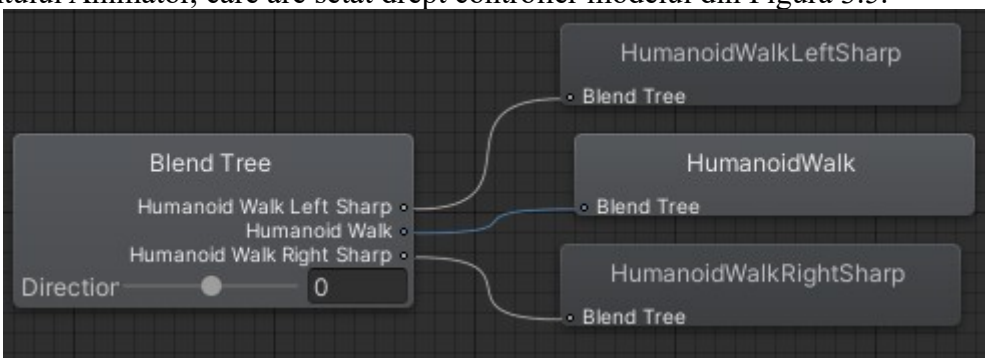

Figura 3.6. BlenTree pentru controlul direcției în mers.

HumanoidWalk este un BlendTree, Figura 3.6, prin intermediul căruia este controlată direcția de mers, în timpul mersului. Practic, parametrul Direction este actualizat, și în funcție de valoarea acestuia este schimbată animația de mers. După cum se poate vedea, sunt trei animații, de mers înainte, de mers la stânga, și de mers la dreapta. În funcție de valoarea parametrului, avem:

- 0 – bot-ul merge înainte;
- (0, 1] – bot-ul se deplasează către dreapta, și cu cât valoarea este mai apropiată de 1, cu atât unghiul este mai strâns;
- [-1, 0) – bot-ul se deplasează către stânga, și cu cât valoarea este mai apropiată de -1, cu atât unghiul este mai strâns.

3.1.3.1. Parametrii DQN

Parametrii de antrenare a rețelei neuronale, și valorile acestora, sunt după cum urmează:

- dimensiunea intrărilor este de 96 valori de tip double;
- dimensiunea acțiunilor este de 5;
- rețeaua neuronală dispune de un strat ascuns de dimensiune egală cu 96, dimensiunea intrărilor;
- rata de învățare este 0.001, de menționat fiind faptul că actualizarea ponderilor rețelei neuronale, în timpul antrenării, este realizată cu ajutorul algoritmului ADAM;
- funcția de activare pentru stratul de ieșire este liniară, iar pentru startul ascuns este „SELU”;
- factorul de discount este 0.99;
- parametrul experienței este setat inițial la 0.8 și scade până la 0.05, în timpul antrenării. Când bot-ul nu antrenează, acesta este setat la 0, în ideea că nu mai este necesară

explorarea.

- la fiecare reluare a experienței se aleg aleatoriu 50 de înregistrări și se antrenează o singură epocă;
- dimensiunea buffer-ului în care este stocată experiența este de 300;
- ponderile rețelei neuronale, care se ocupă de predicția valorilor țintă, sunt actualizate după 100 de pași.

Datele de antrenare, ale rețelei neuronale, sunt păstrate într-un fișier, în format JSON.

Liniile de cod, prezentate în continuare, instanțiază obiectul DeepQWrapper, necesar predicțiilor acțiunilor.

```
private void InitializeDeepQNetwork()
{
 // Set up the neural network properties
 NeuralNetworkWrapper neuralNetwork;
 neuralNetwork = new NeuralNetworkWrapper(Locals.DQN_LEARNING_RATE);

 if (!IsTrainingTheNetwork && File.Exists(Locals.DQN_NETWORK_FILE))
 {
     neuralNetwork.Load(Locals.DQN_NETWORK_FILE);
     // Set up the DQN properties
     Brain = new DeepQWrapper(neuralNetwork);

     // Set experience
     Brain.SetExploration(0);
     Brain.SetMinimumExploration(0);
     Brain.SetExplorationDecay(0);
     Brain.SetGamma(Locals.DQN_GAMMA);
     Brain.SetTrainEpochs(Locals.DQN_TRAIN_EPOCHS);
     Brain.SetReplayBatchSize(Locals.DQN_REPLAY_BATCH_SIZE);
     Brain.SetExperienceMemorySize(Locals.DQN_EXPERIENCE_MEMORY_SIZE);
     Brain.SetUpdateTargetWeightsAfterSteps(Locals.DQN_UPDATE_TARGET_STEPS);
 }
 else
 {
     neuralNetwork.AddLayer(Locals.DQN_HIDDEN_NO_NEURONS,
(uint)Locals.OBSERVATION_SIZE,              Locals.DQN_HIDDEN_LAYER_FUNCTION);
     neuralNetwork.AddLayer((uint)Locals.ACTION_SIZE,
Locals.DQN_HIDDEN_NO_NEURONS,               Locals.DQN_OUTPUT_LAYER_FUNCTION);

     // Set up the DQN properties
     Brain = new DeepQWrapper(neuralNetwork);

     // Set experience
     Brain.SetExploration(Locals.DQN_EXPLORATION);
     Brain.SetMinimumExploration(Locals.DQN_MINIMUM_EXPLORATION);
     Brain.SetExplorationDecay(Locals.DQN_EXPLORATION_DECAY);
     Brain.SetGamma(Locals.DQN_GAMMA);
     Brain.SetTrainEpochs(Locals.DQN_TRAIN_EPOCHS);
     Brain.SetReplayBatchSize(Locals.DQN_REPLAY_BATCH_SIZE);
     Brain.SetExperienceMemorySize(Locals.DQN_EXPERIENCE_MEMORY_SIZE);
```

```
        Brain.SetUpdateTargetWeightsAfterSteps(Locals.DQN_UPDATE_TARGET_STEPS);
    }
}
```

3.1.3.2. Mișcarea și corectarea direcției

Individul se mișcă cu o viteză constantă pe toată perioada de timp a simulării, fiind pornit în momentul notificării de incendiu, și oprit în momentul când ajunge în una dintre stările finale. Rețeaua neuronală se ocupă cu corecția direcției așa cum a fost prezentat în sub-subcapitolul 2.5.2.

```
_context.Action = _context.Brain.GetAction(_context.Observation);
_context.Bot.UpdateDirection(Locals.DIRECTION_VALUES[_context.Action]);
```

