

**Universitatea Tehnică a Moldovei**

**DEZVOLTAREA unui joc de tip 2d platformer**

**development of a 2d platformer game**

|  |  |
| --- | --- |
| **Student:** | **gr. TI-231,**  **Maxian Alexandru**  **Costin Ion** |
| **Coordonator:** | **Coșieru Cătălin, asist.univ.** |

**Chişinău, 2024**

**CUPRINS**

ABREVIERI ................................................................................................................................................ 1

INTRODUCERE ......................................................................................................................................... 2

1 DESCRIEREA PROIECTULUI ȘI ARHITECTURA SISTEMULUI .................................................... 3

2 ALGORITMI ȘI LOGICA JOCULUI ...................................................................................................... 4

2.1 Algoritmi de găsire a drumului optim ............................................................................................................... 5

2.2 Algoritmi de gestionare a resurselor ................................................................................................................. 9

2.3 Algoritmi de gestionare a scorurilor ............................................................................................................... 10

3 IMPLEMENTAREA PROPRIUZISĂ A ALGORITMILOR SELECTAȚI .......................................... 16

3.1 Implementarea algoritmului A\* în cadrul jocului .............................................................................................. 16

3.2 Implementarea algoritmului de prioritizarea a resurselor în cadrul jocului ............................................................. 18

3.3 Implementarea algoritmului Timsort în cadrul jocului ....................................................................................... 19

4 DOCUMENTAREA JOCULUI ............................................................................................................. 21

5 IDEI PENTRU VIITOARE ACTUALIZĂRI ........................................................................................ 24

CONCLUZII .............................................................................................................................................. 25

BIBLIOGRAFIE ........................................................................................................................................ 26

ANEXA A .................................................................................................................................................. 27

ANEXA B .................................................................................................................................................. 28

ANEXA C .................................................................................................................................................. 29

ANEXA D .................................................................................................................................................. 30

ANEXA E .................................................................................................................................................. 31

ANEXA F .................................................................................................................................................. 32

ANEXA G .................................................................................................................................................. 33

**ABREVIERI**

NPC – Non-Player-Character

DFS – Deep First Search

BFS – Breadth-First Search

Gameplay – Experiență de joc

Feedback – Evaluare

Run – Subșir

Divide et impera – Divide și stăpânește

Health – Sănătate

Energy – Energie

Armor – Armură

Frame – Cadru

Cooldown – Interval de timp

Scoreboard – Panou de scoruri

FPS – Frames per second

Text box – Căsuță de text

Game Over – Sfârșit de Joc

Update – Actualizare

Upgrade – Îmbunătățire

Jump – Săritură

Multiplayer – Mai mulți jucători

Co-op – Mod colaborativ

Versus – Adversar

Auto Team Balance – Balansare automată a echipei

Chat – Mesagerie

**INTRODUCERE**

Algoritmii joacă un rol esențial în dezvoltarea software-ului, fiind fundamentali în optimizarea proceselor și soluționarea problemelor complexe. În cadrul disciplinei Analiza și proiectarea algoritmilor, acest proiect își propune să demonstreze modul în care algoritmii pot fi aplicați eficient în dezvoltarea unui joc 2D de tip platformer, oferind o platformă ideală pentru a studia și evalua performanța diferitelor soluții algoritmice. Prin acest proiect, vom explora diverse probleme din cadrul jocurilor video, cum ar fi calculul traseelor optime, gestionarea coliziunilor, precum și optimizarea generală a resurselor și a timpilor de execuție. [1]

Jocurile video reprezintă un domeniu complex din punct de vedere algoritmic, oferind un cadru în care eficiența și scalabilitatea soluțiilor joacă un rol important în asigurarea unei experiențe fluide pentru utilizatori. Prin natura lor, jocurile de platformă 2D implică interacțiuni în timp real, ceea ce necesită algoritmi rapizi și eficienți care să gestioneze rapid diverse operațiuni precum mișcarea personajelor, detectarea coliziunilor și luarea deciziilor de către NPC-uri. În acest context, scopul principal al proiectului este analiza, implementarea și compararea diferitelor clase de algoritmi în cadrul acestor sarcini esențiale, evidențiind impactul alegerilor algoritmice asupra performanței și calității jocului.

În mod particular, raportul va examina algoritmi de găsirea a drumului minim, de asemenea algoritmi Greedy pentru gestionaera resurselor jucatorului, algoritmi de detecție a coliziunilor și algoritmi pentru interacțiunea dintre personaje și mediul înconjurător.

Fiecare dintre acești algoritmi va fi analizat în detaliu, fiind evaluată performanța lor în funcție de timpii de execuție și complexitate. O componentă esențială a acestui raport va fi comparația între diverse implementări algoritmice pentru aceleași probleme, evidențiind avantajele și dezavantajele.

Alegerea unui joc de tip platformer pentru acest proiect nu este întâmplătoare. Acest gen de jocuri este caracterizat printr-un echilibru între complexitatea algoritmică și posibilitatea de a ilustra în mod clar impactul alegerilor algoritmice asupra gameplay-ului. Prin analizarea acestor probleme într-un cadru interactiv și vizual, raportul oferă o imagine clară asupra modului în care soluțiile algoritmice influențează direct funcționalitatea și fluiditatea unui produs software, evidențiind relevanța lor în dezvoltarea aplicațiilor care necesită răspunsuri rapide și precise.

Astfel, proiectul își propune nu doar să implementeze soluții standard, ci să le analizeze și să le compare dintr-o perspectivă teoretică și practică, evaluând eficiența acestora în contextul unui mediu de joc. În final, concluziile raportului vor oferi o perspectivă asupra modului în care soluțiile algoritmice optime pot îmbunătăți experiența utilizatorului și performanța generală a aplicației.

2

**I ARHITECTURA SISTEMULUI**

În acest capitol este prezentată structura generală a jocului, componentele principale, de exemplu, mecanica jocului, motorul grafic, managementul coliziunilor, precum și modul în care acestea interacționează. De asemenea, sunt evidențiate obiectivele și funcționalitățile specifice care sunt relevante pentru implementarea algoritmilor. Aceasta va oferi o ideie asupra modului în care algoritmii se integrează în proiect.

Proiectul propune dezvoltarea unui joc de tip platformer 2D în care jucătorul controlează un personaj ce explorează un mediu virtual, sare pe platforme, și înfruntă inamici. Obiectivul principal este atingerea finalului fiecărui nivel fără a pierde toate viețile. Proiectul utilizează Python, în combinație cu Pygame, o bibliotecă populară pentru dezvoltarea de jocuri în 2D. [2]

Următoarele funcționalități de bază a jocului sunt:

* mișcare și control al personajului;
* obstacole și inamici;
* monitorizarea vieții, energieie și scorului.

Proiectul a fost dezvoltat pentru a explora aspecte fundamentale ale dezvoltării de jocuri, cum ar fi fizica mișcării, gestionarea coliziunilor, randarea graficii 2D și logica de joc.

Jocul este structurat în baza unei arhitecturi modulare, care separă diferitele componente ale jocului pentru a facilita gestionarea și extinderea ulterioară. Arhitectura jocului este bazată pe motorul jocului care este responsabil pentru gestionarea elementelor vizuale și logica de bază a jocului. Principalele funcționalități ale motorului sunt randarea graficii ce presupune că fiecare component vizual a jocului, inclusiv personajul jucătorului, inamicii și fundalul, sunt randate folosind funcțiile Pygame pentru afișarea sprite-urilor.

Update-ul logicii de joc presupune că jocul este actualizat la fiecare ciclu de buclă de joc, gestionând poziția personajului, comportamentul inamicilor și coliziunile dintre obiecte.

Obiectele jocului sunt elementele de bază care populează nivelurile, cum ar fi personajul jucătorului, inamicii, platformele și proiectilele. Exemplu de obiect a clasei jucatorului, se află în Anexa A.

Gestionarea proceselor de joc se face prin intermediul unei bucle de joc continua. Unde se citește starea tastaturii pentru a determina comenzile jucătorului și se actualizează starea jocului. Cod sursă se află în Anexa B.

