**DUNGEON ETERNAL**

**Candidat: Daniel Iurașoc**

**Coordonator științific: Conf.dr.ing. Răzvan Bogdan**

Sesiunea: Septembrie 2022

**REZUMAT**

Dungeon Eternal este un joc video de tip RPG [1], cu perspectiva top-down (privit de sus), în care jucătorul controlează un caracter cu care poate explora diverse încăperi din joc, în care întâlnește diverși inamici cu care trebuie sa lupte, poate interacționa cu diverse caractere prestabilite care îl ghidează pe parcursul nivelelor și îl ajută în îmbunătățirea echipamentului.

Inițial jucătorul este plasat într-o încăpere specială unde poate afla de la un caracter NPC [2] controalele caracterului său, ce trebuie să facă. De asemenea, în acea încăpere există un inamic de antrenament cu ajutorul căruia jucătorul poate să se familiarizeze cu sistemul de luptă din joc. Pentru a începe progresul, accesul la următoarea cameră (dungeon) se face printr-un tunel care este blocat de câteva cutii, care pot fi sparte cu sabia.

Fiecare noua cameră conține un numar prestabilit de inamici de dificultăți diferite, pentru a face lupta mai interesantă. Dupa eliminarea tuturor inamicilor din camera respecivă, un cufăr cu resurse apare din locul unde au apărut inamicii, apoi jucătorul poate trece la urmatorea cameră. Camera finala din fiecare nivel conține câteva obiecte speciale, și anume un portal către o cameră plină cu cufere de resurse (Treasure Dungeon), un portal către lupta finală cu inamicul special mai puternic (Boss Level), care este pragul către urmatorul nivel din joc, precum și un portal către o temință specială în care jucătorul poate lupta un număr infinit de inamici (grupați în valuri) cu scopul de a aduna resurse suplimentare.

Cosmetica jocului este creată folosind grafică de tip PixelArt [3], un stil de arta nostalgic, care ajută în crearea unei atmosfere de temniță. Jocul dispune, de asemenea și de o coloană de sunet simplă (muzică de fundal, sunet la lovire, sunet la moarte, sunet la interacțiunea cu elementele din jur).

Pentru dezvoltarea jocului a fost folosita platforma de dezvoltare Unity [4], care a fost folosită, de asemenea, și pentru realizarea animațiilor din joc. Limbajul de programare folosit este C#, iar ca editor de cod a fost preferat Visual Studio Community 2022.

**CUPRINS**

1. **INTRODUCERE4**

1.1Descrierea temei4

1.2 Motivația alegerii temei5

1. **ANALIZA DOMENIULUI6**

2.1 Aplicații similare existente6

1. **TEHNOLOGII FOLOSITE8**

3.1 Sistemul de operare și platforma de dezvoltare alese8

3.1.1 Avantajele și dezavantajele acesteia8

3.2 Platforma de dezvoltare Unity9

3.2.1 Utilizarea tehnologiilor specifice Unity în cadrul proiectului11

1. **ARHITECTURA JOCULUI12**

4.1 Arhitectura sistemului 12

4.2 Structura scenelor implementate în interiorul jocului13

4.3 Arhitectura software17

4.3.1 Prezentarea modulelor software 18

1. **IMPLEMENTAREA FUNCȚIONALITĂȚILOR37**

5.1 Diagrama Use Case 37

5.2 Controlul mișcarilor37

5.2.1 Controlul caracterului jucătorului 40

5.2.2 Controlul inamicilor41

5.3 Sistemul de luptă44

5.3.1 Implementarea acestuia pentru caracterul jucătorului 45

5.3.2 Implementarea acesuia pentru inamici 47

5.4 Administrarea resurselor și a progresului 48

5.5 Efectele sonore 51

5.6 Animațiile 52

1. **Testarea pe alte configurații55**
2. **Concluzii59**
3. **Bibliografie60**

# INTRODUCERE

## DESCRIEREA TEMEI

Pentru lucrarea de licența am dezvoltat un joc de tip RPG (Role Playing Game) [1], în care jucătorul controlează un caracter și interacționează cu inamici de diferite dificultăți, precum și NPC-uri (Non Playable Character) [2] care îl ghidează în învățarea jocului precum și elemente de decor (spre exemplu cufere cu bunuri, cutii care blochează calea, fântâni de vindecare, portale de teleportare, etc).

Arta/Cosmeticile jocului sunt realizate în stilul Pixel Art [3], un stil de artă retro care se folosește și astăzi pentru efectul de nostalgie pe care îl aduce, precum si datorită ușurinței de folosire în dezvoltare. La baza jocului stă o imagine de dimensiuni 256x256 pixeli, împărțită in pătrate de 16x16 pixeli în care se incadrează toate elementele jocului, atât cele de decor (pereți, podea, scări, stâlpi, etc), precum și inamicii, caracterele de joc, NPC-urile, cuferele, cutiile, etc.

Jucătorul dispune de un sistem de inventar care gestionează cantitatea de gold pe care o deține, precum și armurile/armele de diferite categorii pe care le poate folosi pentru a deveni mai puternic. Caracterul are un sistem de viață și experiența sub forma a două variabile intregi, care se actualizează în timp real, când acesta este lovit/vindecat, sau câștigă experiență prin lupta cu inamicii.

Echipamentul din joc este împarțit in mai multe categorii, și anume : bronze, silver, gold, diamond, cu bronze fiind cea mai slabă si diamond cea mai puternică. Fiecare din aceste categorii are caracteristici aparte, care cresc cu cat echipamentul este mai puternic (puncte de viață extra, rezistență la împingere, daunele provocate de armă, puterea de împingere a armei, etc).

Inamicii sunt împărțiți pe mai multe categorii, și anume : mici, medii, mari, și imanicii finali(boss level). Fiecare categorie are mai multe imagini din care este aleasă una la întâmplare când sunt creați în joc, iar puterea acestora(daunele provocate, puterea de împingere), precum si conținutul de experiență pe care îl oferă când sunt înfrânți sunt diferite pentru fiecare categorie în parte.

Sunetele din joc au fost preluate de pe site-ul itch.io [5], care conține resurse audio sau grafice gratis/contra-cost pentru dezvoltatorii de jocuri. Coloana sonora conține sunete la mișcarea caracterului, sunet la folosirea armei, la suferirea de daune, sunet la moarte, sunet la moartea inamicilor, precum și o melodie simplă de fundal care se repetă.