Liniile de cod, de mai sus, prezintă predicția acțiunii pentru starea curentă și actualizarea direcției țintă a individului. Metoda de update direction setează direcția țintă, la fiecare apel al metodei FixedUpdate direcția efectivă a individului fiind actualizată către direcția țintă.

```
public void FixedUpdate()
{
    // update bot position
    _context.BotPosition = transform.position;

    // Update movement direction
    UpdateDirection();

    // Execute tasks from other threads (1 task per frame)
    Dispatcher.Execute();
}

private void UpdateDirection()
{
    _animator.SetFloat("SpeedMultiplier", SpeedMultiplier);
    _animator.SetFloat("Speed", Speed);

    lock (_lock)
    {
        if (TargetDirection > Direction)
        {
            Direction += 0.1f;
        }
        if (TargetDirection < Direction)
        {
            Direction -= 0.1f;
        }

        _animator.SetFloat("Direction", Direction);
    }
}
```

După cum se vede, din codul de mai sus, operațiile din metoda de FixedUpdate sunt ținute la minimul necesar, pentru a influența cât mai puțin funcționarea sistemului per total. În FixedUpdate este apelată metoda de corectare a direcției către direcția țintă. Individul este pus în mișcare, și este direcționat, prin setarea parametrilor animatorului.

3.2. Managementul căii de navigare

După cum s-a menționat anterior, fiecare individ are propriul thread care se ocupă cu managementul căii de navigare. Acest thread este pornit la primirea notificării de incendiu, după calculul căii de navigare, în starea Start/Idle, și este oprit în una din stările Checkpoint/Trapped/Dead (vezi Figura 2.8).

Practic, rețeaua neuronală ia decizii în funcție de starea bot-ului față de punctul de referință din calea de navigare, care este primul nod din cale. Calea de navigare este o listă de puncte de coordonate în spațiu, a căror urmărire duce la o zonă de siguranță.

Pentru consumarea nodurilor, sunt calculate colțurile unui dreptunghi cu centrul în punctul de coordonate al nodului. La actualizarea poziției bot-ului, se verifică dacă bot-ul este în aria dreptunghiului nodului de referință și să nu existe nici un obstacol între bot și punctul de referință. Dacă condiția este îndeplinită, atunci nodul este eliminat din listă. Limitele dreptunghiului pentru nodul de referință sunt calculate doar dacă este vorba de un nou nod de referință.

```
public bool IsInProximity(Vector3 point, Vector3 rectCenter, Vector3 direction,
                          float distance)
{
 var angleToCompare = Vector3.SignedAngle(_oldDirection, direction, Vector3.up);

 if (_lastCenter != rectCenter)
 {
    RectangleCorners(rectCenter, direction, angleToCompare);
 }

 var apd = TriangleAreaXZ(point, _frontCornerLeft, _backCornerLeft);
 var dpc = TriangleAreaXZ(point, _backCornerRight, _backCornerLeft);
 var cpb = TriangleAreaXZ(point, _frontCornerRight, _backCornerRight);
 var pba = TriangleAreaXZ(point, _frontCornerLeft, _frontCornerRight);

 if (apd + dpc + cpb + pba > _area)
 {
    return false;
 }
 else
 {
    return true;
 }
}
```

Pentru a verifica dacă bot-ul este în dreptunghi, se calculează suma ariilor dintre poziția bot-ului și fiecare latură a dreptunghiului și se compară cu aria dreptunghiului. Dacă suma ariilor este mai mare decât aria dreptunghiului, atunci bot-ul este în afara acestuia, altfel este pe muchii sau în interior.

3.3. Modul general de funcționare

3.3.1. Simularea

Simularea funcționează întocmai cum este prezentat în Figura 2.5, diagrama înfățișând pașii și ordinea în care sunt aceștia executați.

```
private void Update()
{
 UpdateFPS();

 if (_context.CanUseControls && Input.GetKeyDown(KeyCode.Escape))
 {
    controls.SetActive(!controls.activeSelf);
    if (_context.IsResultActive)
    {
           resultsCanvas.SetActive(!resultsCanvas.activeSelf);
           resultsController.SetActive(!resultsController.activeSelf);
    }
 }
}

private void FixedUpdate()
{
 Dispatcher.Execute();
}
```

În cazul simulării, metoda FixedUpdate doar apelează dispatcherul de task-uri. Execuția acestuia din urmă a fost pus în această metodă deoarece se dorește testarea apelului mai rar, iar metoda FixedUpdate este apelată de un număr fix de ori pe secundă, și anume de 50 de ori pe secundă. Metoda Update este apelată la fiecare frame, de cele mai multe ori acesta fiind mult mai mare decât 50. În timpul antrenării FPS-ul a oscilat între 200 și 600 de frame-uri pe secundă, acest lucru depinzând și de sistem.

În metoda Update se actualizează FPS-ul la distanța temporală de o secundă, și se verifică dacă este apăsată tasta escape (ESC) pentru a face vizibile controalele. Verificarea apăsării este necesară în această metodă deoarece e posibil ca uneori la apăsarea tastei escape să nu fie înregistrată comanda de afișare a controalelor.

3.3.2. Controalele în timpul simulării

La apăsarea tastei escape (ESC), vor apărea butoanele din Figura 3.7. Butoanele „Back To Main Menu” și „Quit” pot fi utilizate doar dacă simularea s-a încheiat, altfel sunt inactive. Pentru a opri simularea se apasă butonul „Stop” care va opri simularea după ce runda în curs s-a încheiat. Pentru ascunderea butoanelor se apasă tot tasta escape.

Figura 3.7. Controalele din timpul simulării.

Simularea are o cameră activă, numită „Main Camera”, care supraveghează clădirea din exterior. Fiecare bot are propria cameră care îl urmărește la fiecare pas. Schimbarea între aceste camere se face prin intermediul tastei SPACE. Pentru a se reveni la camera din exterior, se apasă tasta M.

3.4. Rezultate

3.4.1. Rezultatele rundei de simulare

După fiecare rundă de simulare sunt afișate rezultatele indivizilor, acestea constând în:

- nume;
- distanța parcursă;
- timp;
- status.