3

**II ALGORITMI ȘI LOGICA JOCULUI**

În cadrul dezvoltării unui joc platformer 2D, algoritmii și logica de joc sunt necesari pentru a crea o experiență interactivă și captivantă. Aceștia gestionează toate aspectele fundamentale ale mecanicilor jocului, de la mișcarea personajului principal, până la coliziuni și comportamentul inamicilor. Acest capitol va analiza principalele elemente de logică și algoritmi utilizate în proiectul nostru, cu accent pe implementarea acestora în cod.

Să începem cu definiția algoritmului, care spune că un algoritm este un set finit de pași care rezolvă o problemă. În procesul de dezvoltare a jocului am întâlnit mai multe probleme, iar pașii în rezolvarea fiecărei probleme reprezintă algoritmii specifici pentru fiecare problemă în parte.

Pentru a pune o bază și a dezvolta ulterior jocul cu ajutorul algoritmilor mai complecși, au fost folosiți mai întâi algoritmi simpli, caracteristici tipuli dat de jocuri, aceștia fiind:

* algoritm de mișcare a jucatorului;
* algoritmi de coleziune;
* patrulare a inamicilor.

Odată ce baza jocului a fost pusă, putem trece la algoritmi mai complecși. Pentru a realiza un comportament mai natural a inamicilor se va folosi un algoritm ce constă în găsirea drumului minim de la inamic-jucător. Pentru realizarea și implimentarea acestui tip de algoritmi, platformele vor fi considerate noduri, iar distanța dintre ele muchii. Astfel, obținem un graf, Figura 2.1, în baza caruia vom realiza algoritmii propuși.

****

**Figura 2.1 – Graful imaginar al hărții**

4

**2.1 Algoritmi de găsire a drumului optim**

În acest subcapitol sunt prezentați 3 algoritmi de găsire a drumului minim, se vor compara între ei pentru a vedea avantajele și dezavantajele fiecăruia. Astfel, se alege cel mai optim și potrivit algoritm care ne va rezolva problema noastră.

Algoritmii propuși sunt:

* Dijkstra;
* Greedy Best-First Search;
* A\*.

Algoritmul Dijkstra este unul dintre cei mai cunoscuți algoritmi pentru găsirea drumului minim într-un graf, indiferent dacă are ponderi diferite sau nu. Algoritmul începe cu inițializarea distanței pentru fiecare nod cu infinit, iar distanța față de nodul de start este setată la 0, apoi se crează un set de noduri nevizitate. Pentru fiecare nod nevizitat sunt se alege nodul cu distanța minimă față de nodul de start, se actualizează distanța față de vecinii acestui nod, adică, dacă drumul prin acest nod este mai scurt, actualizezi distanța pentru nodurile vecine. După ce s-au actualizat toate distanțele vecinilor, se marchează nodul curent ca vizitat. Se repetă procesul până când toate nodurile sunt vizitate sau a fost găsit drumul minim către destinație.

Avantajele acestui algoritm sunt că el garantează găsirea drumului minim într-un graf cu ponderi positive. De asemenea, funcționează bine pentru grafuri unde toate arcele au ponderi. Însă, dezavantajele sunt că algoritmul nu este eficient pentru spații mari, precum hărți mari în jocuri, deoarece explorează toate nodurile în mod integral. În același timp nu este informativ, adică nu știe cât de departe este de țintă până când o găsește.

Complexitatea de timp a algoritmului Dijkstra este de obicei T(V²) atunci când se folosește o implementare simplă cu un tablou sau T((V + E) log V) atunci când se utilizează o coadă de priorități, unde V reprezintă numărul de vârfuri și E reprezintă numărul de muchii din graf. [3]

Complexitatea spațială auxiliară a algoritmului Dijkstra este de obicei între O(V) și O(E + V), unde V este numărul de vârfuri și E este numărul de muchii din graf, în funcție de implementare și de structurile de date utilizate. Complexitatea spațială auxiliară a algoritmului Dijkstra depinde în principal de structurile de date folosite pentru implementare, în special coada de priorități pentru gestionarea vârfurilor cu distanțele lor associate.

Complexitatea algoritmului este ilustrată în Tabelul 2.1

5

**Tabelul 2.1 – Complexitatea algoritmului Dijkstra**

|  |  |
| --- | --- |
| **Aspect** | **Complexitate** |
| Complexitatea timpului | T((V+E) log V) |
| Complexitatea spațială | O(V) |

Urmatorul algoritm este Greedy Best-First Search este un algoritm euristic ce alege întotdeauna nodul care pare cel mai aproape de destinație, în baza unei funcții euristice. De regulă, folosește distanța euclidiană sau distanța Manhattan ca metodă de a evalua vecinătatea față de nodul final.

Algoritmul Greedy Best-First Search pornește de la nodul inițial, se alege nodul care este cel mai aproape de destinație conform euristicii alese, deplasarea este ghidată doar de euristică și nu ia în calcul lungimea totală a drumului parcurs până la acel punct, algoritmul se oprește când ajunge la nodul destinație.

Principale avantaje ale algoritmului sunt că este foarte rapid și eficient pentru probleme simple unde drumul minim este foarte aproape de o linie dreaptă, și caută doar nodurile care sunt favorabile pe baza distanței estimate până la țintă. Iar, dezavantajele sunt că nu garantează găsirea drumului minim, deoarece se concentrează doar pe euristică și poate ignora drumuri mai lungi care ar fi mai scurte per total, și se poate bloca în căi care par să fie cele mai bune, dar nu sunt.

După analiza modului de funcționare a algoritmului Greedy Best-First, se analizează complexitatea. Pentru analiză următoarele caracteristici sunt:

* factorul de ramificare este B;
* adâncimea soluției este D;
* adâncimea maximă a arborelui este Dm.

Dacă spațiul de căutare este finit, atunci, similar cu căutarea în adâncime, DFS, căutarea Greedy Best-First nu este garantată să găsească drumul cel mai scurt. Deci, nu este optimă. În cel mai rău caz, complexitatea în timp a căutării Greedy Best-First este T(B^Dm), similară cu cea a DFS. Cu toate acestea, performanța căutării Greedy Best-First poate fi îmbunătățită semnificativ prin utilizarea unei funcții euristice bine concepute. În majoritatea cazurilor, complexitatea în timp a căutării Greedy Best-First poate fi mai bună decât T(B^D), care este cea a BFS.

6

Complexitatea spațială a căutării Greedy Best-First este similară cu complexitatea în timp. Căutarea Greedy Best-First funcționează foarte similar cu căutarea de cost uniform. Încearcă să prezică drumul cu costul cel mai mic de la fiecare nod până la obiectiv, folosind funcția O(n). Problema cu căutarea Greedy Best-First este că are o viziune limitată pe termen scurt. Căutarea Greedy Best-First maximizează doar avantajul pe termen scurt, care poate să nu fie cel mai bun pe termen lung.

Complexitatea algoritmului este ilustrată în Tabelul 2.2

**Tabelul 2.2 – Complexitatea algoritmului Greedy Best-First Search**

|  |  |
| --- | --- |
| **Aspect** | **Complexitate** |
| Complexitatea timpului | T(B^D) |
| Complexitatea spațială | O(n) |

Iar, al treilea algoritm propus este A\*, ce combină avantajele algoritmilor Dijkstra și Greedy Best-First Search. Este un algoritm informativ și optimal, deoarece folosește atât distanța parcursă până la un nod, cât și estimarea distanței până la destinație, euristica.

Funcția de cost A\* folosește o funcție de cost, Funcția (2.1), unde g(n)g(n) este costul drumului parcurs până la nodul curent nn., h(n) este euristica, adică estimarea costului rămas până la destinație, de obicei, distanța euclidiană sau Manhattan.

f(n)=g(n)+h(n)f(n)=g(n)+h(n) (2.1)

La fiecare pas, A\* alege nodul cu cel mai mic f(n), adică suma dintre costul drumului parcurs și estimarea costului rămas. Continuă explorarea până ajunge la nodul destinație, actualizând costurile la fel ca în Dijkstra. Avantajele acestui algoritm sunt că garantează găsirea drumului minim, atâta timp cât euristica este admisibilă, este eficient deoarece explorează doar nodurile care sunt relevante pentru a ajunge la destinație și este flexibil și poate folosi euristici variate, adaptându-se diferitelor tipuri de spații.

