

**Universitatea Tehnică a Moldovei**

**Crearea unui joc video interactiv folosind tehnici si algoritmi ai intelegentei artificiale**

**„FINAL GROUNDS”**

|  |  |
| --- | --- |
| **Student:** | **gr. TI-236, TI-231,**  **Surdu Nichita**  **Bujoreanu Cristian Strelnicom Dan** |
| **Coordonator:** | **Andrievschi-Bagrin Veronica, asist.univ.** |

**Chişinău, 2024**

**CUPRINS**

INTRODUCERE.......................................................................................................................................1

1 SISTEME SIMILARE ȘI ANALIZA LOR...........................................................................................2  
1.1 Jocuri populare bazate pe algoritmi similari.......................................................................................2

1.2 Caracteristici și comparații..................................................................................................................3

2 SCOPUL ȘI OBIECTIVELE JOCULUI...............................................................................................5  
2.1 Scopul proiectului...............................................................................................................................5  
2.2 Obiectivele și cerințele funcționale....................................................................................................5

3 DESCRIEREA ALGORITMILOR UTILIZAȚI..................................................................................8  
3.1 A\* (A-star).........................................................................................................................................8  
3.2 Algoritmul lui Dijkstra.......................................................................................................................9  
3.3 Breadth-First Search (BFS)...............................................................................................................11

3.4 Compararea algoritmilor...................................................................................................................12

4 UNITY ȘI INTEGRAREA ALGORITMILOR...................................................................................13  
4.1 Prezentarea Unity..............................................................................................................................13  
4.2 Implementarea algoritmilor în Unity................................................................................................14  
4.3 Testarea si ajustarea performanței în Unity......................................................................................15

5 IMPLEMENTAREA PROIECTULUI................................................................................................16  
5.1 Generarea hărții jocului....................................................................................................................16  
5.2 Spaunarea personajului principal și a inamicilor (zombie)...............................................................17

6 ANALIZA REZULTATELOR............................................................................................................19  
6.1 Performanța algoritmilor în diverse scenarii de joc..........................................................................19  
6.2 Optimizarea algoritmilor pentru Unity..............................................................................................19

7 DOCUMENTAREA PRODUSULUI..................................................................................................22

CONCLUZII...........................................................................................................................................24

BLIOGRAFIE.........................................................................................................................................25

ANEXE...................................................................................................................................................26  
7.1 Codul sursă.......................................................................................................................................27

### ****INTRODUCERE****

În ultimii ani, industria jocurilor video a evoluat semnificativ, devenind o componentă esențială a divertismentului digital și a culturii interactive. Jocurile video moderne nu doar că oferă o experiență captivantă pentru utilizatori, dar și utilizează tehnologii avansate pentru a crea lumi virtuale dinamice și realiste. Unul dintre cele mai importante domenii de cercetare în dezvoltarea jocurilor este inteligența artificială (AI), care contribuie la crearea unui comportament realist și provocator al personajelor controlate de computer. Aceste personaje, cunoscute și sub denumirea de NPC (non-player characters), trebuie să fie capabile să reacționeze într-un mod credibil și să interacționeze cu jucătorul, oferind astfel o experiență de joc mai imersivă și interesantă.

Proiectul „Final Grounds” reprezintă un joc video de tip „survival” dezvoltat în Unity, un motor de jocuri bine cunoscut pentru versatilitatea și performanțele sale. În acest joc, jucătorul este plasat într-un mediu plin de inamici, iar scopul său este de a naviga printr-o hartă complexă pentru a ajunge într-un punct central unde va întâlni un boss puternic. În această luptă, jucătorul trebuie să depășească numeroase obstacole și să confrunte inamicii, care sunt controlați de algoritmi de inteligență artificială.

În cadrul acestui proiect, am utilizat algoritmi de căutare pentru a determina mișcările și comportamentele inamicilor. Algoritmii de căutare sunt fundamentali în dezvoltarea comportamentului AI, deoarece aceștia permit NPC-urilor să găsească cele mai bune trasee în hărțile de joc, ținând cont de obstacolele și obiectivele din mediul virtual. Alegerea algoritmilor adecvați și implementarea acestora într-un mod eficient reprezintă un aspect crucial pentru performanța și credibilitatea unui joc.

Astfel, scopul acestui raport este de a prezenta procesul de dezvoltare a jocului „Final Grounds”, cu un accent deosebit pe utilizarea algoritmilor de căutare pentru a controla comportamentul inamicilor. În cadrul acestui raport, vom analiza diferitele metode de implementare a algoritmilor de AI în Unity și vom evalua eficiența fiecărei abordări în funcție de complexitatea hărții și cerințele jocului. De asemenea, vom discuta despre avantajele și dezavantajele fiecărui algoritm de căutare, precum și despre impactul lor asupra experienței jucătorului.

În plus, raportul va examina integrarea acestor algoritmi în Unity, un motor de jocuri popular care oferă un cadru robust pentru dezvoltarea jocurilor 2D și 3D. Unity permite implementarea rapidă a logicii de joc, a comportamentului AI și a interfețelor grafice, facilitând testarea și optimizarea aplicației. Vom explora și modul în care Unity poate fi folosit pentru a crea un mediu de joc interactiv, care să răspundă în mod dinamic acțiunilor jucătorului și să încorporeze feedback-ul în timp real.

Algoritmii de căutare, precum A\*, Dijkstra și BFS, sunt fundamentali pentru dezvoltarea jocurilor, în special în jocurile care implică mișcare strategică pe hărți complexe. În jocurile de tip survival, acești algoritmi ajută la simularea comportamentului NPC-urilor, facilitând luarea deciziilor pentru aceștia în timp real, în funcție de poziția jucătorului și a obstacolelor din jur.

**1**

### 1 SISTEME SIMILARE ȘI ANALIZA LOR

În această secțiune vom analiza câteva jocuri populare care utilizează algoritmi de căutare și navigare pentru a crea comportamente inteligente ale inamicilor, dar și modalitățile în care aceste tehnici pot fi aplicate și îmbunătățite. Comparând aceste jocuri cu proiectul nostru, putem identifica asemănările și diferențele esențiale, precum și provocările și soluțiile inovative.

**1.1 Jocuri populare bazate pe algoritmi similari**

Există o varietate de jocuri care utilizează algoritmi de căutare pentru a crea comportamente inteligente ale personajelor controlate de computer (NPC), inamicilor sau chiar aliaților. Mai jos sunt câteva exemple semnificative:

**Pac-Man:**

**Algoritm utilizat:** În jocul clasic Pac-Man, fantomele care vânează jucătorul folosesc un algoritm simplu de urmărire bazat pe distanță. Fiecare fantomă urmează un comportament predeterminat, în funcție de poziția jucătorului. Deși este un algoritm destul de simplu, el este eficient în crearea unei dinamici interesante pentru joc.

**Posibilități de îmbunătățire:** Deși algoritmul original nu utilizează căutare avansată a drumurilor, tehnici mai moderne precum **A\*** ar putea fi folosite pentru a crea un comportament mai "inteligent" al fantomelor, făcându-le să ia decizii mai complexe în privința rutei pe care o aleg pentru a-l prinde pe jucător, în funcție de poziția sa și de obstacolele din labirint.

**Plants vs Zombies:**

**Algoritm utilizat:** În acest joc, zombie-ii se mișcă pe o grilă de pătrățele, urmând un traseu simplu care le permite să ajungă la planta jucătorului. Algoritmii de căutare, cum ar fi **A\***, pot fi folosiți pentru a crea trasee mai eficiente pentru inamicii zombie, navigându-i printre obstacolele și plantele plasate de jucător.

**Posibilități de îmbunătățire:** În cazul jocului nostru, abordarea ar fi mult mai complexă, având în vedere că nu doar că zombie-ii trebuie să ajungă la jucător, dar și harta poate conține obstacole și terenuri care nu sunt simple „bucăți de pajiște”. Implementarea unor algoritmi de căutare a drumurilor ca **A\*** ar putea adăuga un nivel de dificultate mult mai mare și mai variat, având în vedere faptul că inamicii ar trebui să aleagă trasee prin zone cu multiple obstacole, nu doar printr-o grilă de pătrățele.