3.4.2. Rezultatele totale

Deoarece sunt mai multe runde pe simulare, devine dificil în a afișa rezultatele tuturor rundelor simulării până la momentul curent, rezultatele rundei curente fiind mult mai relevante. De aceea s-a decis reținerea rezultatelor tuturor rundelor și scrierea lor într-un fișier de rezultate, în format XML, ca în exemplul de mai jos.

```
<RUN Number="1">
  <BOT Name="Bot 0" Distance="71.22" Time="00:00:39" Status="SAFE" />
  <BOT Name="Bot 1" Distance="79.79" Time="00:00:44" Status="SAFE" />
  <BOT Name="Bot 2" Distance="211.95" Time="00:01:53" Status="SAFE" />
  <BOT Name="Bot 3" Distance="143.76" Time="00:01:10" Status="SAFE" />
  <BOT Name="Bot 4" Distance="38.51" Time="00:00:21" Status="SAFE" />
  <BOT Name="Bot 5" Distance="95.77" Time="00:00:50" Status="DEAD" />
  <BOT Name="Bot 6" Distance="47.81" Time="00:00:24" Status="DEAD" />
  <BOT Name="Bot 7" Distance="66.95" Time="00:00:36" Status="SAFE" />
  <BOT Name="Bot 8" Distance="50.37" Time="00:00:28" Status="SAFE" />
  <BOT Name="Bot 9" Distance="82.80" Time="00:00:46" Status="SAFE" />
</RUN>
```

Capitolul 4. Testare

4.1. Antrenarea

În prima fază a antrenării, neexistând date pentru rețeaua neuronală, a fost necesară rularea unei simulări cu 100 de runde și cu un singur individ care să antreneze, pentru adunarea de date, astfel încât acestea să fie folosite pentru a avea boți care să nu antreneze, ci doar să simuleze, astfel încât sa poată fi realizată o antrenare completă.

Având date pentru construirea unei rețele neuronale care să poată fi capabilă să ia decizii cât de cât bune, se poate realiza antrenarea unei rețele neuronale, de la 0, într-o simulare în care să fie mai mulți boți, astfel încât să se poată învăța și interacțiunea dintre boți. Pentru această antrenare se recomandă a folosi un număr de boți care să ducă la interacțiuni cât mai frecvente, de exemplu 20 (din 40, numărul total). Numărul de runde, pentru o antrenare completă, este recomandat să fie de la 500 în sus.

Pentru antrenare trebuie bifată căsuța pentru antrenare, din meniul principal, din configurări, care asigură faptul că primul bot o să realizeze antrenarea. Atenție la faptul că la finalul simulării, atunci când se antrenează, fișierul cu date pentru rețeaua neuronală este înlocuit de unul nou cu noile date de antrenare.

4.2. Testarea propriu zisă

După antrenarea rețelei neuronale, se vor putea rula simulări în care se vor putea configura:

- numărul de indivizi, până la un maxim de 40 – având în vedere că fiecare individ necesită două thread-uri, la 40 de indivizi vor fi 80 de thread-uri + thread-ul principal, astea fiind doar cele cunoscute.
- numărul de runde pe simulare – de la 1 până la 2000.
- timpul de afișare a rezultatelor.
- dimensiunea nodurilor de foc și viteza de propagare.
- volumul sunetului de intro și a sunetului alarmei.
- viteza de mers a indivizilor – de menționat este faptul că antrenarea se face cu viteza de 1. Există posibilitatea, ca în urma modificării acest parametru, să apară inconsistențe în rularea simulării, prin asta făcându-se referire la posibile decizii întârziate din partea individului.

Având în vedere spațiul mare al stărilor, există posibilitatea ca un individ să ajungă într-o situație neprevazută, dar cu o antrenare destul de consistentă acest lucru ar trebui să fie destul de improbabil. În rulările efectuate nu au fost întâlnite astfel de situații.

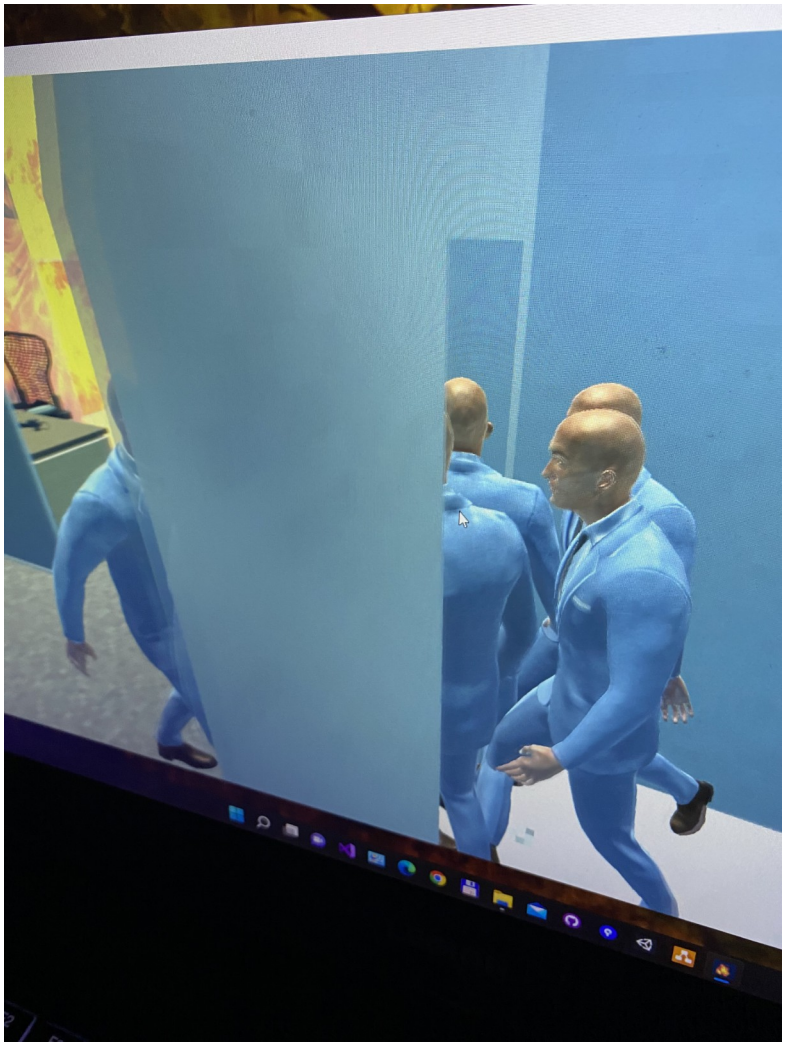

Figura 4.1. Blocaj în timpul antrenării.