Însă dezavantajele sunt că dacă euristica este greșit aleasă, poate deveni mai lent decât Dijkstra și depinde de o balansare corectă între costul drumului parcurs și estimarea drumului rămas.

7

Complexitatea în timp a algoritmului A\* depinde de funcția euristică. În cel mai rău caz, într-un spațiu de căutare nelimitat, numărul de noduri extinse este exponențial în raport cu adâncimea soluției, respective complexitatea temporală devine T(b^d), unde b este factorul de ramificare, numărul mediu de succesori per stare.

Complexitatea spațială a algoritmului A\* este, de asemenea, exponențială în raport cu adâncimea soluției. În cel mai rău caz, A\* stochează toate nodurile generate, deci complexitatea spațială este O(b^d), unde b este factorul de ramificare, iar d este adâncimea soluției, drumul cel mai scurt. Acest lucru poate duce la un consum mare de memorie, mai ales în cazul problemelor cu spațiu mare de căutare.

Complexitatea algoritmului este ilustrată în Tabelul 2.3

**Tabelul 2.3 – Complexitatea algoritmului A\* (A-STAR)**

|  |  |
| --- | --- |
| **Aspect** | **Complexitate** |
| Complexitatea timpului | T(B^D) |
| Complexitatea spațială | O(B^D) |

După analiza și studierea fiecărui algoritm împarte, este necesar de făcut o primă comparație între ei, pentru a deduce care algoritm este mai potrivit pentru a fi implementat în crearea jocului propus. Prin urmare, în Tabelul 2.4, sunt încadrate și comparate proprietățile fiecărui algoritm. De asemenea, în Graficul 2.1, este reprezentat timpul de găsirea drumului minim pentru fiecare algoritm, pentru diferite mărimi de grafuri. Observânduse că, algoritmul A\* are cel mai eficient timp.

**Tabelul 2.4 – Comparația proprietăților algoritmilor**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algoritm** | **Găsește drumul minim?** | **Euristică folosită ?** | **Performanță pe grafuri mari** | **Flexibilitate** |
| **Dijkstar** | Da | Nu | Ineficient (explorează tot) | Medie |
| **Greedy Best-First** | Nu | Da | Rapid, dar nesigur | Ridicată |
| **A\*** | Da | Da | Eficient și optim | Foarte ridicată |

8

**Graficul 2.1 – Timpul de găsirea drumului minim pentru fiecare algoritm**

Pentru jocul propus, unde platformele din joc formează un graf și inamicul trebuie să găsească drumul minim către jucător, A\* este cea mai bună alegere pentru că va găsi întotdeauna cel mai scurt drum posibil între inamic și jucător, spre deosebire de Greedy Best-First Search care poate greși, de asemenea, euristica, permite de a folosi distanța euclidiană pentru a accelera procesul, fără a pierde din corectitudinea soluției. În același timp, eficiența, A\* explorează doar nodurile relevante pentru a ajunge la jucător, fiind mai rapid decât Dijkstra pe spații mari, dar menținându-și optimizarea, iar, flexibilitate, A\* poate fi ajustat ușor în funcție de harta și regulile jocului. Astfel, A\* combină rapiditatea și inteligența euristică din Greedy Best-First Search cu siguranța și completitudinea algoritmului Dijkstra, fiind soluția ideală pentru a găsi drumul minim într-un joc complex cu multe platforme și obstacole.

**2.2 Algoritmi de gestionare a resurselor**

Gestionarea resurselor este un aspect important al designului și experienței oferită de un joc video. Aduce multiple beneficii, cum ar fi creșterea complexității jocului astfel prin gestionarea atentă a resurselor, sănătate, energia și armura, jucătorii sunt obligați să ia decizii strategice care îmbogățesc experiența de joc. În același timp, o gestionare bine echilibrată a resurselor introduce un nivel potrivit de dificultate, ceea ce face jocul captivant, și stimulează luarea deciziilor pentru ca jucătorul să trebuiască să prioritizeze resursele, în funcție de necesitățile lor. Gestionarea resurselor imită situații reale, oferind o experiență mai autentică.

9

În acest joc, resursele principale, cum ar fi bară de sănătate, energie și armură, sunt gestionate atât pentru personajul principal, cât și pentru inamici. Aceasta creează un echilibru între gameplay ofensiv și defensiv. Pentru gestionarea echilibrată și eficientă a resurselor s-a ales un algoritm care preventiv a fost comparat cu alți 2 algoritmi alternativi.

Algoritmii propuși sunt:

* Prioritizarea dinamică bazată pe scoruri;
* Regenerarea ciclică fixă;
* Distribuire aleatorie a resurselor.

Prioritizare dinamică bazată pe scoruri optimizează regenerarea în funcție de situația curentă, oferind flexibilitate, dar necesită mai multe calcule în timp real. Algoritmul de regenerare ciclică fixă presupune o ordine fixă de gestionar, de exemplu mai întâi sănătate, apoi armură, apoi energie. Avantajul principal este simplitate în implementare, dar poate ignora nevoile imediate ale jucătorului, ceea ce scade eficiența. Ce ține de distribuire aleatorie a resurselor, resursele generate sunt aleatorii, fără a lua în considerare prioritățile jucătorului, în așa mod crește nivelul de imprevizibilitate, însă apare riscul să nu fie util pentru jucător dacă primește resurse care nu îi sunt necesare.

Luând în considerare avantajele și dezavantajele fiecărui algoritm, sa decis că algoritmul cel mai potrivit este algoritmul de prioritizarea resurselor prin scoruri comparative.

**2.3 Algoritmi de gestionare a scorurilor**

Gestionarea scorurilor este un element important al jocurilor video, care nu doar oferă feedback jucătorilor, ci și stimulează competiția și interacțiunea dintre aceștia. Scorurile reprezintă o măsură clară a performanței unui jucător, oferindu-le acestora un obiectiv concret și un motiv pentru a se implica activ în joc. Scorurile oferă jucătorilor un obiectiv de atins, promovând competiția și dorința de îmbunătățire. Un sistem de scor bine proiectat indică cât de bine a fost jucat un meci, ajutând utilizatorii să-și înțeleagă progresul. Salvarea și analizarea scorurilor permite studiul evoluției unui jucător în timp.

În cadrul jocului, gestionarea scorurilor implică salvarea performanțelor a mai multor jucători și ordonarea acestora pentru a evidenția cei mai buni participanți. Acest aspect permite crearea unui clasament precis și competitiv, care să reflecte în mod corect realizările fiecărui jucător. Alegerea unui algoritm eficient pentru gestionarea și sortarea scorurilor este vitală pentru asigurarea performanței și fiabilității sistemului.

10

Algoritmii propuși sunt:

* TimSort;
* MergeSort;
* QuickSort.

TimSort identifică și exploatează secvențele deja sortate din date, combinându-le ulterior într-o ordine corespunzătoare folosind o strategie similară cu MergeSort. Algoritmul divizează datele în secvențe mici, apoi se identifică porțiuni deja sortate și se sortează porțiunile scurte folosind Insertion Sort. Aceste secvențe sortate sunt combinate utilizând o strategie asemănătoare MergeSort, menținând stabilitatea. Pseudo cod al acestui algoritm se află în Anexa C. Avantajele acestui algoritm sunt că este eficient pentru datele parțial sortate, este stabil, menține ordinea originală a elementelor egale și este optim pentru aplicații reale datorită combinării dinamice a tehnicilor. Cu toate acestea, dezavantajele sunt că are o complexitate ridicată a implementării și necesită memorie suplimentară, spațiu adițional.