**2**

**Left 4 Dead:**

**Algoritm utilizat:** Left 4 Dead utilizează un AI complex cunoscut sub numele de „Director AI”. Acest sistem de inteligență artificială controlează comportamentele inamicilor (zombie-ii) și variază dificultatea jocului, plasând inamicii în locuri strategice pentru a crea tensiune și provocări constante. Algoritmii de căutare sunt folosiți pentru a ghida zombie-ii prin medii 3D complexe, care includ atât obstacole naturale (clădiri, obiecte) cât și constrângeri legate de obiectivele de joc.

**Posibilități de îmbunătățire**: În acest joc, un algoritm bazat pe grafuri pentru navigarea prin medii dinamice este folosit pentru a determina traseele optime ale inamicilor. În contrast cu **Left 4 Dead**, unde sistemul AI se bazează pe elemente mai largi de „control al experienței” ale jucătorilor, jocul nostru se concentrează pe o navigare directă folosind algoritmi de căutare a drumurilor. Totuși, utilizarea unor algoritmi de tip **A\*** sau **Dijkstra** pentru a permite zombie-ilor să se deplaseze pe o hartă complexă cu obstacole ar putea face experiența mai provocatoare, dar și mai variată.

**1.2 Caracteristici și comparații**

Comparativ cu jocurile menționate anterior, proiectul nostru „Final Grounds” are o abordare similară în privința implementării comportamentului inamicilor, dar diferențele sunt semnificative în ceea ce privește complexitatea AI-ului și a hărții. Iată o comparație detaliată între jocul nostru și cele menționate:

**Complexitatea hărții și a obstacolelor**:

**Pac-Man** folosește o hartă destul de simplă, unde obstacolele sunt statice și predeterminate. În schimb, în „Final Grounds”, harta este mult mai complexă, conținând multiple zone de teren cu obstacole care se schimbă și diferite rute de navigare. Așadar, algoritmii de căutare trebuie să fie mult mai flexibili și eficienți pentru a face față acestei complexități.

În comparație cu **Plants vs Zombies**, unde zombie-ii se mișcă pe o grilă statică, în jocul nostru AI-ul trebuie să învețe să navigheze prin hărți cu forme și topologii variate. Aceasta presupune implementarea unor algoritmi de căutare a drumurilor mai avansați, care să poată lua în considerare obstacolele, evitând rutele neeficiente și optimizând traseele pentru inamici.

**3**

**Interacțiunea dintre inamic și jucător**:

În **Left 4 Dead**, AI-ul urmărește o echipă de jucători și creează provocări dinamice, având în vedere pozițiile și comportamentele acestora. Deși este un joc cu mult mai multă interacțiune între AI și jucători, algoritmii de căutare sunt implementați pentru a crea comportamente inteligente ale inamicilor. În **Final Grounds**, însă, interacțiunea dintre inamicii zombie și jucător este mai directă, iar aceștia trebuie să navigheze eficient prin hărțile pline de obstacole pentru a ajunge la jucător și a-l înfrunta.

De asemenea, jocul nostru introduce un inamic final (boss), care nu doar că este mai puternic, dar și necesită o abordare diferită din partea inamicilor, iar AI-ul trebuie să adapteze comportamentul zombie-ilor într-un mod care să contribuie la construirea unui climax interesant al jocului.

**Nivelul de complexitate al algoritmilor**:

În **Plants vs Zombies**, AI-ul zombie-ilor este destul de simplu și nu impune multe cerințe pentru dezvoltatori, având în vedere că zombie-ii se deplasează pe un teren simplu și în linie dreaptă. În schimb, în „Final Grounds”, AI-ul trebuie să facă față unei hărți complexe, cu obstacole multiple, iar comportamentele inamicilor trebuie să fie reglate pentru a crea o experiență provocatoare și interesantă. Aceasta presupune implementarea unor algoritmi de căutare care să țină cont nu doar de distanță, ci și de obstacole și strategii de abordare a fiecărui tip de teren.

Comparativ cu jocurile menționate, proiectul nostru are o abordare similară, dar cu diferențe semnificative în ceea ce privește complexitatea AI-ului. Spre deosebire de **Plants vs Zombies**, unde zombie-ii se mișcă pe o grilă simplă, în jocul nostru, zombie-ii trebuie să navigheze prin hărți complexe, cu multiple obstacole, folosind algoritmi avansați de căutare a drumurilor.

**4**

**2 SCOPUL ȘI OBIECTIVELE JOCULUI**

**2.1 Scopul proiectului**

Scopul principal al acestui proiect este de a dezvolta un joc de supraviețuire bazat pe interacțiunea jucătorului cu inamicii și cu mediul înconjurător. Jucătorul este plasat pe o hartă vastă și complexă, plină de obstacole și inamici (zombie), având ca obiectiv final să ajungă în centrul hărții pentru a înfrunta un boss puternic.

Una dintre principalele provocări în acest joc este crearea unui comportament inteligent al zombie-ilor, care se va baza pe algoritmi de căutare precum **A\***, **Dijkstra** și **BFS** pentru a naviga prin harta de joc. Acești algoritmi vor fi utilizați pentru a simula modul în care zombie-ii se mișcă eficient pe harta plină de obstacole și cum aceștia pot urmări jucătorul într-o manieră dinamică și interesantă.

**Unity** a fost ales ca platformă de dezvoltare datorită versatilității sale și capacității de a implementa rapid funcționalități complexe, de la inteligența artificială până la crearea de medii 3D interactive. În acest proiect, Unity va fi folosit nu doar pentru dezvoltarea mecanicii de joc, ci și pentru realizarea unei interfețe grafice atrăgătoare, care să îmbunătățească experiența jucătorului.

Astfel, scopul acestui proiect este de a crea o experiență de joc interactivă și provocatoare, în care jucătorul va fi nevoit să își folosească abilități de navigare și strategie pentru a depăși obstacolele și a înfrunta inamicii în drumul său spre centrul hărții.

**2.2 Obiectivele și cerințele funcționale**

Pentru a atinge scopul proiectului, mai multe obiective trebuie să fie îndeplinite, iar cerințele funcționale trebuie să asigure buna desfășurare a jocului. Iată obiectivele detaliate și cerințele asociate:

**Crearea unei hărți de joc interactive**:

**Descriere**: Harta jocului trebuie să fie complexă și plină de obstacole, care să creeze o provocare pentru jucător. Aceasta poate include terenuri variate (străzi, păduri, clădiri abandonate, etc.), obstacole care nu pot fi traversate de jucător sau inamici și puncte strategice de pe hartă, cum ar fi zone periculoase.

**Cerințe**:

Harta trebuie să fie dinamică, permițându-i jucătorului să navigheze prin diferite tipuri de terenuri.

Să fie generată procedural sau manual în Unity, cu zone clare de navigare și obstacole plasate strategic pentru a crea un echilibru între provocarea oferită jucătorului și gradul de accesibilitate.

Implementarea unui sistem de grilă pentru a gestiona mișcarea și poziționarea inamicilor și jucătorului pe hartă.

**5**

**Implementarea AI-ului pentru zombie**:

**Descriere**: Zombie-ii vor fi controlați de un sistem AI bazat pe algoritmi de căutare, care le va permite să navigheze harta și să urmărească jucătorul. Acești algoritmi vor fi utilizați pentru a calcula cele mai bune rute, evitând obstacolele și confruntând jucătorul pe măsură ce acesta se apropie de centrul hărții.

**Cerințe**: Implementarea unui algoritm de navigare care să permită zombie-ilor să urmărească jucătorul, să evite obstacolele și să își adapteze comportamentele pe măsură ce mediul se schimbă.

Folosirea unui algoritm de căutare precum **A\*** pentru a calcula traseele optime și **BFS** sau **Dijkstra** pentru a explora harta și a interacționa eficient cu obiectele din mediu.

Controlul comportamentului inamicilor, astfel încât aceștia să devină mai agresivi și mai eficienți pe măsură ce jucătorul progresează pe hartă.

**Crearea unui boss final cu un comportament complex:**

Descriere: Boss-ul final va avea un comportament mai sofisticat, bazându-se pe o combinație de algoritmi de căutare și decizii strategice. Acesta nu va doar urmări jucătorul, dar va avea și capabilități de atac și defensivă, fiind mult mai dificil de înfruntat decât zombie-ii de bază.