Un lucru important de menționat este faptul că în timpul antrenării, s-a întâmplat ca boții să se blocheze într-o ușă mică, strâmtă. Unii boți se întorcea din drum după detectarea flăcărilor, practic recalcularea căii de navigare, iar unii încercau să treacă pe ușă în sensul opus acestora, neștiind de faptul că în acea direcție așteaptă pericolul. Acest lucru pune accent pe faptul că boții nu comunică între ei, fapt pentru care s-a și produs blocajul din Figura 4.1.

Concluzii

La finalizarea proiectului, în special al părții practice, din punct de vedere al obiectivelor propuse – obținerea unui sistem de evacuare cu indivizi independenți – putem spune cu siguranță că au fost atinse.

Subiectiv, din punctul de vedere al unui nou intrus în domeniul Deep Learning, se poate spune că rezultatele sunt fenomenale, indivizii fiind capabili să evacueze cu succes clădirea în condițiile propice, adică fără să fie blocați sau luați prin surprindere de flăcări.

Obiectiv, sistemul de evacuare mai poate fi îmbunătățit cu siguranță, și putem puncta:

- implementarea unui sistem de notificare/convorbire între indivizi pentru a putea anunța la întâlnire dacă o cale devine indisponibilă;
- utilizarea unei rețele neuronale recurente, pentru a realiza și controlul vitezei de deplasare, în ieșiri separate, și eventual pentru disponibilitatea alergării acolo unde este posibil;
- posibilitatea configurării separate a fiecărui bot în parte, astfel fiind deschisă opțiunea la boți cu viteze de deplasare diferite;
- expansiunea flăcărilor într-un mod mai realist, sau eventual opțiunea de a avea multiple moduri;
- ajustarea elastică a camerelor care urmăresc indivizii.

Proiectul poate fi considerat ca o primă etapă, având în vedere faptul că, deși într-o manieră simplă, și-a atins obiectivul, capabilitatea Deep Learning în acest context fiind demonstrată tocmai prin faptul că sistemul este funcțional și indivizii lucrează în obiectivul evacuării..

Bibliografie

[1] Bogdan Andrei G, Reinforcement learning pentru începători — algoritm pentru o mașină de taxi autonomă [Online], Disponibil la adresa: https://bogdanandreig.medium.com/reinforcement-learning-pentru-%C3%AEncep%C4%83tori-algoritm-pentru-o-ma%C8%99in%C4%83-de-taxi-autonom%C4%83-5ffbf4e81e7f, Accesat: 2021.

[0] samishawl, Epsilon-Greedy Algorithm in Reinforcement Learning [Online], Disponibil la adresa: https://www.geeksforgeeks.org/epsilon-greedy-algorithm-in-reinforcement-learning/, Accesat: 2020.

[2] Necunoscut, A* search algorithm [Online], Disponibil la adresa: https://en.wikipedia.org/wiki/A*_search_algorithm, Accesat: 2021.

[3] Andrei , Algoritmi euristici de explorare a grafurilor. A* [Online], Disponibil la adresa: https://andrei.clubcisco.ro/2pa/labs2012/Lab%2011%20Astar.pdf, Accesat: 2011.

[4] Ștefănescu Marian, Introducere în rețele neuronale – Teorie și aplicații [Online], Disponibil la adresa: https://code-it.ro/introducere-in-retele-neuronale-teorie-si-aplicatii/#wn, Accesat: 2018.

[5] Wikipedia, Rețea neuronală [Online], Disponibil la adresa: https://ro.wikipedia.org/wiki/Re%C8%9Bea_neural%C4%83, Accesat: 2021.

[6] IBM Cloud Education, Deep Learning [Online], Disponibil la adresa: https://www.ibm.com/cloud/learn/deep-learning, Accesat: 2020.

[7] Ankit Choudhary, A Hands-On Introduction to Deep Q-Learning using OpenAI Gym in Python [Online], Disponibil la adresa: https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/, Accesat: 2019.

[0] Chris Yoon, Double Deep Q Networks [Online], Disponibil la adresa: https://towardsdatascience.com/double-deep-q-networks-905dd8325412, Accesat: 2019.

Anexe.