Complexitatea temporală a algoritmului TimSort variază în funcție de structura datelor de intrare. Cel mai bun caz este O(n), dacă datele sunt deja aproape sortate sau conțin subșiruri lungi, TimSort poate sări peste o mare parte din procesul de sortare și efectuează combinații cu un număr minim de comparații. În acest caz, utilizarea Insertion Sort este foarte eficientă, rezultând o complexitate temporală liniară. Caz mediu este O(n log n). În general, TimSort împarte tabloul în bucăți mici și efectuează combinarea acestora într-un mod stabil. Procesul de combinare domină complexitatea temporală, ceea ce duce la o complexitate de O(n log n). Acest comportament este similar cu Merge Sort. Cel mai rău caz este O(n log n). Chiar și în cel mai rău caz, de exemplu date complet nesortate, TimSort funcționează la O(n log n), deoarece se bazează pe împărțirea tablourilor și combinarea acestuia folosind Merge Sort.

TimSort necesită spațiu suplimentar pentru combinarea subșirurilor, deoarece funcționează similar cu Merge Sort. Complexitate spațială în cel mai rău caz este O(n). În cel mai rău caz, TimSort va necesita O(n) spațiu suplimentar pentru combinarea secvențelor. Aceasta se întâmplă deoarece algoritmul trebuie să creeze tablouri temporare pentru a combina două run-uri sortate în timpul procesului de merge. Complexitate spațială în cazuri bune sau medii este O(n). Chiar și în cazurile bune și medii, TimSort are o complexitate spațială de O(n) din cauza procesului de combinare a run-urilor. Totuși, în cazurile în care numărul de run-uri este mic, consumul de memorie poate fi redus, dar în cel mai rău caz complexitatea spațială rămâne liniară.

Complexitatea algoritmului este ilustrată în Tabelul 2.5

11

**Tabelul 2.5 – Complexitatea algoritmului TimSort**

|  |  |
| --- | --- |
| **Aspect** | **Complexitate** |
| Complexitatea timpului | O(nlogn) |
| Complexitatea spațială | O(n) |

Următorul algoritm, Merge Sort, este un algoritm de sortare bazat pe metoda divide et impera. Funcționează prin împărțirea repetată a listei în subliste mai mici până când fiecare sublistă conține un singur element, apoi combinarea lor într-o listă sortată. Mai exact, împarte lista în două jumătăți egale până când fiecare sublistă conține un singur element, sau zero elemente. Apoi, fuzionează sublistele sortate înapoi într-o singură listă sortată. Această operațiune necesită compararea elementelor celor două subliste și introducerea lor într-o listă ordonată.

Avantajele sunt:

* Stabilitate;
* Performanță garantată;
* Potrivit pentru liste mari.

Dezavantajele sunt:

* Utilizare mare de memorie;
* Cost ridicat pentru liste mici;
* Dificil de implementat.

Algoritmul Merge Sort împarte tabloul în două părți egale la fiecare pas, iar după împărțirea completă, începe să le interclaseze. Împărțirea tabloului este împărțit în două la fiecare pas, ceea ce înseamnă că dacă avem un tablou de dimensiune n, se va împărți în două părți de dimensiune n/2, apoi în n/4 și așa mai departe până când fiecare subtablou va conține un singur element. Această împărțire se poate face de log2​n ori. Interclasarea tablourilor După împărțirea în subtablouri de un element, acestea trebuie reîmbinate. Operația de interclasare necesită să fie parcurse toate elementele, deci are o

complexitate de O(n) pentru fiecare nivel de recursivitate. Astfel, avem log2​n nivele de recursivitate

12

și la fiecare nivel se fac O(n) operații de interclasare. Deci, complexitatea totală a algoritmului va fiT(n)=O(nlogn).

Merge Sort necesită spațiu auxiliar pentru a stoca subtablourile create în timpul interclasării. În cazul implementării standard, este necesar un spațiu suplimentar de O(n) pentru stocarea temporară a subtablourilor. Deci, complexitatea spațială este O(n).

Complexitatea algoritmului este ilustrată în Tabelul 2.6

**Tabelul 2.6 – Complexitatea algoritmului MergeSort**

|  |  |
| --- | --- |
| **Aspect** | **Complexitate** |
| Complexitatea timpului | O(nlogn) |
| Complexitatea spațială | O(n) |

Ultimul algoritm de sortare propus pentru analiză este Quick Sort. El este un algoritm de sortare rapid bazat pe metoda divide et impera, dar diferit de Merge Sort. Se bazează pe alegerea unui pivot pentru a partiționa lista. Algoritmul selectează un element din listă ca pivot, de obicei, primul, ultimul sau unul aleatoriu, apoi reorganizează lista astfel încât toate elementele mai mici decât pivotul să fie înaintea acestuia, iar toate elementele mai mari să fie după el. Algoritmul se aplică recursiv algoritmul pe sublistele rezultate, cele din stânga și dreapta pivotului.

Avantajele sunt:

* Eficiență în practică;
* Utilizare eficientă a memoriei;
* Simplu de implementat.

Dezavantajele sunt:

* Performanță variabilă;
* Sensibil la alegerea pivotului;
* Mai puțin eficient pentru liste mici.

13

Quicksort funcționează prin împărțirea repetată a unui tablou în două subtablouri pe baza unui pivot. După ce toate elementele sunt sortate în raport cu pivotul, procesul se repetă recursiv pentru subtablourile rezultate. În cazul mediu, presupunem că la fiecare pas, alegerea pivotului împarte tabloul aproximativ în două părți egale. Dacă tabloul conține n elemente, la fiecare pas de recursivitate, numărul de comparații pentru a împărți elementele este de ordinul O(n), deoarece trebuie comparate toate elementele cu pivotul. După împărțirea inițială, cele două subtablouri rezultate au, în medie, aproximativ 2n​ elemente fiecare. Această împărțire continuă până când subtablourile conțin câte un singur element.

Numărul total de niveluri de recursivitate este log2​n, similar cu Merge Sort. Astfel, pentru fiecare nivel al recursivității, facem O(n) comparații și avem O(logn) nivele. Deci complexitatea totală în cazul mediu este T(n)=O(nlogn). În cel mai bun caz, algoritmul alege mereu un pivot care împarte tabloul exact în două părți egale. Ca și în cazul mediu, se fac O(n) operații la fiecare pas și O(logn) nivele de recursivitate. Deci, în cel mai bun caz, complexitatea este tot O(nlogn). În cel mai rău caz, Quicksort alege mereu un pivot care este fie cel mai mic, fie cel mai mare element, rezultând o împărțire foarte inegală a tabloului, unde unul dintre subtablouri conține toate elementele, iar celălalt nu conține nimic. Dacă alegerea pivotului este foarte proastă, la fiecare pas rămân n−1 elemente de sortat. În acest caz, numărul de niveluri de recursivitate nu mai este O(logn), ci O(n), deoarece tabloul este împărțit dezechilibrat. Astfel, în cel mai rău caz, algoritmul efectuează O(n) operații pentru fiecare nivel și are O(n) nivele de recursivitate, rezultând o complexitate de: T(n)=O(n2).

Complexitatea spațială a algoritmului Quicksort este determinată de adâncimea stivei de apeluri recursive. În cazul mediu și cel mai bun caz, adâncimea maximă a stivei este de O(logn). În cel mai rău caz, adâncimea stivei este O(n) din cauza împărțirii dezechilibrate. Astfel, complexitatea spațială este O(logn) în medie și O(n) în cel mai rău caz.

Complexitatea algoritmului este ilustrată în Tabelul 2.7

**Tabelul 2.7 – Complexitatea algoritmului QuickSort**

|  |  |
| --- | --- |
| **Aspect** | **Complexitate** |
| Complexitatea timpului | O(nlogn) |
| Complexitatea spațială | O(logn) |

14

După analiza și studierea fiecărui algoritm împarte, este necesar de făcut o primă comparație între ei, pentru a deduce care algoritm este mai potrivit pentru a fi implementat în crearea jocului propus.

În Tabelul 2.8, sunt încadrate și comparate proprietățile fiecărui algoritm. De asemenea, în Graficul 2.2, este reprezentat timpul de sortare al algoritmilor pentru diferite mărimi de tablouri și se observă algoritmul TimSort este cel mai eficient în cazul dat.