Jocul este implementat în întregime folosind limbajul de programare C#. Animațiile sunt realizate folosind unealta Animator din cadrul Unity, iar pentru arta PixelArt am folosit tot site-ul itch.io [6], dar am desenat și singur câteva elemente de care am avut nevoie pe parcursul implementării (cum ar fi armurile).

**1.2 MOTIVAȚIA ALEGERII TEMEI**

Mereu am fost fascinat de aceasta industrie a jocurilor video. Înca de la primele jocuri jucate pe calculator (Super Mario, Captain Claw), m-am intrebat mereu cât de ușor/greu ar fi procesul de creare a unui astfel de joc. Cu trecerea timpului, jocurile au devenit mai complexe, cu grafică mult mai frumoasă și realistă (Call of Duty Modern Warfare, BattleField, League of Legends), ceea ce a făcut din jocurile pe calculator un mod placut de relaxare și petrecere a timpului liber.

Prefer jocurile pe calculator datorită confortului (pe telefon de obicei nu păstrez o postură corectă a corpului, ecranul prea aproape de ochi, etc), dar și datorită gamei mult mai largi de jocuri din care pot alege. În general prefer jocurile de aventura/FPS (First Person Shooter) istorice, sau strategie.

Până la momentul actual nu am avut multe tangențe cu industria de jocuri video, dar am decis să încerc să creez și eu un joc, deoarece este o oportunitate bună de folosire a timpului liber pentru a lucra la ceva folositor, iar dupa ceva experiența poate chiar să lansez mici jocuri pe platforme online (de exemplu Steam, Google Play, App Store) care pot genera venituri extra din reclame/ microtranzacții/ preț de cumpărare.

# ANALIZA DOMENIULUI

## 2.1 APLICAȚII SIMILARE EXISTENTE

La momentul actual, exista deja o multitudine de jocuri pe piață, atât de la producatori mici (indie developers), precum și de la companii mari, cu întindere globală, cum ar fi Ubisoft, Blizzard, Bungie, etc. Mai jos sunt prezentate câteva jocuri din aceeași categorie cu jocul creat de mine (RPG).

**World of Warcraft(WoW)**

World of Warcraft este un MMORPG (Massively Multiplayer Online Role Playing Game) creat de Blizzard Entertainment [7]. Este al 4-lea joc din serie, și cu toate că a fost lansat cu mai bine de 15 ani în urmă, continuă să domine scena jocurilor MMORPG datorită complexitații deosebite a experienței de joc, dar și a poveștii deosebit de amănunțită.

Acțiunea se petrece în interiorul unui univers fictiv numit Warcraft, în care jucătorul trebuie sa investească timp pentru a deveni mai puternic prin lupta cu inamicii din jur, cu alți jucători sau în evenimente speciale organizate regulat cu premii. Jocul dispune, de asemenea, și de sistem de interacțiune între jucători, prin funcția de „chat”.

Jocul oferă o atmosferă foarte competitivă jucătorilor, fiind considerat unul din cele mai profitabile jocuri lansate vreodată, desi este gratuit de descărcat și jucat, el oferă o multitudine de opțiuni pentru cei care doresc un progres mai rapid în joc, prin cumpărarea de abonamente lunare/bonusuri.

**Metin2**

Metin2 este un alt joc de tip MMORPG, dezvoltat de Ymir Entertainment in 2004 [8]. Spre deosebire de Warcraft, Metin2 este complet gratuit de jucat, deși avansarea în joc fără tranzacții este foarte înceată. A fost un joc foarte popular la vremea sa, căpătând o popularitate deosebita in România odata cu lansarea serverelor europene, mai ales în rândul adolescenților, și este încă jucat de mulți și astăzi.

Scopul jocului este ca jucătorul să își crească caracterul ales pentru a deveni mai puternic și a debloca artefacte rare, prin „farming” (lupta cu inamici mult mai slabi decât tine pentru a aduna resurse). Jocul permite și realizarea de comerț între jucători, folosind funcția de “chat” local.

**Albion**

Albion a fost creat de Sandbox Interactive în 2016. Acesta are o grafica modernă, în comparație cu Warcraft și Metin2. Este, de asemenea, și cross-platform (poate fi jucat atât de jucători de pe Windows/MacOS, cât și de pe platformele mobile), punând accent mare pe personalizarea caracterelor.

Jocul are acțiunea situată în Evul Mediu [9], unde jucătorul trebuie sa își crească nivelul caracterului creat prin diverse metode de interacțiune cu mediul înconjurător (Player vs Environment), iar după ce este considerat destul de puternic poate lua parte la lupte cu alți jucători (Player vs Player), în evenimente organizate, etc. Și acesta oferă posibilitatea de avansare mai rapidă prin tranzacții în interiorul jocului, dar fiind, de altfel, gratuit de jucat.

Tabel 1 conține o comparație între principalele caracteristici ale jocurilor menționate mai sus, precum și jocul creat de mine pentru proiectul de licență.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Caracterisitici** | **World of Warcraft** | **Metin2** | **Albion** | **Dungeon Eternal** |
| Multiplayer? | Da | Da | Da | Nu |
| Sistem de gestionare inventar | Da | Da | Da | Da |
| Obiecte ajutătoare | Da | Da | Da | Da |
| Poveste complexă | Da | Nu | Nu | Nu |
| Sistem de conversație | Da | Da | Da | Nu |
| Mai multe nivele | Nu | Nu | Nu | Da |
| Cross-platform | Nu | Nu | Da | Nu |

Tabel 1 : Tabel comparativ pentru jocurile prezentate

# TEHNOLOGII FOLOSITE

**3.1 SISTEMUL DE OPERARE ȘI PLATFORMA DE DEZVOLTARE ALESE**

Deoarece este primul meu joc mai complex creat până acum, am decis să folosesc un sistem de operare cu care sunt familiar, și anume Windows. Acest sistem de operare este, de asemenea, unul din cele mai comune platforme pentru jocuri, deși în ultimii ani platformele de jocuri dedicate cum ar fi PlayStation, XBOX sau Nintendo și jocurile pe telefonul mobil au devenit din ce în ce mai populare (Figura 1).

Platforma de dezvoltare aleasă este Unity, deoarece este o platformă des întalnită mai ales în rândul dezvoltatorilor individuali sau amatori. Deoarece este atât de răspândită, găsirea de materiale de învățare este ușoară, iar rezolvarea problemelor care pot apărea pe parcursul dezvoltării este simplă datorită numărului mare de utilizatori. Editorul de cod folosit este Visual Studio 2019 [10] , iar pentru crearea diagramelor am folosit editoare online precum app.diagrams.net [11] și app.creately.com [12].