Anexa 1. Deep Q Network

```
#include "pch.h"
#include "DeepQ.h"

DeepQ::DeepQ(NeuralNetwork* neuralNetwork)
{
 // srand Call for future randoms
 srand(static_cast<unsigned int>(time(NULL)));

 // Create the main network
 this->mainNetwork = std::shared_ptr<NeuralNetwork>(new
NeuralNetwork(*neuralNetwork));
 this->targetNetwork = std::shared_ptr<NeuralNetwork>(new
NeuralNetwork(*neuralNetwork));
}

DeepQ::~DeepQ()
{
}

void DeepQ::synchronizeTargetNetwork()
{
 this->targetNetwork->copyWights(*(this->mainNetwork));
}

void DeepQ::checkPercentageValue(double value)
{
 if (value < 0 && value > 1)
 {
    throw std::invalid_argument("Value must be between 0 and 1.");
 }
}

bool DeepQ::explorePolicy(double probability)
{
 bool shouldExplore = probability <= this->exploration;
 this->updateExploration();

 return shouldExplore;
}

void DeepQ::updateExploration()
{
 if (this->exploration > this->minimumExploration)
 {
```

```
        this->exploration *= this->explorationDecay;
    }
}

int DeepQ::getAction(std::vector<double> state)
{
    // Select action using epsilon greedy policy
    auto explorationProbability = static_cast<double>(rand() /
(static_cast<double>(RAND_MAX / EXPLORATION_UPPER_LIMIT)));
    if (this->explorePolicy(explorationProbability))
    {
        // Chose a random action for exploration
        return rand() % mainNetwork->outputSize();
    }
    else
    {
        // Get a prediction using the network
        auto results = mainNetwork->feedForward(state);
        return std::max_element(results.begin(), results.end()) - results.begin();
    }
}

void DeepQ::remember(std::vector<double> initialState, int action,
std::vector<double> reachedState, double reward, bool gameOver)
{
    if (this->experienceMemory.size() == this->experienceMemorySize)
    {
        this->experienceMemory.pop_front();
    }

    this->experienceMemory.push_back({ initialState, action, reachedState, reward,
gameOver });
}

void DeepQ::replayExperience()
{
    // Check to see if there is experience
    if (experienceMemory.size() == 0)
    {
        return;
    }

    auto memorySize = replayBatchSize;
    if (experienceMemory.size() < replayBatchSize)
    {
        memorySize = experienceMemory.size();
    }

    // Generate indexes
    std::default_random_engine engine;
    std::uniform_int_distribution<size_t> distribution(0, experienceMemory.size() -
```

```
1);

 std::vector<int> indexes;
 for (size_t i = 0; i < memorySize; ++i)
 {
    int index;
    do
    {
          index = distribution(engine);
    } while (std::find(indexes.begin(), indexes.end(), index) != indexes.end());

    indexes.push_back(index);
 }

 std::vector<std::tuple<std::vector<double>, std::vector<double>>> batch;

 // Get batch of training data
 for (auto i : indexes)
 {
    auto mainActionValues = mainNetwork-
>feedForward(experienceMemory[i].initialState);
    auto targetActionValues = targetNetwork-
>feedForward(experienceMemory[i].reachedState);

    // Calculate temporal target
    auto max = *std::max_element(targetActionValues.begin(),
targetActionValues.end());
    if (experienceMemory[i].gameOver)
    {
          mainActionValues[experienceMemory[i].action] =
experienceMemory[i].reward;
    }
    else
    {
          mainActionValues[experienceMemory[i].action] =
experienceMemory[i].reward + gamma * max;
    }

    batch.push_back(std::make_tuple(experienceMemory[i].initialState,
mainActionValues));
 }

 // Train main network for a number of epochs
 for (size_t epoch = 0; epoch < this->trainEpochs; ++epoch)
 {
    for (auto data : batch)
    {
          mainNetwork->feedForward(std::get<0>(data));
          mainNetwork->backPropagate(std::get<1>(data));
    }
 }
```

```
 // Synchronize target
 if (this->stepCounter % this->updateTargetWeightsAfterSteps == 0)
 {
    this->synchronizeTargetNetwork();
 }

 this->stepCounter++;
}

void DeepQ::setExploration(double value)
{
 this->checkPercentageValue(value);
 this->exploration = value;
}

void DeepQ::setMinimumExploration(double value)
{
 this->checkPercentageValue(value);
 this->minimumExploration = value;
}

void DeepQ::setExplorationDecay(double value)
{
 this->checkPercentageValue(value);
 this->explorationDecay = value;
}

void DeepQ::setGamma(double gamma)
{
 this->checkPercentageValue(gamma);
 this->gamma = gamma;
}

void DeepQ::setTrainEpochs(unsigned int trainEpochs)
{
 this->trainEpochs = trainEpochs;
}

void DeepQ::setUpdateTargetWeightsAfterSteps(unsigned int steps)
{
 this->updateTargetWeightsAfterSteps = steps;
}

void DeepQ::setReplayBatchSize(unsigned int numberOfSamples)
{
 this->replayBatchSize = numberOfSamples;
}

void DeepQ::setExperienceMemorySize(unsigned int size)
{
```

```
 this->experienceMemorySize = size;
}

void DeepQ::save(std::string path)
{
 this->mainNetwork->save(path);
}
```