**Tabelul 2.8 – Comparația proprietăților algoritmilor**

|  |  |  |  |
| --- | --- | --- | --- |
| **Criteriu** | **TimSort** | **MergeSort** | **QuickSort** |
| **Complexitate Timp** | O(n log n) | O(n log n) | O (n log n) |
| **Complexitate Spațiu** | O(n) | O(n) | O(log n) |
| **Stabilitate** | Da | Da | Nu |
| **Date parțial sortate** | Foarte eficient | Mai puțin optim | Ineficient |
| **Implementare practică** | Folosit în limbaje moderne | Adesea folosit pentru stabilitatea sa | Rapid, dar instabil |

**Graficul 2.2 – Timpul de sortare al algoritmilor**

Pentru jocul propus, utilizarea TimSort este potrivită pentru gestionarea eficientă a datelor salvate, stabilitatea algoritmului și performanța superioară în practică. Acest algoritm, creat special pentru a îmbunătăți gestionarea listelor reale de date, combină avantajele MergeSort și Insertion Sort pentru a funcționa rapid și eficient. Scorurile jucătorilor vor fi adăugate și actualizate constant, iar multe dintre acestea pot fi deja parțial sortate, de exemplu, când un nou scor este adăugat la o listă sortată. Această situație este frecvent întâlnită în jocurile competitive, unde clasamentele sunt actualizate în timp real. TimSort optimizează acest tip de scenariu, detectând secvențele sortate dinainte și reducând astfel efortul necesar pentru sortarea completă. În plus, algoritmul minimizează numărul de comparații și mișcări de date, ceea ce duce la un timp de execuție redus. TimSort păstrează ordinea originală a jucătorilor cu scoruri egale, ceea ce este important pentru o afișare corectă în clasament.

15

**III IMPLEMENTAREA PROPRIUZISĂ A ALGORITMILOR SELECTAȚI**

În procesul de dezvoltare a unui joc platformer 2D, utilizarea unor algoritmi bine concepuți reprezintă un aspect important pentru a asigura atât funcționarea corectă a mecanismelor de joc, cât și o experiență plăcută pentru utilizatori. Acest capitol se concentrează pe prezentarea detaliată a implementării și utilizării a trei algoritmi importanți: A\*, TimSort și un algoritm personalizat pentru gestionarea resurselor, fiecare fiind integrat pentru a rezolva probleme specifice din cadrul jocului.

Primul algoritm, A\*, este folosit pentru a calcula drumul optim al inamicilor către jucător, pe baza pozițiilor acestora și a platformelor disponibile. Acesta adaugă o dimensiune strategică, oferind o mișcare inteligentă a inamicilor și provocări dinamice pentru utilizator. Algoritmul utilizează o combinație de funcții euristice și costuri de traversare pentru a determina cea mai scurtă cale pe un grafic de platforme interconectate.

Al doilea algoritm este acel de gestionare a resurselor este implementat pentru a prioritiza automat nevoile jucătorului, cum ar fi regenerarea sănătății, armurii și energiei. Acesta evaluează în timp real starea resurselor disponibile și decide care dintre ele necesită suplimentare, contribuind astfel la menținerea echilibrului dintre dificultate și șansele de supraviețuire ale jucătorului.

În final, TimSort, este integrat pentru sortarea eficientă a datelor legate de clasamentele jucătorilor. TimSort combină sortarea prin inserție și sortarea prin interclasare, oferind performanțe ridicate chiar și pentru seturi mari de date. Această funcționalitate asigură afișarea rapidă a unui Top 10 jucători, îmbunătățind interacțiunea utilizatorilor cu sistemul.

În cadrul subcapitolelor următoare, fiecare dintre acești algoritmi va fi analizat detaliat. Vor fi explicate logica implementării, modul în care interacționează cu celelalte componente ale jocului și impactul asupra experienței finale oferite jucătorilor.

**3.1 Implementarea algoritmului A\* în cadrul jocului**

Algoritmul A\* este unul dintre cei mai cunoscuți algoritmi de căutare folosiți pentru a găsi drumul optim între două puncte pe un grafic. În proiectarea jocului, A\* este folosit pentru a calcula drumul optim al inamicilor de pe platforma lor actuală către platforma unde se află jucătorul. Acest lucru contribuie la crearea unei provocări dinamice și strategice pentru jucător, deoarece inamicii nu se mișcă aleatoriu, ci urmează un traseu logic pentru a-l ajunge pe jucător.

Algoritmul A\* combină costul real, definit în program sub numele g-cost, ce reprezintă costul deja parcurs de la punctul de plecare până la nodul curent. Iar, costul estimate, definit ca h-cost, este folosit pentru estimarea distanței rămase de la nodul curent la destinație, folosind o funcție euristică, în acest caz fiind distanța euclidiană. Aceasta se bazează pe Formula (3.1), clasică din geometrie.

16

(3.1)

Funcția care determină cât de promițător este un nod în căutarea drumului optim este data de către funcția f-cost, Funcția (3.2). [4]

f(n) = g(n) + h(n) (3.2)

Scopul algoritmului este să minimizeze f-cost-ul pentru a găsi cel mai scurt drum între punctul de start și cel final.

În joc, graful este reprezentat printr-o serie de platforme interconectate, unde fiecare platformă este un nod, iar legăturile dintre ele sunt muchii cu un anumit cost. Costul unei muchii reprezintă distanța dintre două platforme vecine. Fiecare platformă este un obiect al clasei Platrforms, care conține attribute ca dimensiunile platformei, coordonatele platformei, vecinii platformei și costurile de tranziție către aceștia, stocate într-o listă. Pe lângă clasa Platforms, avem nevoie de clasa Graf pentru a gestiona platformele și legăturile dintre ele. Platformele sunt stocate într-un tablou sub numele de self.platforme, unde cheia este numărul platformei. Legăturile sunt adăugate folosind funcția adauga\_legatura, care calculează costul dintre două platforme și le adaugă ca vecini. În Anexa D avem codul claselor Platforms și Graf.

Funcționarea algoritmului A\* în joc începe prin inițializarea. Când inamicul trebuie să determine drumul către jucător, algoritmul este inițiat cu platforma de start, platforma curentă a inamicului, și platforma țintă, platforma pe care se află jucătorul. Un exemplu practic este situația în care inamicul se află pe platforma de jos, iar jucătorul pe o platformă superioară, la o anumită distanță. A doua etapă este căutarea drumului, în care algoritmul evaluează nodurile în funcție de costul lor total. Începe cu platforma de start și explorează vecinii acesteia, folosind distanța euclidiană dintre platforma curentă și platforma țintă se determină costul estimat până la destinație. Nodurile sunt prioritizate în funcție de aceste costuri, iar algoritmul continuă până când găsește platforma țintă sau nu mai există noduri de explorat. Odată ce platforma țintă este atinsă, algoritmul reconstruiește drumul de la destinație către sursă. Acest drum este apoi utilizat pentru a ghida inamicul către jucător. Funcția a\_star, reprezentată în Anexa E, implementează logica algoritmului. Aceasta identifică drumul optim între două platforme.

În cadrul jocului, drumul calculat de A\* este utilizat pentru a deplasa inamicul pas cu pas pe drumul optim. Dacă drumul implică mai multe platforme intermediare, inamicul se teleportează sau se deplasează în mod realist între acestea. Dacă inamicul este pe platforma de jos și trebuie să ajungă la o platformă mai înaltă, acesta calculează un traseu format din mai multe salturi între platforme

17

intermediare. În cazul în care poziția jucătorului se schimbă în timpul deplasării, algoritmul este recalculat, iar inamicul ajustează traseul. Algoritmul garantează comportament dinamic, astfel inamicii se adaptează în funcție de poziția jucătorului, oferind o experiență mai interactivă. Algoritmul minimizează traseul, reducând timpul necesar pentru ca inamicul să ajungă la jucător. De asemenea, sistemul permite adăugarea sau modificarea platformelor fără a afecta funcționalitatea algoritmului.