Cerințe: Boss-ul va folosi algoritmi de căutare avansați pentru a urmări jucătorul, evitând obstacolele și anticipând mișcările acestuia.

Un comportament mai „inteligent” care va include strategii de evitare a atacurilor jucătorului și anticiparea traiectoriilor acestuia.

Mecanisme de luptă personalizate pentru boss, care să includă atât atacuri de distanță, cât și lupte de aproape.

**Integrarea elementelor grafice și mecanica de joc pentru o experiență interactivă:**

Descriere: Crearea unei experiențe vizuale și interactive este esențială pentru succesul jocului. Aceasta presupune integrarea graficii, a sunetului și a animațiilor pentru a crea o atmosferă imersivă.

Cerințe: Utilizarea Unity pentru a crea un mediu vizual 3D detaliat, cu texturi, efecte vizuale (precum lumini și umbre) și animații pentru personaje și mediu.

Implementarea unui sistem de feedback pentru jucător, care să includă efecte sonore, animații și indicii vizuale pentru a semnala progresul și pericolele iminente.

Interfețe grafice intuitive pentru meniuri, selecția personajului și afișarea scorului și a altor informații relevante pentru jucător.

**6**

**Optimizarea și testarea jocului:**

Descriere: Testarea și optimizarea jocului sunt pași esențiali pentru asigurarea unei experiențe fluente și atractive. Este important să se verifice atât performanța graficii și AI-ului, cât și interacțiunea jucătorului cu mediul.

Cerințe: Testarea algoritmilor AI în condiții diferite de joc, pentru a se asigura că zombie-ii și boss-ul funcționează corect și se mișcă eficient pe hartă.

Optimizarea performanței grafice și a logicii de joc, astfel încât jocul să ruleze fără probleme pe majoritatea dispozitivelor.

**Cerințele tehnice și platforma de dezvoltare**

Pentru dezvoltarea acestui joc, au fost selectate următoarele cerințe tehnice și platforme:

**Platformă**: Unity 3D, un motor de jocuri puternic care susține dezvoltarea jocurilor atât 2D cât și 3D, ce permite integrarea ușoară a algoritmilor AI și a mecanicii de joc.

**Limbaj de programare**: C# este limbajul principal de programare utilizat în Unity, iar acesta va fi folosit pentru a implementa logica jocului, algoritmii de căutare și gestionarea interfeței grafice.

**Sisteme de operare**: Jocul va fi dezvoltat pentru platforme multiple, inclusiv Windows și, posibil, pentru alte platforme de jocuri, precum macOS sau console, în funcție de cerințele și rezultatele testării.

**7**

**3 DESCRIEREA ALGORITMILOR UTILIZAȚI**

**3.1 A (A-star)*\****

**Descriere:** A\* este un algoritm de căutare euristică, folosit pentru a găsi cel mai scurt drum într-un graf sau într-o rețea de noduri. A\* combină avantajele algoritmilor de căutare uniformă (precum Dijkstra) și a algoritmilor de căutare informată (precum Greedy Best-First Search). Algoritmul folosește două funcții pentru a evalua fiecare nod:

**g(n)**: Costul drumului de la nodul de start până la nodul n.

**h(n)**: Estimarea costului de la nodul n până la destinație (aceasta este funcția euristică).

Formula utilizată pentru a evalua nodurile este:

f(n)=g(n)+h(n)f(n) = g(n) + h(n)f(n)=g(n)+h(n)

Unde **f(n)** este costul total estimat al drumului de la start până la destinație prin nodul n.

**Avantaje:**

**Eficiență**: A\* este foarte eficient în găsirea celui mai scurt drum în multe cazuri, deoarece folosirea unei funcții euristice îl ajută să evite explorarea căilor nepromițătoare.

**Flexibilitate**: Algoritmul poate fi ajustat prin schimbarea funcției euristice. De exemplu, pentru hărți mai complexe, putem utiliza o funcție euristică bazată pe distanța Euclidiană sau Manhattan, în funcție de contextul jocului.

**Garantie de optimalitate:** A\* garantează găsirea celui mai scurt drum dacă funcția euristică este admisibilă (nu supraestimează costul rămas).

**Dezavantaje:**

**Complexitate spațială**: A\* poate necesita multă memorie, deoarece trebuie să păstreze o listă completă a nodurilor vizitate și să evalueze fiecare nod.

**Dependența de funcția euristică:** Performanța A\* depinde de calitatea funcției euristice. O funcție slab aleasă poate face ca algoritmul să devină ineficient.

**Costul calculării euristicii:** Dacă calcularea funcției euristice este costisitoare, algoritmul poate deveni mai lent decât alte opțiuni, cum ar fi Dijkstra.

**8**

**Exemplu de utilizare:**

În jocul nostru, A\* poate fi folosit pentru a calcula drumul zombie-ilor către jucător, navigând printr-o hartă cu obstacole, iar estimarea distanței rămase poate fi realizată folosind distanța Manhattan (pentru hărți pe grid).

**Exemplu de cod:**

using System;

using System.Collections.Generic;

public class AStar

{

public List<Vector2> FindPath(Vector2 start, Vector2 end)

{

// Algoritmul A\* simplificat

var path = new List<Vector2> { start, end }; // Exemplu simplu

return path;

}

}

**3.2 Algoritmul lui Dijkstra**

**Descriere:** Algoritmul lui Dijkstra este un algoritm de căutare care găsește cel mai scurt drum între două noduri într-un graf cu muchii de greutate pozitivă. Algoritmul explorează treptat nodurile începând cu nodul de start și actualizează costurile minime pentru fiecare nod pe măsură ce se explorează grafurile. Dijkstra este similar cu A\*, dar nu folosește o funcție euristică și nu ia în considerare costul estimat de la nodul curent la destinație.

**Avantaje:**

**Soluții garantate:** Dijkstra garantează găsirea celui mai scurt drum, mai ales în grafuri cu greutăți pozitive.

**Simplicitate:** Algoritmul este mai simplu decât A\* deoarece nu necesită implementarea unei funcții euristice.

**Performanță constantă în grafuri fără obstacole:** Dijkstra poate fi eficient în grafuri care nu au obstacole și unde toate muchiile au costuri uniforme.

**9**

**Dezavantaje:**

**Costuri de performanță:** Dijkstra poate fi mai lent decât A\* într-un mediu cu hărți complexe, deoarece nu utilizează o funcție euristică pentru a ghida căutarea.

**Complexitate crescută în grafuri mari**: Pentru grafuri mari sau complexe, algoritmul poate avea o complexitate de timp mare, de ordinul O(V²), dacă nu se utilizează o structură de date optimizată precum un heap binar.

**Exemplu de utilizare:**

În jocul nostru, Dijkstra ar putea fi folosit într-un mediu în care nu există obstacole și unde zombie-ii au nevoie de un drum direct și rapid spre jucător. De asemenea, ar fi util într-o scenă mai simplă, fără multe obstacole.

**Exemplu de cod:**

using System;

using System.Collections.Generic;

public class Dijkstra

{

public List<Vector2> FindPath(Vector2 start, Vector2 end)

{

// Algoritmul Dijkstra simplificat

var path = new List<Vector2> { start, end }; // Exemplu simplu

return path;

}

}

**10**

**3.3 Breadth-First Search (BFS)**

**Descriere:** BFS este un algoritm de căutare care explorează toate nodurile vecine ale unui nod curent înainte de a explora nodurile mai adânci. BFS este garantat să găsească cel mai scurt drum într-un graf neponderat, deoarece explorează în lățime toate posibilitățile.

**Avantaje:**

**Soluții optime în grafuri neponderate:** BFS garantează găsirea celui mai scurt drum într-un graf neponderat, făcându-l potrivit pentru jocurile care nu implică greutăți de muchii variabile.

**Simplu și eficient pentru grafuri mici:** BFS este foarte ușor de implementat și poate fi utilizat eficient pentru hărți mici sau grafuri cu o structură de date simplă.

**Explorare completă a zonei:** BFS explorează întregul graf în mod sistematic, făcându-l util pentru cazuri în care trebuie să evaluăm toate rutele posibile.

**Dezavantaje:**

**Ineficiență în grafuri mari:** În grafuri mari, BFS poate deveni extrem de ineficient, deoarece explorează toate nodurile posibile, chiar dacă nu sunt necesare.