Anexa 2. Calculul limitelor dreptunghiului pentru nodul de referință

```
using UnityEngine;

namespace Assets.Scripts.Prefabs.Bot.States.Constants
{
    class ProximityValidator
    {
        private Vector3 _frontCornerRight;
        private Vector3 _frontCornerLeft;
        private Vector3 _backCornerRight;
        private Vector3 _backCornerLeft;

        private float _area;

        private readonly float _lowWidth = 0.5f;
        private readonly float _highWidth = 0.8f;
        private readonly float _length = 2f;

        private Vector3 _oldDirection = Vector3.zero;
        private Vector3 _lastCenter = Vector3.zero;

        BotController _bot;
        //private int _botLayerMask = ~(1 << 10);
        private LayerMask _rectLayerMask = ~(LayerMask.GetMask("Fire") |
LayerMask.GetMask("Bot"));

        public ProximityValidator(BotController bot)
        {
            _bot = bot;
        }

        public bool IsInProximity(Vector3 point, Vector3 rectCenter, Vector3
direction, float distance)
        {
            var angleToCompare = Vector3.SignedAngle(_oldDirection, direction,
Vector3.up);

            if (_lastCenter != rectCenter)
            {
                RectangleCorners(rectCenter, direction, angleToCompare);
```

```
            }

            var apd = TriangleAreaXZ(point, _frontCornerLeft, _backCornerLeft);
            var dpc = TriangleAreaXZ(point, _backCornerRight, _backCornerLeft);
            var cpb = TriangleAreaXZ(point, _frontCornerRight, _backCornerRight);
            var pba = TriangleAreaXZ(point, _frontCornerLeft, _frontCornerRight);

            //_bot.Dispatcher.Schedule(() =>
            //{
            //    UnityEngine.Debug.DrawLine(_frontCornerLeft, _frontCornerRight,
Color.blue, 0.3f);
            //    UnityEngine.Debug.DrawLine(_frontCornerRight, _backCornerRight,
Color.red, 0.3f);
            //    UnityEngine.Debug.DrawLine(_backCornerRight, _backCornerLeft,
Color.yellow, 0.3f);
            //    UnityEngine.Debug.DrawLine(_backCornerLeft, _frontCornerLeft,
Color.red, 0.3f);
            //});

            if (apd + dpc + cpb + pba > _area)
            {
                return false;
            }
            else
            {
                return true;
            }
        }

        private void RectangleCorners(Vector3 center, Vector3 direction, float
angleToCompare)
        {
            var directionToUse = direction.normalized;

            var width = _highWidth;
            var angleWidth = 0f;

            float leftDistance = 0;
            float rightDistance = 0;
            bool hitLeft = false;
            bool hitRight = false;

            if (_oldDirection != Vector3.zero)
            {
                if ((int)Mathf.Abs(angleToCompare) == 90)
                {
                    angleWidth = (-1f * angleToCompare) / 2f;

                    // Update direction
                    directionToUse = Quaternion.AngleAxis(angleWidth, Vector3.up)
* directionToUse;
```

```
                    _bot.Dispatcher.Schedule(() =>
                    {
                        switch ((int)angleWidth)
                        {
                            case 45:
                                hitLeft = Physics.Raycast(origin: center,
direction: Quaternion.AngleAxis(-90, Vector3.up) * directionToUse, hitInfo: out
var hitMiddleLeft, maxDistance: 0.75f, layerMask: _rectLayerMask);
                                leftDistance = hitMiddleLeft.distance;
                                break;
                            case -45:
                                hitRight = Physics.Raycast(origin: center,
direction: Quaternion.AngleAxis(90, Vector3.up) * directionToUse, hitInfo: out var
hitMiddleRight, maxDistance: 0.75f, layerMask: _rectLayerMask);
                                rightDistance = hitMiddleRight.distance;
                                break;
                        }
                    }).WaitOne();
                }
            }


            if (hitLeft || hitRight)
                width = _lowWidth;

            var mfront = center + directionToUse * width;
            var mback = center - directionToUse * width;

            var directionLeft = Quaternion.AngleAxis(-90, Vector3.up) *
directionToUse;
            var directionRight = Quaternion.AngleAxis(90, Vector3.up) *
directionToUse;

            float distanceFrontLeft = 0;
            float distanceFrontRight = 0;
            float distanceBackLeft = 0;
            float distanceBackRight = 0;

            _bot.Dispatcher.Schedule(() =>
            {
                if (!hitLeft)
                {
                    Physics.Raycast(origin: mfront, direction: directionLeft,
hitInfo: out var hitFrontLeft, maxDistance: _length, layerMask: _rectLayerMask);
                    distanceFrontLeft = hitFrontLeft.distance;
                    Physics.Raycast(origin: mback, direction: directionLeft,
hitInfo: out var hitBackLeft, maxDistance: _length, layerMask: _rectLayerMask);
                    distanceBackLeft = hitBackLeft.distance;
                }
```

```
                if (!hitRight)
                {
                    Physics.Raycast(origin: mfront, direction: directionRight,
hitInfo: out var hitFrontRight, maxDistance: _length, layerMask: _rectLayerMask);
                    distanceFrontRight = hitFrontRight.distance;
                    Physics.Raycast(origin: mback, direction: directionRight,
hitInfo: out var hitBackRight, maxDistance: _length, layerMask: _rectLayerMask);
                    distanceBackRight = hitBackRight.distance;
                }
            }).WaitOne();

            distanceFrontLeft = distanceFrontLeft == 0 ? _length :
distanceFrontLeft;
            distanceFrontRight = distanceFrontRight == 0 ? _length :
distanceFrontRight;
            distanceBackLeft = distanceBackLeft == 0 ? _length : distanceBackLeft;
            distanceBackRight = distanceBackRight == 0 ? _length :
distanceBackRight;

            if (leftDistance == 0)
                leftDistance = distanceBackLeft < distanceFrontLeft ?
distanceBackLeft : distanceFrontLeft;

            if (rightDistance == 0)
                rightDistance = distanceBackRight < distanceFrontRight ?
distanceBackRight : distanceFrontRight;

            if(leftDistance < 0.5 && rightDistance < 0.5)
            {
                leftDistance = 2f;
                rightDistance = 2f;
            }

            _frontCornerLeft = mfront + directionLeft * leftDistance;
            _frontCornerRight = mfront + directionRight * rightDistance;
            _backCornerLeft = mback + directionLeft * leftDistance;
            _backCornerRight = mback + directionRight * rightDistance;

            _area = Vector3.Distance(_frontCornerLeft, _frontCornerRight) *
Vector3.Distance(_frontCornerRight, _backCornerRight);

            _lastCenter = center;
            _oldDirection = direction.normalized;
        }

        private float TriangleAreaXZ(Vector3 a, Vector3 b, Vector3 c)
        {
            var ab = Mathf.Sqrt(Mathf.Pow((a.x - b.x), 2) + Mathf.Pow((a.z - b.z),
2));
            var bc = Mathf.Sqrt(Mathf.Pow((b.x - c.x), 2) + Mathf.Pow((b.z - c.z),
2));
```

```
            var ca = Mathf.Sqrt(Mathf.Pow((c.x - a.x), 2) + Mathf.Pow((c.z - a.z),
2));
            var s = (ab + bc + ca) / 2;
            return Mathf.Sqrt(s * (s - ab) * (s - bc) * (s - ca));
        }
    }
}
```