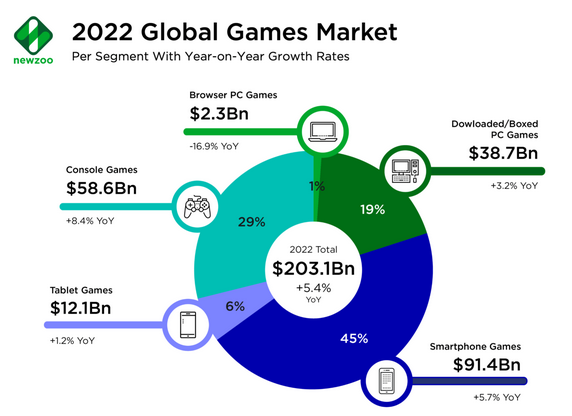


Figura 1: Distribuirea pieții jocurilor video pe diversele modalități de joc, imagine preluată din sondajul “PC Gaming to have less growth than Console and mobile in 2022” [13].

**3.1.1 AVANTAJELE ȘI DEZAVANTAJELE ACESTORA**

**Avantaje:**

* Calculatorul dispune de un număr foarte mare de utilizatori, în mod special cei care preferă Windows ca sistemul de operare principal;
* Exista opțiuni pentru orice buget, permițând o gamă largă de customizare pentru a avea dispozitivul perfect pentru fiecare utilizator în parte;
* Există o gamă largă de jocuri și dezvoltatori care creează jocuri pentru această platformă, cu prețuri accesibile;
* Este rentabil pe termen lung, deoarece performanțele sistemului pot fi îmbunătățite prin înlocuirea diferitelor componente mai slabe fără a fi nevoie de schimbarea întregii unități. Acest proces este unul relativ simplu, nu necesită unelte speciale sau asistența tehnică, putând fi realizat chiar de către utilizator;
* Exista o multitudine de resurse pentru învățare/rezolvare de probleme, având la dispoziție tutoriale, cursuri online gratuite/contra-cost, forumuri pentru întrebări, precum și o gamă variată de motoare de dezvoltare de jocuri, cum ar fi Unity, Unreal Engine, GameMaker, etc.

**Dezavantaje:**

* Deși a dominat scena jocurilor video ani de zile, calculatorul a început să fie depășit de către alte platforme, cum ar fi consolele sau telefonul mobil. Acest lucru duce la scăderea interesului manifestat de către dezvoltatorii de jocuri pentru această platformă, deoarece există altele mai profitabile pentru ei;
* Nu oferă consumatorilor aceeași satisfacție la utlilizare precum consolele. Acestea sunt unități specializate doar pentru rularea de jocuri, ceea ce oferă un sentiment de realizare/satisfacție, pe când calculatorul este folosit pentru un număr mare de activități, jocurile fiind doar unul dintre acestea. De asemenea, aceste platforme oferă titluri exclusive, ceea ce le dau un statut special, mai ales în rândul colecționarilor.
* Achiziționarea unui sistem capabil să ruleze ultimele apariții pe piață este mult mai scumpă în comparație cu o consolă sau un telefon mobil.

**3.2 PLATFORMA DE DEZVOLTARE UNITY**

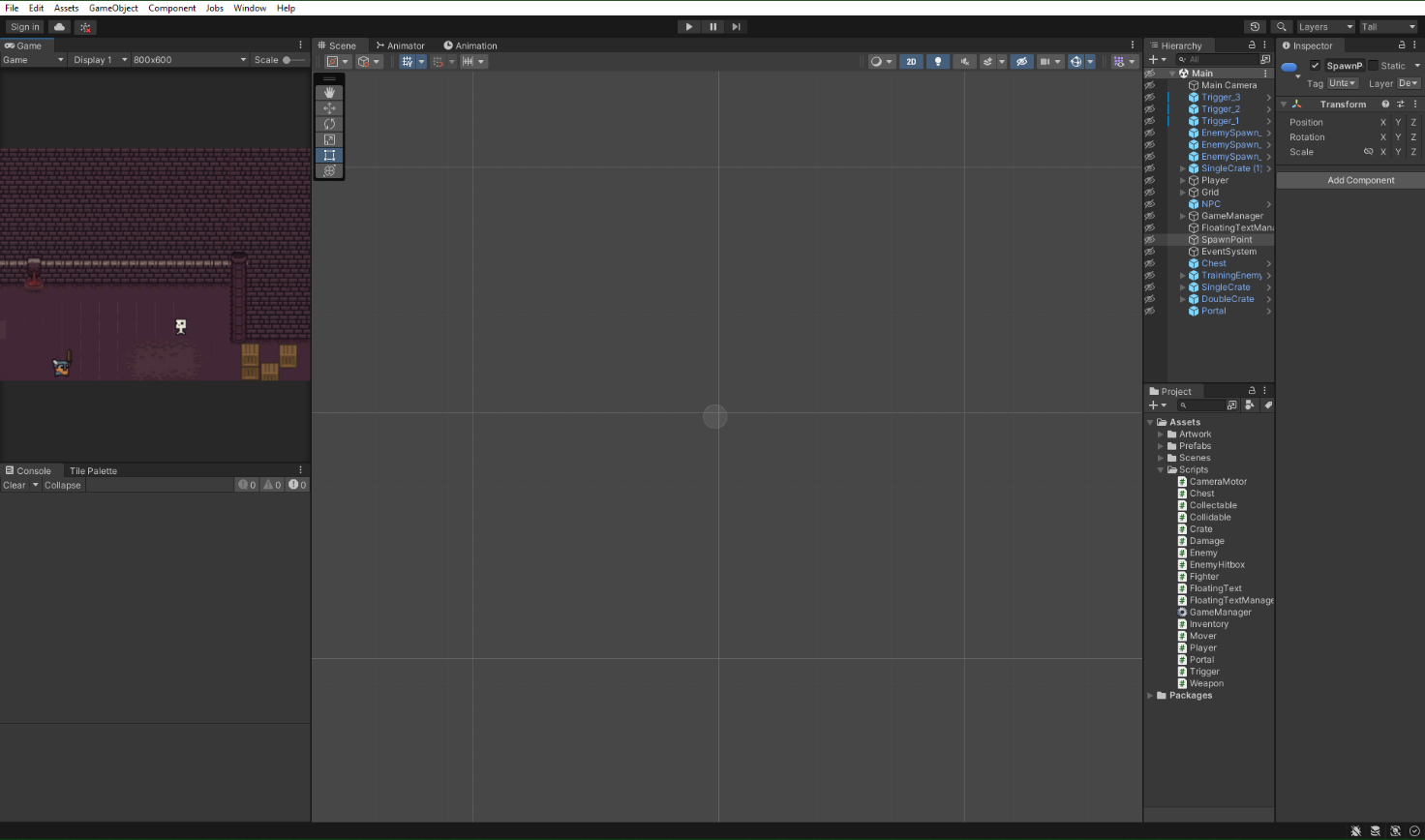
Este platforma de dezvoltare de jocuri video cea mai des întâlnită în rândul devoltatorilor amatori individuali/ echipe mici. A fost creată de către compania Unity Technologies în anul 2005, cu scopul de a crea un standard pentru dezvoltarea jocurilor video, făcând această ocupație mai accesibilă către mai multi dezvoltatori.