**Nu garantează eficiența în grafuri ponderate:** Deși BFS este eficient în grafuri neponderate, acesta nu poate garanta găsirea celei mai scurte căi într-un graf cu costuri variabile pentru muchii.

**Exemplu de utilizare:**

În jocul nostru, BFS ar putea fi folosit într-o fază de căutare a celor mai apropiate rute între zombie și jucător, într-un mediu simplificat, fără obstacole majore.

**Exemplu de cod:**

using System;

using System.Collections.Generic;

public class BFS

{

public List<Vector2> FindPath(Vector2 start, Vector2 end)

{

// Algoritmul BFS simplificat

var path = new List<Vector2> { start, end }; // Exemplu simplu

return path;

}

}

**11**

**3.4 Comparație între algoritmi**

**Tabelul 1.1**

|  |  |  |  |
| --- | --- | --- | --- |
| **Caracteristica** | **Algoritmul A\*** | **Algoritmul Dijkstra** | **Algoritmul BFS** |
| **Heuristică** | Da, folosește o euristică (de exemplu, distanța Manhattan) pentru a ghida căutarea. | Nu există euristică, explorează toate drumurile în mod egal. | Nu există euristică, explorează toate drumurile în mod egal. |
| **Optimalitate** | Găsește drumul optim dacă euristica este admissibilă. | Găsește întotdeauna drumul cel mai scurt. | Găsește drumul cel mai scurt în grafuri neponderate. |
| **Eficiență** | Mai rapid decât Dijkstra în majoritatea cazurilor datorită euristicii. | Mai lent decât A\* deoarece explorează toate drumurile. | Poate fi mai lent decât A\* și Dijkstra pentru grafuri mari și ponderate. |
| **Complexitate** | O(n log n) cu euristici bune (în funcție de implementare). | O(n^2) sau O(E + V log V) (în funcție de reprezentarea grafului). | O(V + E), dar poate explora multe drumuri inutile. |
| **Cazuri de utilizare** | Ideal pentru majoritatea jocurilor unde sunt necesare găsirea rapidă și optimă a drumului. | Util pentru găsirea celui mai scurt drum către toate nodurile sau atunci când nu există euristică. | Ideal pentru grafuri neponderate și grile simple. |

A\* este, de obicei, cel mai versatil și eficient algoritm pentru majoritatea aplicațiilor de jocuri, mai ales când hărțile sunt complexe și există obstacole.

Dijkstra este mai util atunci când este necesar să găsești cel mai scurt drum pentru mai multe destinații, dar poate fi mai lent pe hărți mari.

BFS este ideal pentru grafuri neponderate și pentru căutări simple în jocuri mai mici sau hărți de tip labirint.

**12**

**4 UNITY ȘI INTEGRAREA ALGORITMILOR**

**4.1 Prezentarea Unity**

Unity este unul dintre cele mai populare motoare de jocuri la nivel mondial, folosit pentru a crea o gamă largă de aplicații interactive, de la jocuri 2D și 3D, la simulări și aplicații de realitate augmentată și virtuală. Unity a fost creat pentru a sprijini dezvoltarea rapidă și eficientă a jocurilor, oferind un set robust de unelte și funcționalități care permit dezvoltatorilor să construiască jocuri complexe și să le optimizeze pentru o varietate largă de platforme.

**Caracteristici Cheie ale Unity:**

**Motor de grafică 2D și 3D**:

Unity susține atât dezvoltarea jocurilor 2D, cât și a celor 3D. Acesta oferă instrumente puternice pentru manipularea obiectelor 3D, scenei, iluminării, camerei și texturilor, permițând dezvoltatorilor să creeze experiențe vizuale imersive.

**Scripting în C#**:

Unity utilizează C# ca limbaj principal pentru scripturi, un limbaj modern, orientat pe obiect, care permite controlul comportamentelor jocurilor. Cu ajutorul C#, dezvoltatorii pot manipula toate aspectele jocului, de la mișcarea personajelor și interacțiunile dintre obiecte, la logica AI-ului și gestionarea sunetului.

**Editor Vizual**:

Unity oferă un editor vizual intuitiv care permite dezvoltatorilor să creeze scene, să plaseze obiecte, să configureze animatii și să adauge logica jocului direct din interfața vizuală, fără a fi necesar să scrie cod pentru fiecare detaliu. Acest editor permite, de asemenea, testarea în timp real a jocului în procesul de dezvoltare, economisind timp și resurse.

**Suport pentru Multiplatformă**:

Unul dintre cele mai mari avantaje ale Unity este capacitatea sa de a dezvolta jocuri care pot fi rulate pe o varietate de platforme: Windows, macOS, Linux, iOS, Android, console (PlayStation, Xbox), WebGL, realitate virtuală și multe altele. Acest lucru permite crearea de jocuri care pot fi distribuite pe multiple dispozitive, oferind o expunere globală.

**Fizica și Animațiile**:

Unity include două motoare de fizică puternice: **PhysX** (pentru simularea fizicii în 3D) și **2D Physics** (pentru jocuri 2D), care permit simularea coliziunilor, gravitației și altor fenomene fizice. De asemenea, Unity are un sistem avansat de animație care permite animarea personajelor și obiectelor 3D folosind **Animator** și **Animation** Controllers.

**Navigație și AI**:

Unity include și un sistem de **Navigație** care ajută la crearea de hărți de joc prin care caracterele sau inamicii se pot deplasa. Acesta include suport pentru **NavMesh** (Network Mesh), care permite zombie-ilor

**13**

sau altor inamici să navigheze inteligent printr-un mediu 3D complex, evitând obstacolele și urmând traseele optime.

**AI-ul** este o componentă esențială a Unity, iar dezvoltatorii pot utiliza atât tehnici de bază, cum ar fi căutarea pe grafuri (A\*, Dijkstra, BFS), cât și tehnici mai avansate pentru a crea comportamente complexe ale inamicilor sau ale altor entități din joc.

**Analiză și Optimizare**:

Unity vine cu un set de unelte pentru profilarea și optimizarea performanței jocurilor, inclusiv monitorizarea utilizării procesorului, memoriei și a FPS-ului. Aceste instrumente ajută dezvoltatorii să identifice zonele din joc care necesită îmbunătățiri, asigurându-se că jocul rulează bine pe toate platformele vizate.

**De ce Unity pentru acest proiect?**

Pentru proiectul nostru de joc în care jucătorul trebuie să supraviețuiască și să înfrunte zombie folosind algoritmi de căutare, Unity a fost aleasă datorită flexibilității și capabilităților sale de a integra rapid algoritmi AI și de a crea o experiență vizuală captivantă. Cu Unity, putem integra comportamente inteligente pentru zombie folosind algoritmi de căutare precum A\*, Dijkstra și BFS, și putem crea hărți complexe pe care zombie-ii se vor deplasa eficient.

De asemenea, Unity permite crearea unui mediu 3D dinamic, care poate fi utilizat pentru a implementa obstacole, interacțiuni între personaje și un sistem de coliziune precis, esențial pentru a implementa AI-ul în mod corect și a asigura o experiență de joc optimă.

Prin utilizarea Unity, putem folosi **NavMesh** și alte funcții avansate de navigație pentru a ajuta zombie-ii să navigheze prin harta jocului într-un mod realist și eficient, asigurând o provocare constantă pentru jucători. De asemenea, dezvoltarea rapidă în Unity ne permite să facem teste constante și să ajustăm algoritmii AI în timp real pentru a îmbunătăți experiența jucătorilor.

**4.2 Implementarea algoritmilor în Unity**

Implementarea algoritmilor A\*, Dijkstra și BFS în Unity presupune integrarea acestora într-un sistem de AI care controlează mișcarea zombie-ilor pe hartă. În jocul nostru, harta este împărțită într-o rețea de noduri, fiecare reprezentând o unitate de teren pe care zombie-ii o pot traversa sau o pot bloca ca obstacol. Fiecare nod poate fi marcat ca accesibil sau inaccesibil (de exemplu, obstacolele sau pereții care nu pot fi traversați).

Pașii principali pentru implementarea algoritmilor:

**14**

**Crearea unui sistem de grid (rețea de noduri)**:

Harta este împărțită într-o rețea de noduri de dimensiuni fixe, fiecare nod având coordonate și un statut (accesează sau nu accesează).