Anexa 3. Preluarea observațiilor

```
private void GetObservation()
{
    _context.Observation = new double[Locals.OBSERVATION_SIZE];
    float angle = 0;
    float distance = 0;

    _context.Bot.Dispatcher.Schedule(() =>
    {
        if (Vector3.Distance(_context.Bot.transform.position,
_context.GoalPosition) > 7)
        {
            lock (_context.NavigationLock)
            {
                if (_context.NavigationPath != null &&
_context.NavigationPath.Count >= 2)
                {
                    // Calculate angles
                    var direction = _context.NavigationPath[0].Position -
_context.Bot.transform.position;
                    angle = Vector3.SignedAngle(_context.Bot.transform.forward,
direction, Vector3.up);

                    // Calculate distances
                    distance = Vector3.Distance(_context.Bot.transform.position,
_context.NavigationPath[0].Position);
                }
            }
        }
        else
        {
            // Calculate angles
            var direction = _context.GoalPosition -
_context.Bot.transform.position;
            angle = Vector3.SignedAngle(_context.Bot.transform.forward, direction,
Vector3.up);

            // Calculate distances
            distance = Vector3.Distance(_context.Bot.transform.position,
_context.GoalPosition);
```

```
        }

        // Get proximities
        // Create commands for horizontal rays
        var middlePoint = _context.Bot.transform.position + _raysMiddlePoint;
        var bottomPoint = _context.Bot.transform.position + _raysStartingPoint -
Vector3.up * _raysColliderDistanceToPoints;

        var results = new
NativeArray<RaycastHit>(Locals.OBSERVATION_TOTAL_NUMBER_OF_RAYS,
Allocator.TempJob);
        var commands = new
NativeArray<RaycastCommand>(Locals.OBSERVATION_TOTAL_NUMBER_OF_RAYS,
Allocator.TempJob);

        for (var i = 0; i < Locals.OBSERVATION_NUMBER_OF_RAYS; ++i)
        {
            var direction =
Quaternion.AngleAxis(Locals.OBSERVATION_START_ANGLE_OF_RAYS +
Locals.OBSERVATION_OFFSET_ANGLE_OF_RAYS * i, _context.Bot.transform.up) *
                _context.Bot.transform.forward;

            commands[i] = new RaycastCommand(middlePoint, direction, distance:
Locals.OBSERVATION_MAX_DISTANCE_OF_RAYS, layerMask: _botLayerMask);
            commands[Locals.OBSERVATION_NUMBER_OF_RAYS + i] = new
RaycastCommand(bottomPoint, direction, distance:
Locals.OBSERVATION_MAX_DISTANCE_OF_RAYS, layerMask: _botLayerMask);
        }

        var handle = RaycastCommand.ScheduleBatch(commands, results, 1);
        handle.Complete();

        // Write proximities to observation
        for (var i = 0; i < Locals.OBSERVATION_TOTAL_NUMBER_OF_RAYS; ++i)
        {
            // Get distance and fire
            if (results[i].collider != null)
            {
                _context.Observation[i] = results[i].distance;
                switch (results[i].collider.tag)
                {
                    case "Fire":

_context.Observation[Locals.OBSERVATION_TOTAL_NUMBER_OF_RAYS + i] = 2;
                        break;
                    case "Bot":

_context.Observation[Locals.OBSERVATION_TOTAL_NUMBER_OF_RAYS + i] = 1;
                        break;
                    default:
```

```
_context.Observation[Locals.OBSERVATION_TOTAL_NUMBER_OF_RAYS + i] = 0;
                        break;
                }
            }
            else
            {
                _context.Observation[i] = Locals.OBSERVATION_MAX_DISTANCE_OF_RAYS;
            }
        }

        // Free buffers
        results.Dispose();
        commands.Dispose();

        // Prepare data for collision
        var startCapsule = _context.Bot.transform.position +
_startCapsuleColliderConstant;
        var endCapsule = _context.Bot.transform.position +
_endCapsuleColliderConstant;
        var colliders = Physics.OverlapCapsule(startCapsule, endCapsule,
_radiusCapsuleCollider);

        // Get collision
        foreach (var collider in colliders)
        {
            if (collider.name != _context.Bot.Name)
            {
                switch (collider.tag)
                {
                    case "Bot":

_context.Observation[Locals.OBSERVATION_COLLISION_WITH_BOT_INDEX] = 1;
                        break;
                    case "Fire":
                        _context.IsDead = true;

_context.Observation[Locals.OBSERVATION_COLLISION_WITH_FIRE_INDEX] = 1;
                        break;
                    default:

_context.Observation[Locals.OBSERVATION_COLLISION_WITH_ANYTHING_ELSE_INDEX] = 1;
                        break;
                }
            }
        }

        // Get angle from goal
        _context.Observation[Locals.OBSERVATION_ANGLE_FROM_GOAL_INDEX] = angle;
        // Get angle velocity
        _context.Observation[Locals.OBSERVATION_ANGLE_VELOCITY_INDEX] =
_context.Bot.Direction;
```

```
        // Get distance
        _context.Observation[Locals.OBSERVATION_DISTANCE_INDEX] = distance;

        // Update distance
        _context.AddDistanceWalked(_context.Bot.transform.position);
    }).WaitOne();

    // Check if bot reached checkpoint
    var distanceFromCheckpoint = Vector3.Distance(_context.BotPosition,
_context.GoalPosition);
    if (!_context.RandomGoalPositionSet && distanceFromCheckpoint < 7)
    {
        var center = _context.GoalPosition;
        _context.Bot.Dispatcher.Schedule(() =>
        {
            do
            {
                _context.GoalPosition = center + (Vector3)(7 *
UnityEngine.Random.insideUnitCircle);
            } while (Physics.CheckSphere(_context.GoalPosition,
_radiusCapsuleCollider));
        }).WaitOne();
        _context.RandomGoalPositionSet = true;
    }

    if (_context.RandomGoalPositionSet)
    {
        _context.ObservationsAroundGoal++;
        if (_context.ObservationsAroundGoal >= 200 || distanceFromCheckpoint <=
0.5)
        {
            _context.IsCheckpointReached = true;
        }
    }

}
```