A fost creată inițial pentru MacOS X, dar a fost adăugat ulterior suport pentru Microsoft Windows, Android și Web Development. Motorul de dezvoltare poate fi folosit pentru crearea unei game largi de aplicații în două (2D) sau trei (3D) dimensiuni, atât pentru calculator, mobil, console, browser cât și pentru platformele de realitate virtuală mai nou.

Dezvoltatorii se pot folosi de resursele gratuite existente în Unity, sau pot importa/cumpăra grafică sau modele 3D. Unity oferă modalități de interacționare cu camera, cu caracterele din joc și implementarea mișcarii caracterelor. Pentru jocuri 2D, Unity oferă opțiuni de importare de “sprite”-uri (imagini care conțin diverse elemente folosite în crearea graficii jocului), unelte de construire a lumii jocului folosind o paletă personalizabilă de elemente grafice și crearea de animații simple. Pentru jocuri 3D, oferă funcții precum compresie de texturi, mipmaps(piramide de imagini cu rezolutia din ce în ce mai scăzută, pentru a ajuta la randare si reducerea efectului de aliasing), prelucrarea reflexiilor, formelor și texturilor, a luminilor și umbrelor [4].

Inițial, Unity a fost comercializat ca produs final, dar în 2016 compania a adoptat modalitatea de abonare ca metodă de comercializare, atât gratuită cât și plătită. Licența gratuită este orientată spre dezvoltatorii individuali sau echipele mici, care nu generează venituri mai mari de $200.000 dolari/an din jocurile create folosind Unity. Pentru utilizatori cu venituri mai mari de $200.000 dolari/an, este obligatorie licența plătită. De asemenea, este impusă licența platită pentru cei care dezvoltă jocuri pentru console, deși o parte din producătorii de console oferă licența de Unity prin pachetul SDK prin înscrierea în programul Preffered Platform Licence Key [4].

Interfața aplicației este împărțită în mai multe ferestre, care sunt folosite pentru a naviga între diferitele obiecte care se află în scena curentă, ofera detalii despre obiectul selectat, un exporator de fisiere pentru directoarele proiectului, consola de executare și perspectiva jocului. Mai jos este atașată o captură cu interfața Unity, versiunea 2021.3.6f1.

Figura 2: Interfața Unity, versiunea 2021.3.6f1

În captura de mai sus se observă în stânga sus perspectiva jocului, iar în stânga jos consola. Pe fereastra principală este perspectiva scenei curente, cu toate obiectele pe care aceasta le conține. În dreapta putem observa Ierarhia scenei curente, cu obiectele conținute. Sub Ierarhie se află Exploratorul proiectului, unde se poate naviga în diversele directoare ale proiectului (cum ar fi cel de script-uri, arta, prefabricate, scene). Ultima coloană din dreapta este Inspectorul, care conține toate proprietățile unui obiect selectat, permițând atât vizionarea cât și modificarea acestora. În Inspector se pot vedea variabilele publice pe care un obiect le are prin adaugarea de script-uri, și permite modificarea valorilor acestora, aceasta funcționalitate fiind foarte utilă mai ales la inițializarea variabilelor de tip GameObject/clase particulare.

**3.2.1 UTILIZAREA TEHNOLOGIILOR SPECIFICE UNITY ÎN CADRUL PROIECTULUI**

Unity oferă o gamă largă de metode pre-implementate, gata de a fi utilizate de către dezvoltatori, cum ar fi metode pentru detectarea coliziunilor sau detectarea de input-uri din partea utilizatorului în timpul rulării jocului.

* **Metoda Collider2D.OverlapCollider()**

Primește ca parametri un obiect de tip ContactFilter2D care reprezintă setul de parametri pentru filtrarea obiectelor cu care este detectat contactul, și un vector de obiecte tip Collider2D în care vor fi stocate obiectele cu care obiectul de pe care este apelată această metodă este în coliziune [14].

* **Metoda Physics2D.BoxCast()**

Primește ca parametri obiecte de tip Vector2 pentru origine, dimensiune, direcție, float pentru unghi și distanță și un întreg pentru layerMask (în cazul meu, am folosit metoda GetMask care primește ca parametri șiruri de caractere care reprezintă layer-ele cu care se dorește detectarea coliziunii) [15]. Metoda incearcă să creeze o cutie cu proprietățile și locația specificate prin parametri. Dacă cutia este creată cu succes, atunci returnează null, însemnând că este posibilă mutarea caracterului la locația dorită.

* **Metoda Vector3.Distance()**

Primește ca parametri 2 obiecte de tip Vector3 și returnează distanța dintre ele [16].

* **Metoda GameObject.SendMessage()**

Primește ca parametri un șir de caractere cu numele metodei pe care dorim să o apelăm, și un obiect care reprezintă mesajul pe care dorim să îl transmitem. La apelare, această metodă apelează metoda primită ca parametru cu mesajul dorit pe obiectul de pe care este apelată [17].

* **Metoda Input.GetKeyDown()**

Primește ca parametru un enum care reprezintă tasta pe care dorim să o supraveghem pentru a detecta apăsările. Returnează true la cadrul în care este detectat inceputul apăsării tastei primită ca parametru [18].

* **Metoda Transform.Translate()**

Primește ca parametri 3 float-uri care reprezintă destinația la care se dorește mutarea obiectului caruia îi este atașată componenta Transform care apelează metoda [19].

* **Metoda PlayerPrefs.SetString()**

Primește ca parametri 2 string-uri care reprezintă numele unei chei și valoarea corespunzătoare acesteia [20].

* **Metoda PlayerPrefs.HasKey()**

Primește ca parametru un string care reprezintă numele cheii cautate, și returnează true dacă aceasta există, sau false în cazul contrariu [21].