Se presupune că jucătorul se află pe platforma 5 și inamicul începe pe platforma 1. Atunci, algoritmul A\* calculează traseul optim 1, 2, 4, 5.

**3.2 Implementarea algoritmului de prioritizarea a resurselor în cadrul jocului**

Algoritmul de prioritizare a resurselor implementat în codul jocului este conceput pentru a evalua starea actuală a trei resurse cheie ale jucătorului: health, energy și armor. Acesta decide care resursă are cea mai mare nevoie de regenerare, ghidând astfel generarea de bonusuri în joc. Funcția prioritize\_bars, atașată în Anexa F, este responsabilă pentru calcularea priorității dintre cele trei resurse. Parametrii health, armor și energy, reprezentați în Figura 3.1, sunt valorile curente ale resurselor jucătorului.



**Figura 3.1 - Parametrii health, armor și energy**

Metodologia presupune calcularea procentajul rămas pentru fiecare resursă față de valoarea maximă, ea fiind 100%. Se generează un scor invers proporțional cu procentajul rămas, cu cât resursa este mai scăzută, cu atât scorul este mai mare. În final, se returnează resursa cu prioritatea cea mai mare, cea mai scăzută valoare curentă. Pe baza resursei prioretizate identificate, jocul generează un bonus pe o platformă aleatorea. Acest bonus fiind reprezentat în Figura 3.2, sub forma de o bilă roșie, albastră sau sură, în dependență de necesitățile jucătorului.

18



**Figura 3.2 – Bonusul generat pe hartă pentru regenerarea sănătății**

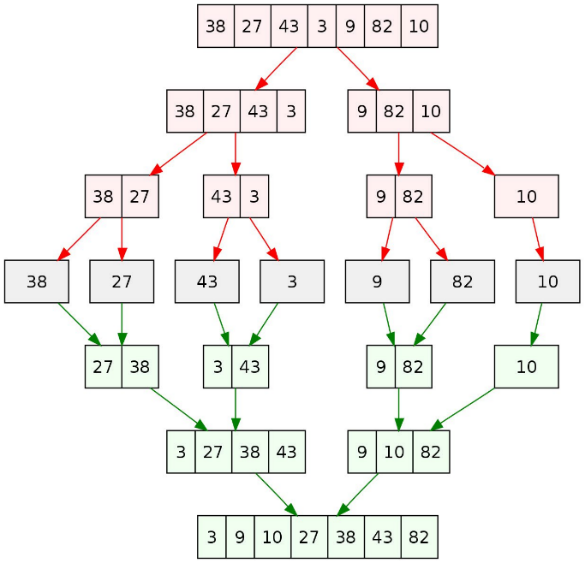
Când jucătorul se apropie suficient de aproape de cadou, resursa respectivă este regenerată. Cadourile de sănătate regenerează lent sănătatea la o rată de +0.1 per frame, cadourile de energie regenerează energia instantaneu la 100% și cadourile de armură regenerează gradual armura la o rată de +0.5 per frame. Dacă energia jucătorului scade și nu există un bonus de energie pe hartă, aceasta se regenerează automat la intervale regulate la fiecare cooldown setat la fiecare 5 secunde, cu un increment de 10%. La fiecare frame, este apelată funcția care determină ce resursă necesită prioritate. Dacă prioritatea există, jocul generează un bonus corespunzător pe o platformă. Jucătorul poate colecta bonusul pentru a regenera resursa indicată. Dacă nu sunt bonusuri disponibile, regenerarea automată a energiei menține jucătorul funcțional. Acest algoritm de gestionare a resurselor adaugă dinamică, flexibilitate și echilibru în joc. Prioritizarea în timp real a resurselor creează o experiență de joc mai captivantă, algoritmul poate fi extins pentru a include noi tipuri de resurse sau reguli suplimentare și jucătorul este ajutat să rămână în viață chiar și în condiții dificile prin regenerare automată sau distribuția strategică a bonusurilor.

**3.3 Implementarea algoritmului TimSort în cadrul jocului**

TimSort este un algoritm de sortare hibrid, extrem de eficient, utilizat pentru a combina avantajele sortării prin inserție și sortării prin interclasare. TimSort este utilizat pentru a sorta scorurile jucătorilor în funcție de punctaj, în ordine descrescătoare, pentru afișarea unui Top 10 jucători. În codul tău, TimSort este utilizat pentru a sorta scorurile jucătorilor salvați în fișierul data\_file.txt. Datele sunt citite din fișier și stocate într-o listă Python. Apoi, algoritmul TimSort sortează lista în funcție de scorurile jucătorilor. În final, după sortare, primii 10 jucători cu cele mai mari scoruri sunt afișați pe ecran. Acest flux asigură că doar cei mai buni jucători sunt afișați.

19

Algoritmul începe prin împărțirea listei în bucăți mici, runs, de lungime minimă definite. Pe aceste subsecvențe mici, TimSort folosește sortarea prin inserție, deoarece aceasta este rapidă pentru seturi mici de date. Subsecvențele sortate sunt apoi combinate folosind sortarea prin interclasare, care este eficientă pentru a uni liste mari deja sortate. Algoritmul TimSort împreună cu funcțiile InsertionSort și MergeSort sunt incluse în Anexa G. De asemenea, în Figura 3.3, este reprezentat procesul de sortare a algoritmului TimSort.



**Figura 3.3 – Procesul de sortare a algoritmului TimSort**

Unde se observă cum lista inițială 38, 27, 43, 3, 9, 82, 10 este împărțită în segmente mici, lista a fost împărțită în două grupuri, 38, 27, 43, 3 de stânga și 9, 82, 10 de dreapta. Apoi fiecare run este sortat individual folosind sortarea prin inserție. Se vede cum tabloul 38, 27, 43, 3, este împărțit și apoi sortat treptat în 3, 27, 38, 43, iar 9, 82, 10 devine 9, 10, 82. După ce fiecare run este sortat, algoritmul le combină folosind tehnica de interclasare. Interclasarea finală combină 3, 27, 38, 43 și 9, 10, 82, pentru a obține lista 3, 9, 10, 27, 38, 43, 82, complet sortată.

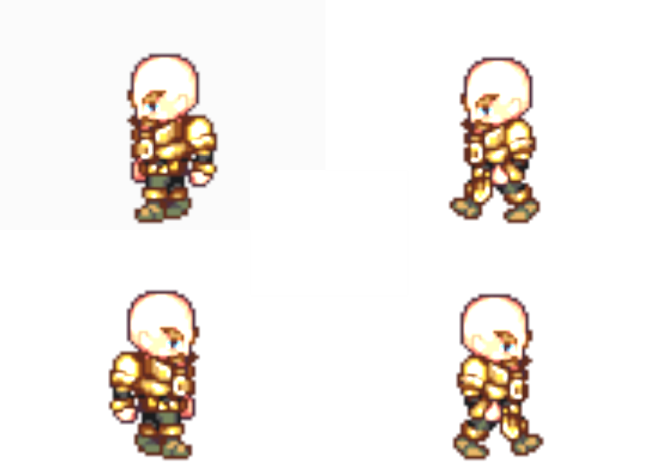
După înțelegerea modului de funcționare și aplicare a algoritmului, a fost implementat în codul jocului, iar acum fiecare jucător poate accesa din meniul jocului, apăsând butonul Scoreboard, panoul de scoruri cu jucătorii top.

20

**IV DOCUMENTAREA JOCULUI**

Aplicația dezvoltată este un joc de tip 2D Platformer care are la bază evitarea, supraviețuirea și eliminarea inamicului ce se află pe hartă cu jucătorul. Scopul jucătorilor este de a obține un scor cât mai mare prin eliminarea inamicului de câte mai multe ori. Pe parcurs, dificultatea crește prin faptul că inamicul devine cât mai rezistent la atacuri. Atât jucătorul, cât și inamicul, sunt capabili de a lansa proiectile, iar jocul detectează dacă a existat coleziune între proiectilul jucătorului cu inamic și proiectilul inamicului cu jucătorul. Prin urmare, dacă au loc coleziunile, jucătorul sau inamicul pierd un procentaj din sănătatea lor. De asemenea, jucătorul are la dispoziție 3 tipuri de resurse, acestea fiind sănătatea, energia și armura. Aceste resurse pot fi regenerate cu ajutorul bonusurilor ce sunt generate de un algoritm specific jocului. În joc sunt animații de deplasare a proiectilelor, a jucătorului și a inamicului.