*Anexa 4. A**

```
using System.Collections.Generic;
using System.Threading;
using UnityEngine;

namespace Assets.Scripts.Utils.AStar
{
    static class Navigator
    {
        public static bool IsGenerated { get; set; }
        public static Vector3 WorldBottomLeft { get; set; }
```

```
        public static Vector3 GridWorldSize { get; set; }
        public static Node[,,] Grid { get; set; }
        public static int GridSizeX { get; set; }
        public static int GridSizeY { get; set; }
        public static int GridSizeZ { get; set; }
        public static float NodeDiameter { get; set; }

        private static readonly ReaderWriterLock _lock = new ReaderWriterLock();
        private static List<(int x, int y, int z)> _nodesInvalidated = new
List<(int x, int y, int z)>();

        public static List<Node> FindPath(Vector3 startPos, Vector3 targetPos)
        {
            if(Grid == null)
            {
                return null;
            }

            try
            {
                _lock.AcquireReaderLock(Timeout.Infinite);

                var tracer = new PathRetracer();
                var containers = new Dictionary<Node, AStarNodeContainer<Node>>();
                Node startNode = NodeFromWorldPoint(startPos);
                Node targetNode = NodeFromWorldPoint(targetPos);

                if(startNode == null)
                {
                    return null;
                }

                if (targetNode == null)
                {
                    return null;
                }

                var openSet = new Heap<AStarNodeContainer<Node>>((uint)(GridSizeX
* GridSizeY));
                var closedSet = new HashSet<Node>();

                var startContainer = new AStarNodeContainer<Node>(startNode);
                openSet.Add(startContainer);

                while (openSet.Count > 0)
                {
                    var currentNodeContainer = openSet.RemoveFirst();
                    var currentNode = currentNodeContainer.Data;
                    closedSet.Add(currentNode);

                    if (currentNode == targetNode)
```

```
                    {
                        break;
                    }

                    foreach (Node neighbour in Neighbours(currentNode))
                    {
                        AStarNodeContainer<Node> neighbourContainer;
                        if (containers.ContainsKey(neighbour))
                        {
                            neighbourContainer = containers[neighbour];
                        }
                        else
                        {
                            neighbourContainer = new
AStarNodeContainer<Node>(neighbour);
                            containers.Add(neighbour, neighbourContainer);
                        }

                        float cost = currentNodeContainer.G +
Distance(currentNode, neighbour) * neighbour.GridY;

                        if (closedSet.Contains(neighbour) && neighbourContainer.G
< cost)
                        {
                            continue;
                        }

                        if (cost < neighbourContainer.G || !
openSet.Contains(neighbourContainer))
                        {
                            neighbourContainer.G = cost;
                            neighbourContainer.H = Distance(neighbour, targetNode)
;
                            tracer.Add(neighbour, currentNode);

                            if (!openSet.Contains(neighbourContainer))
                            {
                                openSet.Add(neighbourContainer);
                            }
                        }
                    }
                }

                var path = tracer.RetraceBack(targetNode, startNode);

                return path;
            }
            finally
            {
                _lock.ReleaseReaderLock();
            }
```

```
        }

        private static float Distance(Node a, Node b)
        {
            return Mathf.Sqrt(Mathf.Pow((b.Position.x - a.Position.x), 2) +
Mathf.Pow((b.Position.y - a.Position.y), 2) + Mathf.Pow((b.Position.z -
a.Position.z), 2));
        }

        public static List<Node> Neighbours(Node node)
        {
            var result = new List<Node>();

            for (var x = -1; x <= 1; x++)
            {
                for (var y = 1; y >= -1; y--)
                {
                    for (var z = -1; z <= 1; z++)
                    {
                        if (x == 0 && y == 0 && z == 0)
                        {
                            continue;
                        }

                        int checkX = node.GridX + x;
                        int checkY = node.GridY + y;
                        int checkZ = node.GridZ + z;

                        if (checkX >= 0
                            && checkX < GridSizeX
                            && checkY >= 0
                            && checkY < GridSizeY
                            && checkZ >= 0
                            && checkZ < GridSizeZ
                            && Grid[checkX, checkY, checkZ] != null
                            && Grid[checkX, checkY, checkZ].Walkable)
                        {
                            result.Add(Grid[checkX, checkY, checkZ]);
                        }
                    }
                }
            }

            return result;
        }

        public static List<Node> Neighbours(int node_x, int node_y, int node_z)
        {
            var result = new List<Node>();
```

```
            for (var x = -1; x <= 1; x++)
            {
                for (var y = 3; y >= 0; y--)
                {
                    for (var z = -1; z <= 1; z++)
                    {
                        if (x == 0 && y == 0 && z == 0)
                        {
                            continue;
                        }

                        int checkX = node_x + x;
                        int checkY = node_y + y;
                        int checkZ = node_z + z;

                        if (checkX >= 0
                            && checkX < GridSizeX
                            && checkY >= 0
                            && checkY < GridSizeY
                            && checkZ >= 0
                            && checkZ < GridSizeZ
                            && Grid[checkX, checkY, checkZ] != null
                            && Grid[checkX, checkY, checkZ].Walkable)
                        {
                            result.Add(Grid[checkX, checkY, checkZ]);
                        }
                    }
                }
            }

            return result;
        }

        public static Node NodeFromWorldPoint(Vector3 worldPosition)
        {
            (int x, int y, int z) = GetIndexInGrid(worldPosition);

            if (Grid[x, y, z] == null || !Grid[x, y, z].Walkable)
            {
                foreach (Node neighbour in Neighbours(x, y, z))
                {
                    return neighbour;
                }
            }

            return Grid[x, y, z];
        }

        public static void SetNodesUnwalkable(Vector3 position, float radius)
        {
            try
```

```
            {
                _lock.AcquireWriterLock(Timeout.Infinite);

                (int x, int y, int z) = GetIndexInGrid(position);

                var nodesPerRadius = Mathf.CeilToInt(radius / NodeDiameter +
NodeDiameter);

                for (var idx = -nodesPerRadius; idx <= nodesPerRadius; ++idx)
                {
                    for (var idy = -nodesPerRadius; idy <= nodesPerRadius; ++idy)
                    {
                        for (var idz = -nodesPerRadius; idz <= nodesPerRadius; +
+idz)
                        {
                            int checkX = x + idx;
                            int checkY = y + idy;
                            int checkZ = z + idz;

                            if (checkX >= 0
                                && checkX < GridSizeX
                                && checkY >= 0
                                && checkY < GridSizeY
                                && checkZ >= 0
                                && checkZ < GridSizeZ
                                && Grid[checkX, checkY, checkZ] != null)
                            {
                                Grid[checkX, checkY, checkZ].Walkable = false;
                                _nodesInvalidated.Add((checkX, checkY, checkZ));
                            }
                        }
                    }
                }
            }
            finally
            {
                _lock.ReleaseWriterLock();
            }
        }

        public static void ResetGrid()
        {
            try
            {
                _lock.AcquireWriterLock(Timeout.Infinite);

                foreach (var index in _nodesInvalidated)
                {
                    Grid[index.x, index.y, index.z].Walkable = true;
                }
```

```
                _nodesInvalidated = new List<(int x, int y, int z)>();
            }
            finally
            {
                _lock.ReleaseWriterLock();
            }
        }

        private static (int x, int y, int z) GetIndexInGrid(Vector3 position)
        {
            float percentX = Mathf.Abs(position.x - WorldBottomLeft.x) * 100 /
GridWorldSize.x / 100;
            float percentY = Mathf.Abs(position.y - WorldBottomLeft.y) * 100 /
GridWorldSize.y / 100;
            float percentZ = Mathf.Abs(position.z - WorldBottomLeft.z) * 100 /
GridWorldSize.z / 100;

            int x = Mathf.RoundToInt((GridSizeX - 1) * percentX);
            int y = Mathf.RoundToInt((GridSizeY - 1) * percentY);
            int z = Mathf.RoundToInt((GridSizeZ - 1) * percentZ);

            return (x, y, z);
        }
    }
}
```