* **Metoda PlayerPrefs.GetString ()**

Primește ca parametru un string care reprezintă numele cheii cautate, și returnează un șir de caractere cu valoarea atribuită cheii date [22].

# ARHITECTURA JOCULUI

**4.1 ARHITECTURA SISTEMULUI**

Folosind platforma Unity se tratează toate interacțiunile dintre obiectele din joc, efectele vizuale, etc. Unity oferă pentru aceasta sisteme de gestionare a fișierelor audio, precum și unealta Animator care se poate folosi pentru a crea animații fie folosind tehnica de stop-cadru, cât și generarea automată de cadre intermediare dintre cel inițial si cel final ales la creare.

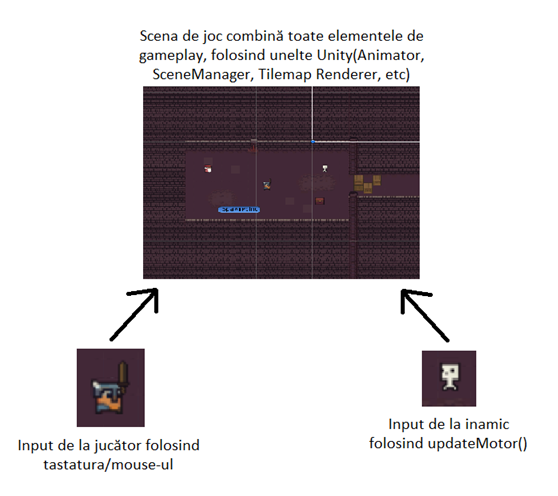


Figura 3. Arhitectura sistemului

Jucătorul controlează caracterul său folosind comenzi de la tastatură(WASD/săgeți pentru mișcare, SPACE pentru atac cu arma), și mouse-ul pentru a naviga meniurile.

Inamicii sunt obiecte statice cât timp caracterul jucătorului nu este în raza lor de acțiune prestabilită. Când acesta se apropie destul de mult de un inamic, acesta din urmă va începe să caute jucătorul și îl urmarește cu scopul de a ataca cât timp nu s-a îndepărtat înca destul de mult de poziția sa inițială. Când detectează că este prea departe, se va intoarce la poziția inițială și ciclul de așteptare se reia de la capăt.

**4.2 STRUCTURA SCENELOR IMPLEMENTATE ÎN INTERIORUL JOCULUI**

Structura de bază a jocului este formată din mai multe scene, prin care utilizatorul poate naviga fie prin apăsarea de butoane (din meniul principal, spre exemplu), cât și prin interacțiunea cu obiectele înconjurătoare cu proprietăți speciale (portalul, spre exemplu, un obiect care, la coliziunea cu jucătorul va declanșa încărcarea unei noi scene alese).

O scenă reprezintă o colecție de obiecte special aranjate și inițializate de către dezvoltatorul jocului pentru a crea mediul de joc dorit (care poate fi un nou nivel, un meniu de start/final, un meniu de încărcare sau pauză, etc).

Principalele obiecte pe care le conține o scenă (în cazul unui joc 2D, cum este Dungeon Eternal) sunt :

* **MainCamera.** Este un obiect de tip Cameră, care reprezintă ceea ce jucătorul va vedea pe ecran în momentul rulării jocului. Acest obiect poate conține un script pentru implementarea funcției de urmărire a jucătorului în timp real.
* **Grid + Tilemap.** Grid este un obiect care împarte scena în blocuri de dimensiuni egale, pentru o mai ușoară creare a mediului de joc. Acest obiect conține în interiorul lui obiecte de tip Tilemap, care sunt “straturi” de decor, care împreună construiesc fundalul jocului. Acestea sunt suprapuse unul peste altul(spre exemplu, o scenă în jocul în cauză conține în general 4 straturi : podea, strat intermediar între podea și design, design, și un strat mai special de coliziune). Stratul de coliziune este un strat invizibil care conține limitele în care jucătorului îi este permis să meargă.

Jocul Dungeon Eternal este alcătuit din 6 scene, organizate astfel :

1. Scena inițială numită StartMenu;
2. Scena corespunzătoare primului nivel din joc, numită MainDungeon;
3. Scena cu lupta finală pentru primul nivel, numită BossDungeon. Aceasta face legătura între nivelul 1 si nivelul 2 din joc, dacă jucătorul a fost victorios în lupta cu inamicul final;
4. Scena corespunătoare celui de-al doilea nivel din joc, numită SecondDungeon;

* Scenele 2 si 4 conțin la rândul lor portale către scenele următoare :

1. Scena nivelului secret cu resurse, numită TreasureDungeon;
2. Scena nivelului de luptă infinită cu inamici pentru adunarea de resurse, numită EternalDungeon.

**Scena 1 : Meniul de start**

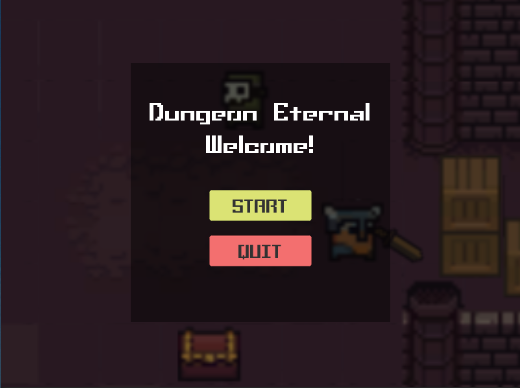


Figura 4. Captură de ecran cu prima scenă din joc

Este scena care se încarcă la deschiderea jocului. La rularea fișierului

DungeonEternal.exe, se va deschide meniul principal al jocului, care oferă utilizatorului 2 opțiuni, și anume începerea jocului sau închiderea acestuia. Scena este alcătuită din cameră, un fundal, și un obiect de tip Canvas care conține un fundal mai întunecat, un bloc text cu titlul jocului și cele două butoane (Figura 4).

**Scena 2 : Primul nivel de joc**

****

Figura 5. Captură de ecran la încărcarea primului nivel de joc

La apăsarea butonului de Start din scena anterioară, această scenă (Figura 5)

este încarcată pe ecranul jocului. Scenele corespunzătoare nivelelor principale sunt cele mai complexe din joc, conținând o multitudine de obiecte care reprezintă cufere, inamici, caractere de ajutor (NPC-uri), fântâni de vindecare, declanșatoare (trigger), portale, precum și elemente de interfață cum ar fi butoanele de meniuri (pauză, crafting și utilizare de poțiune de vindecare), bara de viață și banii deținuți la momentul respectiv.