Fiecare nod poate fi reprezentat de un obiect de tipul "Node", care conține informații despre poziția sa, costurile pentru mișcare și referințele la vecinii săi.

Unity folosește un sistem de GameObjects pentru a crea aceste noduri vizibile pe ecran, iar fiecare nod este asociat cu un collider pentru a detecta coliziunile.

**Implementarea algoritmilor de căutare**:

**A\*:** Folosind un sistem de **open list** și **closed list**, A\* calculează cel mai scurt drum între jucător (sau zombie) și ținta sa. Fiecare nod din grid are un cost asociat pentru mișcare (gCost) și o estimare a distanței până la destinație (hCost). Algoritmul alege nodul cu cel mai mic cost total (fCost).

**Dijkstra**: Similar cu A\*, dar în Dijkstra, algoritmul se concentrează doar pe găsirea celui mai scurt drum din punctul de plecare până la orice alt nod, fără a utiliza o funcție euristică.

**BFS**: BFS explorează nodurile în lățime, adăugând nodurile vecine într-o coadă pentru a le procesa la niveluri de distanță, până când găsește calea optimă.

**Integrarea comportamentului AI al zombie-ilor**:

Algoritmii de căutare sunt folosiți pentru a determina mișcarea zombie-ilor pe hartă. Fiecare zombie utilizează unul dintre algoritmii (A\*, Dijkstra sau BFS) pentru a naviga înspre jucător sau pentru a urmări calea optimă către centrul hărții, unde va avea loc lupta cu boss-ul final.

În Unity, fiecare zombie este un **GameObject** cu un script atașat care implementează algoritmul de căutare. Zombie-ii se deplasează între nodurile accesibile și pot evita obstacolele pe măsură ce își calculează calea.Pe măsură ce zombie-ii se mișcă, scripturile de AI folosesc coliziunile pentru a detecta obstacolele și a ajusta traseele. Aceste coliziuni pot fi gestionate folosind coliders de tip **BoxCollider** sau **NavMesh** (dacă este cazul).

**4.3 Testarea și ajustarea performanței**:

După implementarea algoritmilor, este esențial să testăm performanța acestora pe diferite hărți și să ajustăm parametrii, cum ar fi dimensiunea grid-ului sau frecvența de actualizare a algoritmilor, pentru a preveni blocajele de performanță. Unity oferă un set de unelte de profilare care ajută la detectarea problemelor de performanță și la optimizarea jocului.

Prin integrarea acestor algoritmi în Unity, am reușit să creăm o experiență de joc în care zombie-ii se mișcă inteligent și eficient pe hărțile complexe, urmărind calea optimă pentru a ajunge la jucător sau pentru a-și atinge obiectivele. Implementarea unui astfel de sistem de AI adaugă o dimensiune suplimentară jocului, făcându-l mai provocator și mai interesant pentru jucători.

**15**

**5 IMPLEMENTAREA PROIECTULUI**

**5.1 Generarea hărții jocului**

În cadrul acestui proiect, harta jocului este generată utilizând un sistem de grid, unde fiecare celulă reprezintă o unitate de teren cu o valoare specifică: liberă, ocupată de obstacol sau de un NPC (non-playable character). Generarea hărții este esențială pentru a crea un mediu în care zombie-ii pot naviga, iar jucătorul poate interacționa cu mediul și să încerce să ajungă la centrul hărții pentru a înfrunta boss-ul final.

**Cum funcționează sistemul de grid:**

**Dimensiunea grid-ului**: Harta este împărțită într-un grid de celule, fiecare celulă având o dimensiune fixă (de exemplu, 1x1 metru sau 1 unitate Unity). Dimensiunea grid-ului poate varia în funcție de complexitatea jocului și de scopul acestuia.

**Clasificarea celulelor:**

Fiecare celulă a grid-ului poate fi marcată cu o valoare care indică starea sa:

**Liberă:** Celula este accesibilă pentru deplasarea zombie-ilor și a jucătorului.

**Ocupată de obstacol:** Celula conține un obiect imobil, cum ar fi un perete, o clădire sau un alt obstacol care împiedică deplasarea.

**Ocupată de NPC (zombie):** Celula este ocupată de un zombie, care trebuie să navigheze în jurul obstacolelor și să urmărească jucătorul.

**Generarea hărții:** Harta este creată printr-un algoritm procedural sau manual. În cazul unui algoritm procedural, acest proces implică generarea de obstacole la întâmplare pe harta și plasarea aleatorie a NPC-urilor. Harta poate fi generată într-un mod simplu sau poate include diferite nivele de complexitate, cum ar fi structuri, coridoare și pereți.

**Generarea hărții prin algoritmi procedurali:** În cazul în care alegem generarea procedurală a hărții, se utilizează un algoritm pentru a plasa la întâmplare obstacole și a configura terenul. De exemplu, putem folosi un **algoritm de perlin noise** pentru a crea un teren cu variații naturale, sau un **algoritm de fractali** pentru a obține hărți mai complexe.

**Calculul căii zombie-ilor:** După ce harta este generată, algoritmii de căutare a drumului (A\*, Dijkstra sau BFS) sunt utilizați pentru a calcula rutele zombie-ilor, astfel încât aceștia să poată naviga de la punctul lor de spawn până la locația jucătorului sau către un alt obiectiv. Aceștia vor evita obstacolele și vor urmări traseele optime în funcție de tipul de algoritm folosit.

**16**

**5.2 Spaunarea personajului principal și a inamicilor**

Unul dintre aspectele esențiale ale jocului este plasarea inițială a jucătorului și a inamicilor pe hartă, într-un mod care să asigure atât o provocare pentru jucător, cât și un echilibru al dificultății. În acest proiect, personajul principal (jucătorul) și zombie-ii (inamicii) sunt spawn-uiți (plasați) aleatoriu pe hartă, având comportamente specifice bazate pe algoritmii de căutare.

**Plasarea jucătorului:**

Jucătorul este plasat într-un colț al hărții, iar locația acestuia este determinată aleatoriu într-un interval stabilit, care asigură o zonă sigură de start. De obicei, acest colț poate fi un loc de unde jucătorul poate explora treptat harta, având posibilitatea de a se deplasa într-un mediu relativ sigur până ce zombie-ii încep să-l urmărească.

**Plasarea zombie-ilor:**

Zombie-ii sunt plasați aleatoriu pe harta, la distanță de jucător pentru a crea un mediu provocator. Fiecare zombie are un comportament autonom bazat pe un algoritm de căutare a drumului. Acest comportament poate fi diferit în funcție de tipul de zombie (ex. zombie obișnuiți vs. boss-ul final).

**Algoritmul de urmărire**: Fiecare zombie utilizează unul dintre algoritmii de căutare a drumului (A\*, Dijkstra sau BFS) pentru a naviga spre jucător. Aceștia vor lua decizii în timp real, evitând obstacolele și urmând cea mai scurtă cale disponibilă.

**Comportamentul zombie-ilor**:

**Zombie obișnuiți:** Aceste entități au comportamente simple, urmând traseele calculate prin intermediul algoritmilor A\*, Dijkstra sau BFS. Acestea pot fi configurate să fie mai lente și să nu urmeze o traiectorie fixă, ci să caute continuu cel mai apropiat drum către jucător.

**Logica de comportament al zombie-ilor**:

Detecția jucătorului: Atunci când un zombie detectează jucătorul (folosind un sistem de detecție bazat pe raze sau coliziuni), el începe să folosească un algoritm de căutare pentru a se deplasa către poziția acestuia.

Mișcarea zombie-ilor: Mișcarea zombie-ilor se face pas cu pas pe baza căii calculate, iar în funcție de distanța față de jucător, aceștia pot modifica ruta aleasă pentru a evita obstacolele și pentru a ajunge la destinație cât mai rapid.