În Figura 4.1, sunt reprezentate 4 frame-uri a personajului care se interschimnă în funcție de acțiune, la o rată de 60 FPS. [5]



**Figura 4.1 – Frame-urile caracterului**

Până la începerea jocului propriuzis, player-ul este întâmpinat de un meniu, reprezentat în Figura 4.2. Meniul oferă jucătorului 3 opțiuni. Jucătorul poate alege să înceapă un nou joc prin apăsarea butonului Start Game, poate alege sa vizualizeze panoul de scoruri prin apăsarea butonului Scoreboard sau poate închide jocul prin apăsarea butonului Exit.

21



**Figura 4.2 – Meniul jocului**

În Figura 4.3, este ilustrat panoul de scoruri, ce apare la apăsarea butonului Scoreboard. Panoul de scoruri afișează în ordine descrescătoare scorurile la cei Top 10 jucători cu cele mai mari scoruri. Iar, prin apăsarea butonului Back to menu, jucătorul revine la meniul principal al jocului.



**Figura 4.3 – Panoul de scoruri**

Revenind la meniul principal, prin apăsarea butonului Start Game, se afișează pe ecran un text box, prezentată în Figura 4.4, în care jucătorul își poate introduce numele dorit care se salvează într-un fișier current.txt, ca apoi la finalul jocului numele jucătorului împreună cu scorul său sunt transferate în fișierul data\_file.txt, unde se află toate rezultatele jucătorilor.

22



**Figura 4.4 – Căsuța de text pentru numele jucătorului**

După introducerea numelui în căsuța de text, player-ul poate să înceapă jocul. În figura 4.5, este redată harta jocului, ce cuprinde platformele, player-ul, inamicul, bonusurile și UI-ul, ce la rândul său cuprinde indicatorii de resurse și scorul curent acumulat.



**Figura 4.5 – Interfața principală a jocului**

În final, când jucătorul rămâne fără viață, health, pe ecran se afișează mesajul Game Over și apare opțiunea de reântoarcere în meniu și începerea unui nou meci.

23

**V IDEI PENTRU VIITOARE ACTUALIZĂRI**

Cu toate că jocul deja poate capta atenția jucătorului, există multe moduri de a îmbunătăți experiența de joc.

Ca primă ideie care va fi realizată este implementarea unui algoritm genetic și aplicarea acestuia asupra unui nou inamic, care respective va avea un comportament mai natural, imprevizibil și potrivit pentru anumite situații.

În joc se simte lipsa de dezvoltare prin lipsa de niveluri și upgrade-uri pentru caracterul jucătorului. Prin prezența nivelurilor, pot fi implementate noi tipuri de inamici sau modificări a hărții întrucât să fie o experiența diferită de fiecare data când un nou meci este început. Upgrade-urile asupra echipamentului sau fizicul caracterului, pot aduce diversificare în experiența de joc dar și în modul în care diferite situații de ofensivă sau defensivă sunt abordate.

O îmbunătățire a fizicii jocului este o următoare prioritate spre îmbunătățirea gameplay-ului. Se va adăuga funcția de jump, atât cât pentru player, cât și pentru inamic. De asemenea, este posibil de adăugat efecte sonore și vizuale, pentru aprofundare complete a player-ului în joc.

O ideie care persist încă de la începutul dezvoltării acestui joc, este de a crea un mod multiplayer. Care ar aduce mai mulți jucători și ar face jocul mai popular și competitiv. Modul multiplayer va conține mai multe moduri de joc. Aceste moduri vor fi co-op și versus. Co-op va oferi posibilitatea de a juca în echipă cu o altă persoană reală, împotriva înamicilor jocului. Iar, modul versus va oferi posibilitatea de a juca împotriva unei persoane reale. Un algoritm propus pentru modul versus este unul pentru Auto Team Balance, această funcționalitate fiind garantată de un algoritm de tip Greedy.

Important și esențial este de a îmbunătăți grafica jocului. Prin redactarea texturilor mai detaliate și a model-urilor caracterelor și inamicilor. Bonusurile vor avea o aparență mai clară pe hartă și nu se vor pierde din vedere. De asemenea, la adăugarea unor noi arme și echipamente cu texturi bine definite, jucătorului îi va fi mult mai ușor să se aprofundeze în lumea jocului.

O ultima ideie posibilă de realizat este de a crea o experiență narativă prin implementarea dialogului dintre jucător și inamic. Iar în cazul modului multiplayer, este necesar de a face posibilă comunicarea printr-un chat sau mesaje predefinite.

24

**CONCLUZII**

Acest proiect de an reprezintă o demonstrație clara a aplicării tehnicilor de analiză algoritmică și de dezvoltare software într-un context interactiv și atractiv, specific jocurilor video. Realizat cu o metodologie bine aranjata și structurat pe mai multe etape fundamentale, proiectul reflectă nu doar capacitate, ci și înțelegerea profundă a mecanicilor de joc, a arhitecturii sistemelor și a principiilor de design interactiv.

Un element de baza al lucrării este integrarea algoritmilor utilizati pentru gestionarea principalelor componente ale jocului, fiecare contribuind mult la funcționalitatea și fluiditatea experienței utilizatorului. Algoritmul A\*, utilizat pentru determinarea drumului optim al inamicilor către jucător, a fost ales și implementat cu o abordare exacta. Prin utilizarea combinației dintre costuri reale și estimative, algoritmul oferă un comportament strategic inamicilor, creând o provocare dinamică și captivantă pentru jucători. Această implementare nu doar asigură un gameplay realist, ci și demonstrează abilitatea de a adapta concepte teoretice complexe la cerințele unui mediu interactiv.

În plus, integrarea algoritmului TimSort pentru sortarea clasamentelor accentueaza o abordare inclinata către eficiență și performanță. Prin stabilitatea sa și capacitatea de a lucra cu seturi mari de date, acest algoritm îmbunătățește sistemul de afișare a scorurilor, oferind o buna experiență chiar și în condiții de utilizare intensă. Prin gestionarea inteligentă a regenerării sănătății, armurii și energiei, proiectul creează o experiență de joc echilibrată și captivantă, Dincolo de realizările tehnice, proiectul se evidentiază și prin potențialul său de extindere. Propunerile pentru actualizări viitoare, precum integrarea unui mod multiplayer, adăugarea unor niveluri și îmbunătățirea graficii, indica spre o viziune ambitioasa asupra direcțiilor de dezvoltare. Proectul are un potential de dezvoltare si evoluare. În proect prin utilizarea unor paradigme și instrumente diverse, precum motorul Pygame, programarea modulară și analiza comparativă a algoritmilor, lucrarea oferă o perspectivă pe inteles despre procesului de dezvoltare a aplicațiilor complexe. Alegerea unui joc de tip platformer, un gen cunoscut pentru echilibrul său între simplitate și complexitate, a permis explorarea unor carcteristici de baza, precum detectarea coliziunilor, gestionarea resurselor și optimizarea drumurilor. Prin abordarea teoretică și implementarea practică , proiectul demonstrează impactul direct al soluțiilor algoritmice asupra performanței și calității unui produs software. În același timp, evidențiază relevanța cunoștințelor acumulate în cadrul disciplinei Analiza și proiectarea algoritmilor, subliniind modul în care acestea pot fi aplicate într-un context practic și creativ.

Astfel, proectul si-a indeplinit scopul principal de înțelegerea a rolului algoritmilor în dezvoltarea de produse software interactive. Rezultatul final al acestui proiect constă în îmbinarea eficienta a teoriei și practicii, oferind un exemplu potrivit al modului în care i creativitatea poate transforma cunoștințele tehnice în experiențe captivante pentru utilizatori.