Aceste elemente de interfață se găsesc într-un obiect de tip UI Canvas, care este vizibil mereu în timpul jocului, situat în partea de jos a ecranului, și cu ajutorul căruia jucătorul poate naviga jocul mai ușor. Butoanele din colțul stânga jos declanșează apariția meniurilor de pauză, respectiv crafting. Bara de viață permite vizualizarea nivelului de viață al jucătorului actualizat în timp real, printr-o imagine care își schimbă dimensiunea, și un bloc de text care arată numeric viața. În colțul dreapta jos se găsește un bloc de text care arată cantitatea de bani pe care jucătorul o deține în inventar la momentul respectiv, iar deasupra acestui text se află un buton care, la apăsare, consumă o poțiune de viață pentru a crește nivelul de viață.

Meniurile de pauză și craftare sunt construite dintr-un UI Canvas care conține ca fundal un buton de dimensiunea ecranului ce va permite revenirea la joc prin click oriunde în afara meniului. Acestea apar pe ecran folosind o animație simplă de mișcare verticală, care este din nou utilizată pentru închiderea meniurilor la click în afara acestora.

Meniul de pauză (Figura 6) este împărțit în patru segmente. În stânga este încărcat inventarul jucătorului, care conține armele, armurile și poțiunile. În mijloc sunt opțiuni de customizare a caracterului și a tipului de armă utilizat. În dreapta se află informațiile caracterului precum și un buton de resetare a progresului. Sub acestea se găsește bara de experiență, care arată vizual nivelul de experiență printr-o imagine care își schimbă dimensiunea și un bloc de text peste aceasta.



Figura 6. Captură de ecran cu meniul de pauză prin folosirea butonului reprezentat de Cufăr

Meniul de crafting (Figura 7) este alcătuit din trei segmente. În stânga se află din nou inventarul, iar în dreapta se află opțiunile de cumpărare folosind butoane de navigare și cumpărare, precum și costurile pentru fiecare tip de armă/ armură/ poțiune. Sub acestea se află un loc destinat mesajelor de informare asupra acțiunilor solicitate (care poate fi de succes sau lipsa unei resurse necesare).



Figura 7. Captură de ecran la declanșarea meniului de Crafting prin

apăsarea butonului reprezentat de Ciocan

Când jucătorul moare, este declanșat un meniu special (Figura 8) în care sunt prezentate în stânga opțiunile de reîncercare, revenirea la meniul principal și închiderea jocului. În partea dreaptă sunt afișate cateva statistici ale progresului, Acest meniu nu mai poate fi închis prin click în afara meniului, ci doar prin folosirea butoanelor puse la dispoziție.



Figura 8. Captură de ecran la momentul morții jucătorului, cu meniul corespunzător

**4.3 ARHITECTURA SOFTWARE**

Deoarece principalele obiecte din joc (caracterul jucătorului, inamicii, cutiile, portalele, etc) au unele elemente comune, precum coliziunea, viața, daune provocate sau mișcarea, am folosit o serie de clase derivate pentru a evita duplicarea de cod. Structura acestor clase este prezentată în diagrama din Figura 9.

Am definit patru clase principale, Collidable, Collectable, Fighter și Mover, din care sunt apoi derivate toate celelalte clase folosite pentru crearea de obiecte.

**Collidable** va fi moștenită de toate clasele care au coliziunea ca interacțiune principală (pentru declanșarea de acțiuni, afișarea mesajelor, etc). Este moștenită de catre clasele Portal, Trigger, EternalTrigger, Weapon, NPC, HealingFountain și Collectable.

**Collectable** moșteneste clasa Collidable, dar în plus față de aceasta are și atribute/ metode pentru colectarea de resurse (cum ar fi bani, armuri, arme). La rândul ei, Collectable este moștenită de clasa Chest, care este folosită de toate cuferele din joc.

**Fighter** este o clasă care definește obiecte ce pot participa în luptă, adică pot suferi daune, și pot muri dacă viața lor a ajuns la 0. Este moștenită de clasele Mover și Crate.

**Mover** definește obiectele care se pot mișca în joc. Aceasta moștenește clasa Fighter, deoarece doar aceste obiecte se pot mișca în joc. Este moștenită la rândul ei de catre clasele Player și Enemy.

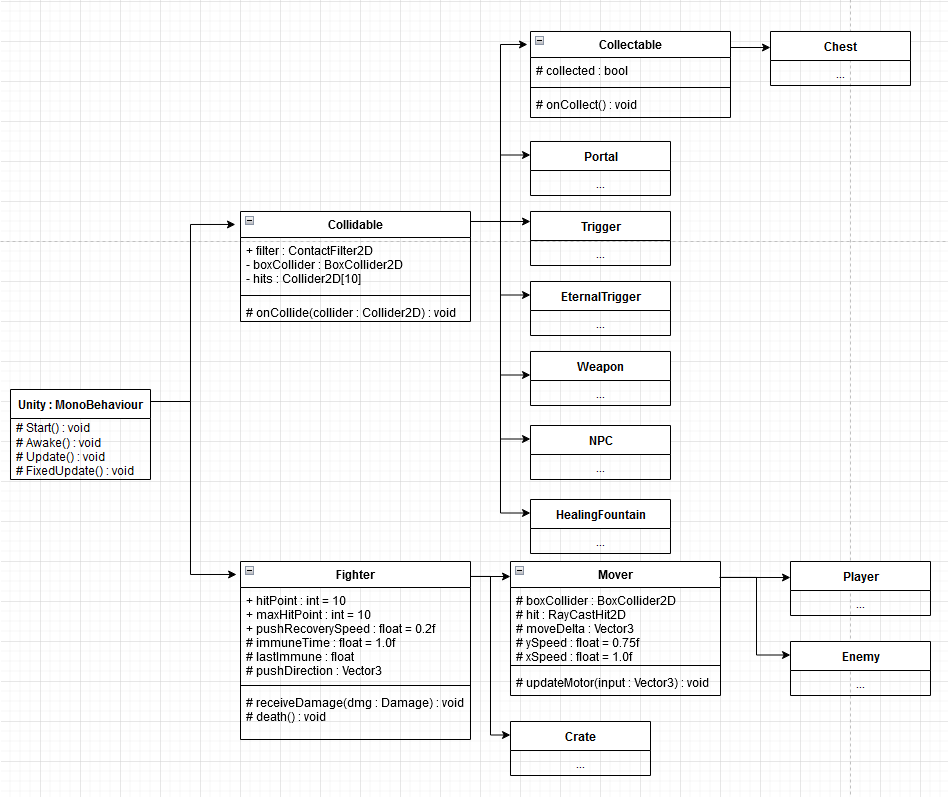


Figura 9. Diagrama claselor principalelor obiecte

**4.3.1 PREZENTAREA MODULELOR SOFTWARE**

În total, 29 de clase diferite sunt folosite pentru o implementare cât mai optimă a diferitelor interacțiuni ce apar în timpul jocului. În acest capitol sunt prezentate modulele software folosite în dezvoltarea jocului, folosind diagrame UML și explicații pentru atributele/metodele care le alcătuiesc.