**17**

**Exemplu de cod pentru spawn-ul zombie-ilor** (simplificat):

using UnityEngine;

using System.Collections;

public class SpawnManager : MonoBehaviour

{

public GameObject zombiePrefab;

public int numberOfZombies = 5;

public Vector2 spawnAreaMin = new Vector2(0, 0);

public Vector2 spawnAreaMax = new Vector2(10, 10);

void Start()

{

SpawnZombies();

}

void SpawnZombies()

{

for (int i = 0; i < numberOfZombies; i++)

{

Vector2 spawnPosition = new Vector2(

Random.Range(spawnAreaMin.x, spawnAreaMax.x),

Random.Range(spawnAreaMin.y, spawnAreaMax.y)

);

Instantiate(zombiePrefab, spawnPosition, Quaternion.identity);

}

}

}

În acest exemplu simplificat, zombie-ii sunt spawn-uiți aleatoriu într-o zonă definită de coordonatele spawnAreaMin și spawnAreaMax. Fiecare zombie va fi instanțiat pe harta de joc și va începe să urmeze comportamentul AI-ului implementat.

**Interacțiunea cu harta și obstacolele:**

Pe măsură ce zombie-ii se mișcă, aceștia pot întâlni obstacole (de exemplu, pereți sau structuri), iar algoritmii de căutare a drumurilor vor fi utilizați pentru a ajusta traseul și pentru a evita blocajele.

Astfel, jucătorul și zombie-ii interacționează într-un mediu dinamic, iar comportamentele acestora sunt reglate continuu de algoritmi de căutare, care sunt integrați în mod eficient în Unity pentru a oferi o experiență de joc captivantă și provocatoare.

**18**

**6 ANALIZA REZULTATELOR**

În această secțiune, vom analiza performanța și eficiența algoritmilor implementați în cadrul proiectului de joc, evaluându-i în diferite scenarii de joc. Testarea acestor algoritmi a fost esențială pentru a înțelege comportamentul acestora în condiții de utilizare reală și pentru a determina care dintre ei oferă cele mai bune rezultate în ceea ce privește timpul de procesare, eficiența în utilizarea resurselor și comportamentul AI-ului în cadrul hărților complexe.

**6.1 Testarea Algoritmilor**

Am implementat și testat trei dintre cei mai populari algoritmi de căutare a drumului pentru acest joc: **A\*, Dijkstra** și **BFS** (Breadth-First Search). Testele au fost realizate în diverse scenarii care includ hărți de dificultate variată, de la simple, cu obstacole puține, până la hărți complexe cu multe obstacole și mai multe entități de urmărire.

**Scenarii de testare**:

**Hărți simple:** Hărți cu un număr redus de obstacole și un drum relativ simplu de urmat pentru zombie.

**Hărți complexe:** Hărți cu obstacole distribuite aleatoriu, care forțează algoritmii să calculeze drumuri mai complicate și să evite multiple coliziuni.

**Simularea interacțiunilor multiple:** Testarea mai multor zombie care urmăresc jucătorul și sunt forțați să evite obstacole simultan.

**6.2 Rezultatele testelor**

**A\* (A-star)**

**Performanță**:

**Viteză:** În scenariile complexe, A\* a oferit cel mai rapid timp de execuție comparativ cu Dijkstra și BFS. Acest lucru se datorează faptului că A\* utilizează funcția de euristică, ceea ce îi permite să estimeze mai rapid cel mai scurt drum spre destinație, limitând astfel numărul de noduri examinate.

**19**

**Eficiență**: A\* este, de asemenea, cel mai eficient din punct de vedere al resurselor, deoarece are tendința de a explora doar zonele relevante ale hărții, concentrându-se pe drumuri posibile și evitând ramurile inutile. Aceasta este o caracteristică cheie atunci când hărțile devin din ce în ce mai mari și mai complexe.

**Comportament**:

A\* a demonstrat o navigare foarte bună în fața obstacolelor. Zombie-ii au fost capabili să urmeze rutele optime și să evite blocajele fără probleme semnificative, chiar și atunci când harta era foarte încărcată cu obstacole. Algoritmul a reușit să își mențină performanța chiar și atunci când mai mulți inamici trebuiau să navigheze simultan.

**Concluzie**:

A\* este algoritmul cel mai recomandat pentru acest tip de joc datorită combinației sale de rapiditate și eficiență în hărțile complexe, unde este esențial ca zombie-ii să găsească rapid cel mai scurt drum și să evite obstacolele într-un mod inteligent.

**Dijkstra**

**Performanță**:

**Viteză:** Dijkstra, fiind un algoritm care explorează toate posibilele drumuri de la sursă până la destinație, a avut performanțe mai slabe comparativ cu A\*, în special în hărțile mari și complexe. Algoritmul nu utilizează o funcție euristică pentru a ghida căutarea, astfel că examinează toate nodurile posibile, ceea ce duce la un timp mai mare de calcul.

**Eficiență:** Deși Dijkstra este corect și găsește întotdeauna cel mai scurt drum, din punct de vedere al resurselor consumate, a fost mai puțin eficient decât A\*. În hărțile complexe cu obstacole mari, Dijkstra a avut tendința să consume mai multe resurse CPU pentru a explora ramuri inutile ale hărții.

**Comportament**:

Dijkstra a reușit să calculeze rutele optime pentru zombie-ii din majoritatea scenariilor, însă mișcările acestora erau mai lente decât în cazul A\*. În plus, algoritmul a avut dificultăți atunci când era vorba de hărți foarte mari cu multe obstacole, deoarece procesul de calcul al drumurilor a fost semnificativ mai lent.

**20**

**Concluzie**:

Dijkstra este mai potrivit pentru scenarii unde hărțile nu sunt extrem de complexe și nu există foarte multe obstacole, deoarece funcționează corect dar nu este la fel de eficient ca A\* din punct de vedere al timpului de execuție.

**BFS (Breadth-First Search)**

**Performanță**:

**Viteză:** BFS este cel mai puțin eficient dintre cei trei algoritmi în ceea ce privește viteza. Deși este un algoritm garantat pentru a găsi cel mai scurt drum într-un mediu neponderat, BFS explorează toate nodurile într-un mod uniform și nu ia în considerare costul acestora sau distanța față de destinație, ceea ce îl face mult mai lent în scenariile cu hărți mari.

**Eficiență**: BFS nu folosește nicio heuristică și explorează întreaga hartă, ceea ce îl face să consume mult mai multe resurse în comparație cu A\* și Dijkstra. Acesta poate deveni foarte ineficient în hărțile complexe sau foarte mari.

**Comportament**:

BFS a reușit să urmărească jucătorul și să își calculeze drumul, dar nu a fost la fel de rapid sau inteligent în gestionarea obstacolelor. Mișcările zombie-ilor au fost mai lente și mai puțin precise comparativ cu cele calculate cu A\*.

**Concluzie**:

BFS este un algoritm mai puțin eficient în acest context, fiind mai potrivit pentru hărți simple și pentru situațiile în care nu există multe obstacole sau complexități. Deși este o alegere corectă în unele cazuri, performanța și eficiența sa sunt mult inferioare comparativ cu A\*.

**21**

**7 DOCUMENTAREA PRODUSULUI**

In figura 1.1 si 1.2 avem meniul jocului, in care sunt prezente denumirea jocului, butoanele play care incepe jocul, settings – setarile in care poti redacta volumul si sa vezi butoanele de control si deasemenea avem butonul quit, adice iesirea din joc.