25

**BIBLIOGRAFIE**

1. [support@steampowered.com](mailto:support@steampowered.com), “Steam.” [Online]. Available: <https://store.steampowered.com/search/?tags=5379&supportedlang=english&ndl=1>
2. “tutorialspoint.” [Online]. Available: <https://www.tutorialspoint.com/pygame/pygame_locals_module.htm>
3. “Else.” [Online]. Available: <https://else.fcim.utm.md/pluginfile.php/78743/mod_resource/content/0/alg2004_curs4-5.pdf>
4. [patrick@policyalmanac.org](mailto:patrick@policyalmanac.org), Patrick Lester, “A\* Pathfinding for Beginners.” [Online]. Available: <https://web.archive.org/web/20171022224528/http://www.policyalmanac.org:80/games/aStarTutorial.htm>
5. [dsacompliance@udemy.com](mailto:dsacompliance@udemy.com), “udemy.” [Online] Available: <https://www.udemy.com/course/learn-to-code-by-making-games-in-python/?couponCode=BFCPSALE24\>

26

**ANEXA A**

**Crearea obiectului piersonajului**

class player():

def init(self, x, y, width, height):

self.x = x

self.y = y

self.prev\_y=0

self.width = width

self.height = height

self.step = 5

self.is\_jump = False

self.left = False

self.right = False

self.walk\_count = 0

self.jump\_count=True

self.jump\_speed=0.15

self.repause = True

self.hitbox = (self.x + 17, self.y, 32, 70)

self.platform=0

jony = player(50, 730, 64, 64)

27

**ANEXA B**

**Bucla principală a jocului**

while run:

clock.tick(FPS)

for event in pygame.event.get():

if event.type == pygame.QUIT:

run = False

for bomb in bombs:

if bomb.y - bomb.radius < goblin.hitbox[1] + goblin.hitbox[3] and bomb.y + bomb.radius > goblin.hitbox[1]:

if bomb.x + bomb.radius > goblin.hitbox[0] and bomb.x - bomb.radius < goblin.hitbox[0] + goblin.hitbox[2]:

goblin.hit()

bombs.pop(bombs.index(bomb))

if 0 < bomb.x < 1700:

bomb.x += bomb.vel

else:

bombs.pop(bombs.index(bomb))

keys = pygame.key.get\_pressed()

if keys[pygame.K\_SPACE]:

facing = -1 if jony.left else 1

if len(bombs) < 5:

bombs.append(projectil(round(jony.x + jony.width // 2), round(jony.y + jony.height // 2), 5, (247, 120, 2), facing))

if keys[pygame.K\_LEFT] and jony.x > 0:

jony.x -= jony.step,jony.left = True,jony.right = False,jony.repause = False,

elif keys[pygame.K\_RIGHT] and jony.x + jony.width < win\_width:

jony.x += jony.step,jony.right = True,jony.left = False,jony.repause = False

else:

jony.repause = True, jony.walk\_count = 0

current\_time = pygame.time.get\_ticks()

if keys[pygame.K\_UP] and (current\_time - last\_jump\_time > jump\_cooldown):

last\_jump\_time = current\_time,jony.is\_jump = True,jony.walk\_count = 0

jony.y=jony.on\_platform(platform\_1,platform\_2,platform\_3,platform\_4,platform\_5,platform\_6,platform\_8,platform\_7,platform\_9,platform\_11,platform\_10,platform\_13,platform\_12)

jony.y=jony.fall(platform\_1,platform\_2,platform\_3,platform\_4,platform\_5,platform\_6,platform\_8,platform\_7,platform\_9,platform\_11,platform\_10,platform\_13,platform\_12)

redraw\_game\_window()

28

**ANEXA C**

**Pseudo cod pentru algoritmul Timsort**

FUNCTION timSort(array)

n = LENGTH(array)

# Sortare run-uri folosind Insertion Sort

FOR i FROM 0 TO n STEP MIN\_RUN

insertionSort(array, i, MIN(i + MIN\_RUN - 1, n - 1))

# Mărirea run-urilor și combinarea lor

size = MIN\_RUN

WHILE size < n

FOR start FROM 0 TO n STEP size \* 2

mid = MIN(n - 1, start + size - 1)

end = MIN(start + 2 \* size - 1, n - 1)

IF mid < end

merge(array, start, mid, end)

size = size \* 2

END FUNCTION

29

**ANEXA D**

**Clasa Platforms și Graf**

class Platrforms:

def init(self, platform\_number, x, y, width, height):

self.platform\_number = platform\_number

self.x = x

self.y = y

self.width = width

self.height = height

self.vecini = []

class Graf:

def init(self):

self.platforme = {}

def adauga\_platforma(self, number, x, y, w, h):

platforma = Platrforms(number, x, y, w, h)

self.platforme[number] = platforma

return platforma

def adauga\_legatura(self, num\_1, num\_2, cost):

if num\_1 in self.platforme and num\_2 in self.platforme:

self.platforme[num\_1].adauga\_vecin(self.platforme[num\_2], cost)

self.platforme[num\_2].adauga\_vecin(self.platforme[num\_1], cost)

30

**ANEXA E**

**Implementarea algoritmului A\***

def a\_star(self, start, end):

deschis = []

heapq.heappush(deschis, (0, self.platforme[start]))

costuri = {nume: float('inf') for nume in self.platforme}

costuri[start] = 0

parinte = {nume: None for nume in self.platforme}

while deschis:

f\_current, current = heapq.heappop(deschis)

if current.platform\_number == end:

drum = []

while current:

drum.append(current.platform\_number)

current = parinte[current.platform\_number]

return drum[::-1]

for vecin, cost in current.vecini:

g\_cost = costuri[current.platform\_number] + cost

f\_cost = g\_cost + current.distanta\_euclidiana(self.platforme[end])

if g\_cost < costuri[vecin.platform\_number]:

costuri[vecin.platform\_number] = g\_cost

parinte[vecin.platform\_number] = current

heapq.heappush(deschis, (f\_cost, vecin))

return None

31

**ANEXA F**

**Implementarea algoritmului de prioritizarea a resurselor în cadrul jocului**

def prioritize\_bars(health, armor, energy):

max\_health, max\_armor, max\_energy = 100, 100, 100

health\_percent = (health / max\_health) \* 100

armor\_percent = (armor / max\_armor) \* 100

energy\_percent = (energy / max\_energy) \* 100

scores = {

"health": 100 - health\_percent,

"armor": 100 - armor\_percent,

"energy": 100 - energy\_percent

}

sorted\_bars = sorted(scores.items(), key=lambda x: x[1], reverse=True)

return sorted\_bars[0][0]

32

**ANEXA G**

**Implementarea algoritmului TimSort în cadrul jocului**

MIN\_RUN = 32

def insertion\_sort(arr, left, right):

for i in range(left + 1, right + 1):

key = arr[i]

j = i - 1

while j >= left and arr[j][1] < key[1]:

arr[j + 1] = arr[j]

j -= 1

arr[j + 1] = key

def merge(arr, left, mid, right):

left\_part = arr[left:mid + 1]

right\_part = arr[mid + 1:right + 1]

i, j, k = 0, 0, left

while i < len(left\_part) and j < len(right\_part):

if left\_part[i][1] >= right\_part[j][1]:

arr[k] = left\_part[i]

i += 1

else:

arr[k] = right\_part[j]

j += 1

k += 1

while i < len(left\_part):

arr[k] = left\_part[i]

i += 1

k += 1

while j < len(right\_part):

arr[k] = right\_part[j]

j += 1

k += 1

def tim\_sort(arr):

n = len(arr)

for i in range(0, n, MIN\_RUN):

insertion\_sort(arr, i, min(i + MIN\_RUN - 1, n - 1))

size = MIN\_RUN

while size < n:

for start in range(0, n, size \* 2):

mid = min(n - 1, start + size - 1)

end = min((start + 2 \* size - 1), (n - 1))

if mid < end:

merge(arr, start, mid, end)

size \*= 2

33