* **Unity MonoBehaviour** [23].

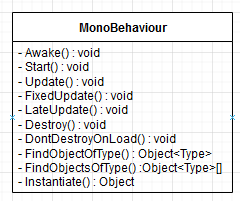


Figura 10. Diagrama UML pentru clasa MonoBehaviour

Această clasă este specifică Unity, și conține metode cu comportament special. Este

moștenită de mai multe clase din proiect, de aceea am decis să o prezint prima pentru a evita repetarea exagerată a diagramei.

Awake() este prima metodă apelată la rularea scriptului, înaintea tuturor celorlalte,

urmată de Start(). Update() este o metodă care se apelează la inceputul fiecărui cadru, LateUpdate() este apelată la finalul fiecărui cadru, iar FixedUpdate() se apelează la un interval fixat de timp, configurabil din setările proiectului. Această metodă este recomandată pentru implementarea funcționalităților de bază, cum ar fi mișcarea, pentru a nu exista diferențe de timp dacă jocul este rulat de pe un sistem mai puternic/slab.

Destroy() distruge obiectul primit ca parametru. DontDestroyOnLoad() face obiectul primit ca parametru să persiste. FindObjectOfType/FindObjectsOfType sunt metode folosite pentru a găsi obiecte din joc, iar Instantiate() este o metodă care clonează obiectul primit ca parametru și returnează clona lui.

1. **CameraMotor, extinde MonoBehaviour**

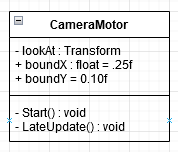
****

Figura 11. Diagrama UML pentru clasa CameraMotor

Este script-ul atașat obiectului de tip Cameră din scenele de joc (meniul principal are o cameră statică, de exemplu, fără nevoia unui script de mișcare). Acesta conține 3 atribute folosite pentru a determina comportamentul camerei pentru a asigura un câmp vizual optim pe parcursul rulării jocului. Variabila lookAt reprezintă pozitia obiectului pe care camera îl urmărește (în cazul de față fiind Player), iar cele doua atribute boundX, respectiv boundY sunt toleranțele acceptate până camera începe efectiv să se miște (daca s-a înregistrat doar o mișcare scurtă camera nu se va mișca).

Metoda Start(), fiind o metodă specifică Unity din clasa MonoBehaviour, se apelează la pornirea scenei curente, fiind folosită pentru inițializarea variabilei lookAt.

Metoda LateUpdate() este tot o metodă specifică Unity din clasa MonoBehaviour, și este apelată la finalul fiecărui cadru. Am ales finalul cadrului deoarece camera urmărește jucătorul, și trebuie să își actualizeze poziția doar după ce jucătorul a introdus comenzile sale de mișcare pentru scena respectivă (ceea ce se întamplă la începutul fiecărui cadru).

Pentru mișcare, se verifică dacă poziția camerei relativ de cea a Player-ului se găsește între limitele de inactivitate, iar dacă una din limite este depășită, camera se va deplasa în direcția respectivă pentru a-și corecta poziția.

1. **Damage**

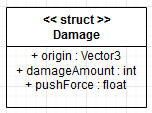
****

Figura 12. Diagrama UML pentru structura Damage

Este o simplă structură care reprezintă daunele provocate de un obiect printr-un atac. Conține 3 câmpuri folosite pentru a determina autorul atacului, puterea atacului și împingerea înapoi. Daunele sunt transmise către obiectul atacat folosind metoda Unity sendMessage() [17], unde un obiect de tip Damage este transmis ca parametru.

1. **DontDestroy, extinde MonoBehaviour**

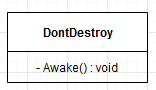
****

Figura 13. Diagrama UML pentru clasa DontDestroy

Am ales să creez o clasă separată pentru a fi mai ușor de modificat obiectele prin adăugarea/ștergerea acestui script din interfața Unity decât editarea fiecărui script în parte.

Conține o singură instrucțiune, aflată în interiorul metodei Awake(), care se apelează chiar și înaintea metodei Start(), și face obiectul căruia îi este atașat scriptul rezistent la distrugere când se încarcă o scenă. Principalele obiecte care trebuie sa nu fie distruse sunt : Player, HUD, FloatingTextManager, SoundManager și GameManager.

1. **FloatingTextManager, extinde MonoBehaviour**

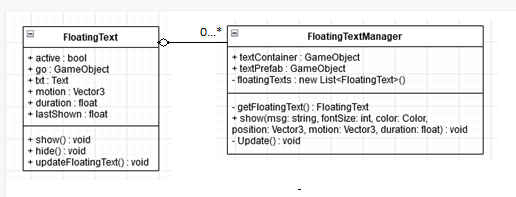
****

Figura 14. Diagrama UML pentru clasele FloatingText și FloatingTextManager

Este clasa care se ocupă de afișarea/ascunderea mesajelor în joc. Conține un câmp textContainer care este Canvas-ul pe care se afișează mesajele, un câmp textPrefab care este legat la prefabricatul de tip FloatingText pentru a accesa mai ușor proprietățile mesajelor, și o listă de obiecte de tip FloatingText care conține toate mesajele care au fost/sunt afișate.

Metoda getFloatingText() caută în lista de obiecte unul care este inactiv (poate fi reutilizat), și îl returnează. Dacă lista este goală, se creează un obiect nou folosind prefabricatul și îl returnează.

Metoda show() atribuie proprietăți obiectului returnat de getFloatingText(), cum ar fi mesajul care trebuie afișat, culoarea, dimensiunea font-ului, etc.

Metoda Update() va apela updateFloatingText() din cadrul clasei FloatingText pentru fiecare obiect din lista de mesaje.

1. **FloatingText**

Acestă clasă conține proprietățile unui mesaj, folosind atributele active, go, txt,

motion, duration, lastShown. Acestea sunt folosite pentru a determina dacă obiectul trebuie să fie afișat, păstreză evidența timpului pentru a ști când mesajul trebuie ascuns, etc.