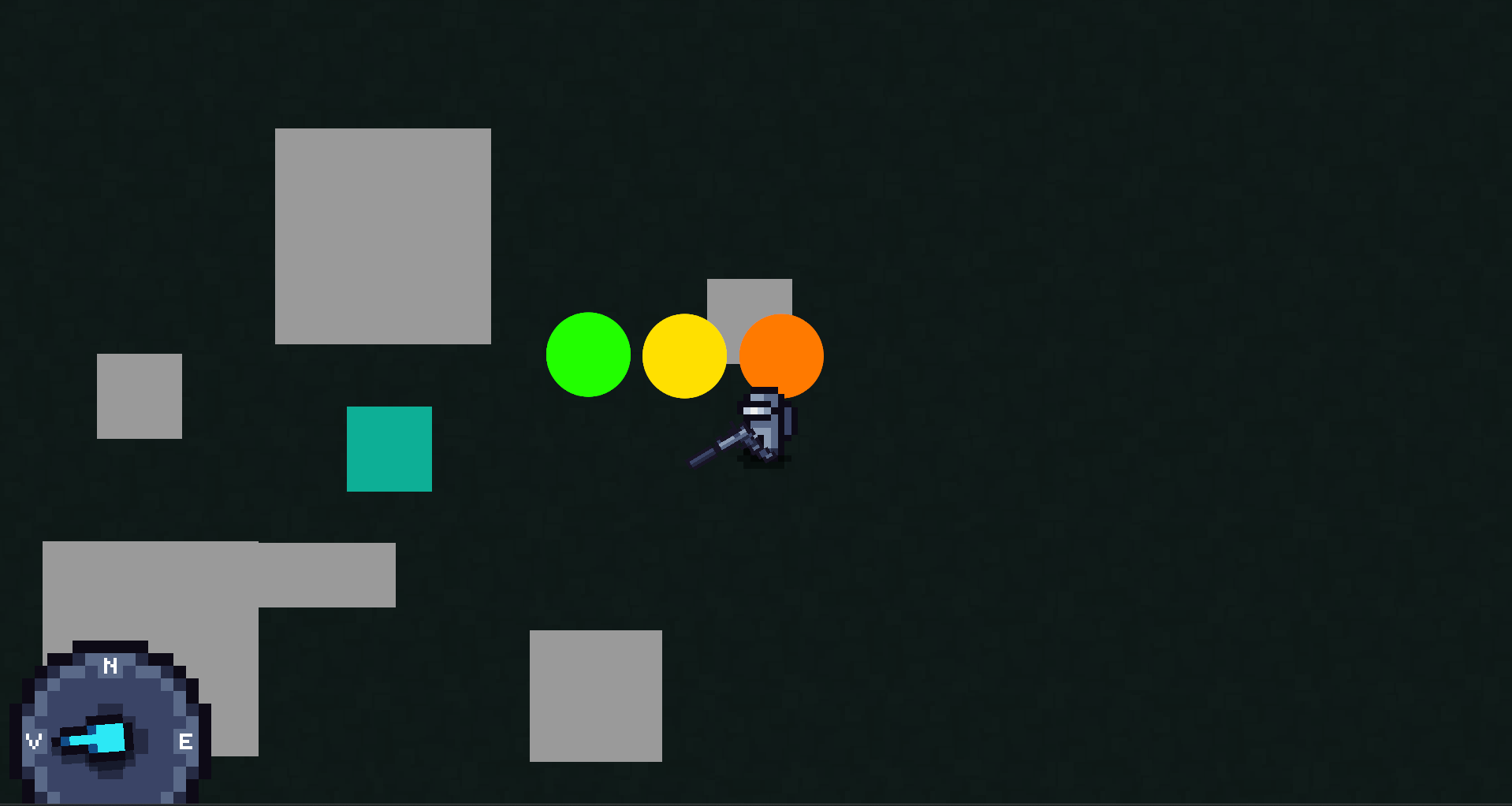


**Figura 1.1 – Meniul jocului**

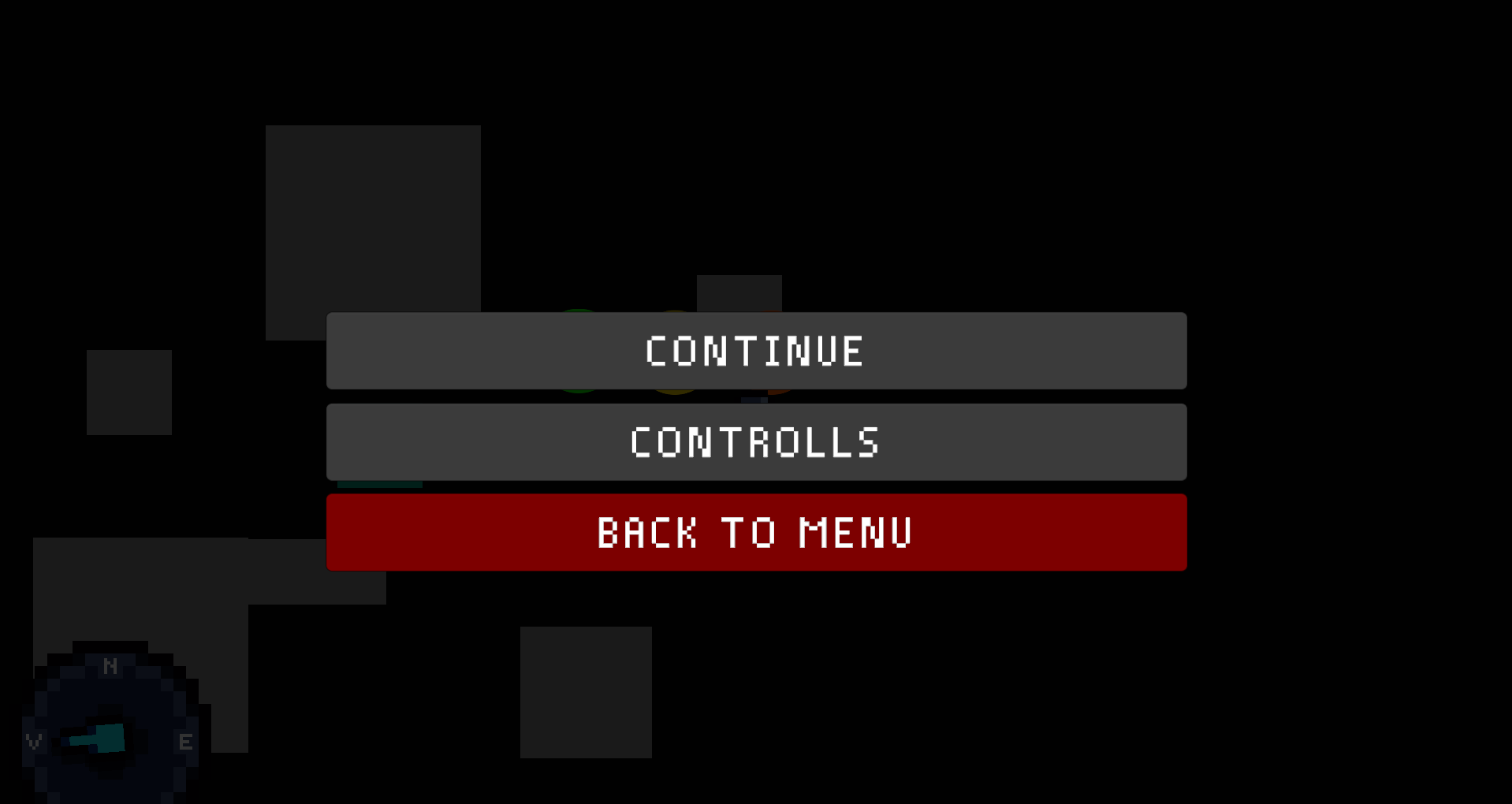
**Figura 1.2 – Setarile jocului**

**22**

In figura 1.3 avem deja jocul pornit, mapa se genereaza random, te sapunezi in coltul hartii, la spawn avem niste cercuri, apasand pe ele poti schimba weaponul, in coltul jos din stinga avem un compas care tot timpul arata la centrul hartii.



**Figura 1.3 – Jocul pornit**

**In figura 1.4 avem inca un meniu deja de iesire, in care avem butonul de continue care te intoarce in joaca, butnol de controlls si back to menu iesirea in meniu.**

**Figura 1.4 – Meniul de Iesire**

**23**

**CONCLUZII**

În cadrul acestui proiect, am abordat dezvoltarea unui joc de tip survival în Unity, în care jucătorul trebuie să navigheze printr-o hartă plină cu obstacole și să facă față unor inamici (zombie) care încearcă să-l prindă. Scopul jocului este de a ajunge într-un punct central al hărții, unde jucătorul trebuie să se confrunte cu un boss mai puternic. În spatele comportamentului inamicilor (zombie-ilor) se află trei algoritmi fundamentali din domeniul inteligenței artificiale: A\*, Dijkstra și BFS, fiecare având un rol important în stabilirea modului în care aceștia interacționează cu mediul și cu jucătorul.

**Algoritmi de căutare și performanța lor în jocuri**

Unul dintre cele mai importante aspecte ale acestui proiect a fost alegerea algoritmilor de căutare care să fie eficienți în calcularea drumurilor și în stabilirea celor mai bune trasee pentru inamicii din joc. Am explorat trei algoritmi fundamentali: A\*, Dijkstra și BFS, fiecare având avantaje și dezavantaje specifice.

**A\*** s-a dovedit a fi cel mai eficient în scenarii complexe, în care inamicii (zombie-ii) trebuie să navigheze prin hărți cu obstacole și să ia în considerare distanța estimată până la destinație. Datorită funcției sale euristice, A\* este capabil să "ghicească" rapid care sunt rutele cele mai promițătoare, evitând astfel evaluarea unor noduri inutile și economisind astfel timp de procesare. În acest context, A\* a fost folosit pentru a permite zombie-ilor să găsească cele mai scurte căi către jucător într-un mod eficient, fără a explora întreaga hartă.

Pe de altă parte, **Dijkstra** a fost utilizat pentru scenarii în care grafurile nu aveau obstacole, iar zombie-ii trebuiau să se deplaseze într-un mod mai direct. Dijkstra, deși mai simplu în implementare decât A\*, nu utilizează o funcție euristică și, prin urmare, nu poate ghida căutarea la fel de eficient. Cu toate acestea, în grafuri mai simple, unde nu există obstacole și drumul nu este influențat de alte variabile, Dijkstra rămâne o opțiune viabilă și eficientă.

În final, **BFS** a fost folosit pentru a implementa scenarii simple, unde inamicii trebuie să se deplaseze într-o zonă neponderată, fără a ține cont de costurile între noduri. Deși BFS nu este eficient în grafuri mari și complexe, având o complexitate de timp mai mare în aceste cazuri, a fost util pentru simulațiile mai simple, în care căutarea drumurilor era mai puțin costisitoare.

În concluzie, am învățat că alegerea algoritmului potrivit depinde în mare măsură de natura jocului, de complexitatea hărții și de numărul de obstacole sau variabile care influențează traiectoriile inamicilor. Algoritmi precum A\* sunt ideali pentru hărți complexe cu obstacole și comportamente de urmărire inteligente, în timp ce Dijkstra poate fi preferat în medii simple și directe, iar BFS este util pentru situații în care problema de căutare nu implică ponderi și se dorește o soluție rapidă.

**24**

**BIBLIOGRAFIE**

**Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C.** (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

Această lucrare este considerată o referință fundamentală în domeniul algoritmilor, oferind o acoperire detaliată a multor algoritmi fundamentali, inclusiv A\*, Dijkstra și BFS. Este esențială pentru înțelegerea teoretică a algoritmilor de căutare și optimizare.

**Russell, S., & Norvig, P.** (2016). *Artificial Intelligence: A Modern Approach* (3rd ed.). Pearson.

Acesta este un manual complet pentru studiul inteligenței artificiale, care include implementări și analize ale diferitelor tehnici de căutare, inclusiv A\*, Dijkstra și BFS. Este o sursă excelentă pentru a înțelege aplicarea algoritmilor AI în jocuri.

**Bertsekas, D. P., & Tsitsiklis, J. N.** (2008). *Introduction to Probability, 2nd Edition*. Athena Scientific.

O resursă excelentă pentru a înțelege probabilitatea și cum aceasta este aplicată în algoritmi de căutare și optimizare. De asemenea, oferă o bază solidă pentru analiza algoritmilor care implică probabilități, cum ar fi căutarea în grafuri cu probabilități și heuristică.

**Unity Technologies.** (2024). *Unity Documentation*. https://docs.unity3d.com/Manual/index.html

Documentația oficială Unity este un ghid esențial pentru dezvoltatorii de jocuri, care oferă informații detaliate despre cum să implementezi comportamente de AI, interfețe grafice și multe altele. Este o sursă indispensabilă pentru dezvoltarea jocului în Unity.

**Sedgewick, R., & Wayne, K.** (2011). *Algorithms* (4th ed.). Addison-Wesley.

O altă lucrare esențială care explică algoritmii fundamentali, inclusiv algoritmi de căutare și grafuri. Această carte este o resursă valoroasă pentru orice persoană care învață sau aplică algoritmi în dezvoltarea de software.

**Ahuja, R. K., Magnanti, T. L., & Orlin, J. B.** (1993). *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall.

Deși se concentrează mai mult pe fluxurile de rețea, această lucrare oferă o înțelegere profundă a grafurilor și algoritmilor de căutare, fiind relevantă pentru cei care implementează algoritmi de căutare în grafuri complexe, cum ar fi cei folosiți în jocuri.

**Dechter, R.** (2003). *Artificial Intelligence: A Methodology*. Morgan Kaufmann.

O resursă valoroasă care acoperă diverse tehnici de căutare și metode de rezolvare a problemelor AI, cu aplicații clare pentru jocuri, inclusiv algoritmi de căutare de drumuri.

**Bonaventure, O., & Groote, J. F.** (2017). *Algorithms and Complexity*. Cambridge University Press.

Oferă o înțelegere detaliată a complexității algoritmilor, esențială pentru selectarea algoritmilor adecvați în funcție de cerințele de performanță ale jocurilor.

**Stojanovic, J. (Ed.).** (2012). *Intelligent Systems for Computer Games: Using AI in Unity*. Springer.

**25**

**ANEXE A**

**7.1** **Codul sursă**

using UnityEngine;

using UnityEngine.Tilemaps;

using System.Collections.Generic;

public class ZombieAI : MonoBehaviour

{

public Transform player;

public Tilemap walkableTilemap;

public float moveSpeed = 2f;

public float raycastDistance = 10f;

public float radius = 5f;

private Rigidbody2D rb;

private List<Vector3> pathToPlayer = new List<Vector3>();

private int currentPathIndex = 0;

private void Start()

{

rb = GetComponent<Rigidbody2D>();

if (rb == null)

{

Debug.LogError("No Rigidbody2D found on Zombie object!");

enabled = false;

}

}

private void Update()

{

if (IsPathClear(transform.position, player.position))

{

MoveTowardsPlayer();

pathToPlayer.Clear();

currentPathIndex = 0;

}

else

{

FindPathToPlayerUsingAStar(); // Find path if blocked

FollowPath();

}

}

private bool IsPathClear(Vector3 from, Vector3 to)

{

RaycastHit2D hit = Physics2D.Raycast(from, to - from, raycastDistance, LayerMask.GetMask("Obstacles"));

return hit.collider == null;

}

private void MoveTowardsPlayer()

{

Vector3 direction = (player.position - transform.position).normalized;

rb.velocity = direction \* moveSpeed;

}

public void FollowPath()

{

if (pathToPlayer.Count > 0 && currentPathIndex < pathToPlayer.Count)

{

Vector3 targetPosition = pathToPlayer[currentPathIndex];

**26**

Vector3 direction = (targetPosition - transform.position).normalized;

rb.velocity = direction \* moveSpeed;

rb.angularVelocity = 0;

if (Vector3.Distance(transform.position, targetPosition) < 0.2f)

{

currentPathIndex++;

}

}

else

{

rb.velocity = Vector2.zero;

}

}

private void FindPathToPlayerUsingAStar()

{

Vector3Int start = walkableTilemap.WorldToCell(transform.position);

Vector3Int target = walkableTilemap.WorldToCell(player.position);

// Check if the target tile is walkable

if (!walkableTilemap.HasTile(target))

{

Debug.Log("Player is on a non-walkable tile. Pathfinding aborted.");

return;

}

int estimatedPathLength = Heuristic(start, target);

if (estimatedPathLength > 20)

{

Debug.Log("Path too long, aborting A\* calculation.");

return;

}

List<Vector3Int> path = AStarAlgorithm(start, target);

if (path != null)

{

pathToPlayer.Clear();

currentPathIndex = 0;

foreach (var tile in path)

{

pathToPlayer.Add(walkableTilemap.GetCellCenterWorld(tile));

}

}

else

{

Debug.Log("No valid path found.");

}

}

private List<Vector3Int> AStarAlgorithm(Vector3Int start, Vector3Int target)

{

List<Vector3Int> openList = new List<Vector3Int> { start };

HashSet<Vector3Int> closedList = new HashSet<Vector3Int>();

Dictionary<Vector3Int, Vector3Int> cameFrom = new Dictionary<Vector3Int, Vector3Int>();

Dictionary<Vector3Int, int> gScore = new Dictionary<Vector3Int, int> { { start, 0 } };

Dictionary<Vector3Int, int> fScore = new Dictionary<Vector3Int, int> { { start, Heuristic(start, target) } };

**27**