Metoda show() este apelată la crearea mesajului. Va salva timpul la apelare, setează contorul active, și activează obiectul pentru a fi vizibil pe ecran.

Metoda hide() se apelează când a expirat durata de timp în care mesajul trebuia să fie vizibil, și dezactivează obiectul.

Metoda updateFloatingText() este folosită pentru a decide dacă este momentul apelării metodei hide() prin compararea timpului curent cu cel la care a fost creat mesajul și durata de activitate a acestuia. Dacă mesajul încă este valabil, atunci se actualizează poziția acestuia pe ecran pentru a adăuga mișcarea dorită(textul să plutească ușor în sus, lateral, etc). Diagrama acestei clase este prezentată în Figura 14.

1. **Inventory, extinde MonoBehaviour**

Conține doar câmpuri cu valori întregi ce reprezintă inventarul jucătorului. Sunt folosite 10 variabile int pentru a ține evidența progresului caracterului, prin bani, armuri, arme și poțiuni de vindecare. Este instanțiată în interiorul clase GameManager. Diagrama acestei clase este vizibilă în Figura 15 de mai jos.

1. **GameManager, extinde MonoBehaviour**

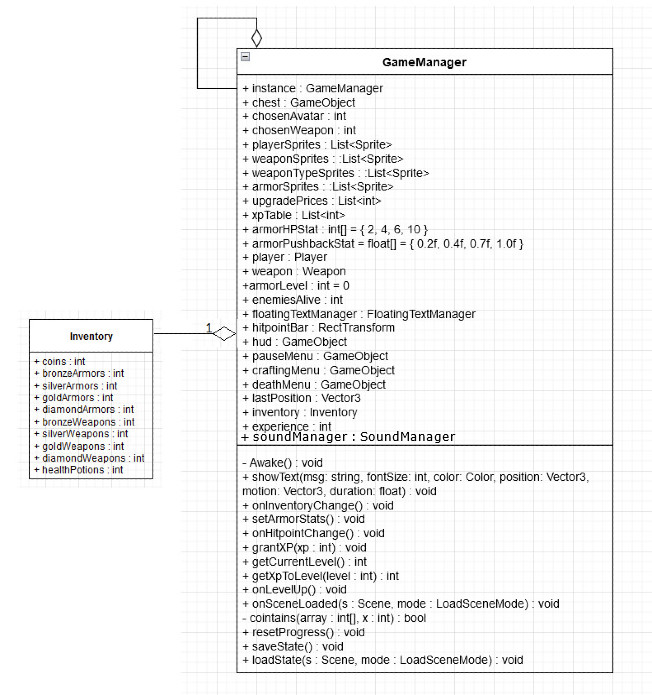


Figura 15. Diagrama UML pentru clasele Inventory și GameManager

Una dintre cele mai importante clase din joc. Această clasă este legată de un obiect gol în scena primului nivel, care apoi este păstrat pe tot parcursul jocului, indiferent de scena curent, pentru a se monitoriza și salva progresul făcut.

Conține o instanța proprie public, pentru a putea avea un obiect GameManager care poate fi apelat de oriunde din joc, precum și diverse variabile care pastrează legaturile cu player-ul, arma acestuia (weapon), inventarul (inventory), experiența (experience), ultima pozitie a caracterului la părăsirea scenei (lastPosition), folosită pentru a ști unde să fie plasat caracterul la revenire, nivelul armurii (armorLevel). De asemenea, conține și variabile de referințe spre diferite obiecte din scene, cum ar fi managerul de mesaje (floatingTextManager), managerul de sunete(soundManager), interfața de joc(hud, hitpointBar), meniurile care sunt folosite (pauseMenu, craftingMenu, deathMenu), numarul curent de inamici in viață (enemiesAlive), precum și variabile folosite pentru setările/dezvoltarea jocului (imaginile cu armele, armurile și avatar-urile folosite, proprietățile acestora, costurile de îmbunătățire/creștere în nivel).

În interiorul metodei Awake() se inițializează anumite variabile, și obiectele care sunt persistente (au atribuit scriptul DontDestroy) sunt făcute de tip Singleton, adică poate exista o singură instanță a acelui obiect (GameManager, player, floatingTextManager, HUD), pentru a combate crearea acestora din nou la încărcarea unei noi scene, de exemplu.

Exista diferite metode folosite pentru a implementa funcționalități din joc, cum ar fi show (folosită pentru a afișa mesaje), setArmorStats (folosită pentru a actualiza rezistențele jucătorului în funcție de armura curentă), metode de gestionare a experienței: grantXP(), getCurrentLevel(), getXpToLevel(), precum și metode care se apelează la anumite schimbări din joc: onInventoryChange(), onLevelUp(), onSceneLoaded(). Aceasta din urmă este apelată la încărcarea fiecărei noi scene, și, permite inițializarea corectă a scenelor (actualizarea pozitiei jucătorului la început, crearea/ascunderea de obiecte cum ar fi cufere, portale, NPC-uri sau inamici).

În final, există și metode de gestionare a progresului făcut, prin salvarea (saveState), încarcarea (loadState) sau resetarea progresului (resetProgress). Aceste lucruri sunt realizate folosind metodele SetString(), respectiv HasKey() din cadrul clasey PlayerPrefs din Unity.

1. **EternalController, extinde MonoBehaviour**

Similar cu GameManager, și acest script, corespunzător scenei EternalDungeon, este conectat la un obiect gol, cu rolul de a monitoriza comportamentul acestei scene când este rulată. Conține o instanță a sa, pentru a putea apela metodele din clasă. Este folosit, în principal, pentru controla valurile de inamici care sunt creați din 15 în 15 secunde, care devin din ce în ce mai grele pe masură ce timpul trece. Conține atribute pentru tipurile de inamici care pot fi creați, dimensiunile fiecărui tip în parte, precum și numarul de inamici de fiecare tip pentru valul curent.

Metoda Start() este folosită pentru a face instanța proprie un Singleton, pentru a detecta locația de apariție a inamicilor, găsirea componentelor și inițializarea primului val de inamici. Metoda Update() verifică dacă a trecut suficient timp de la ultimul val și este momentul unui nou val, mai dificil.

Metoda printMsgAndSpawn() este apelată pentru a afișa mesajele de avertizare că un nou val de inamici se apropie, la cate o secundă distanță, iar după mesaje se apelează spawnEnemies() care este efectiv metoda de generare a inamicilor, după setările primite ca parametri. Pentru a găsi poziția de generare se apelează metoda getRandomPosition(), care va returna o variabila Vector3 aleatoare, aflată în vecinătatea poziției inițiale.

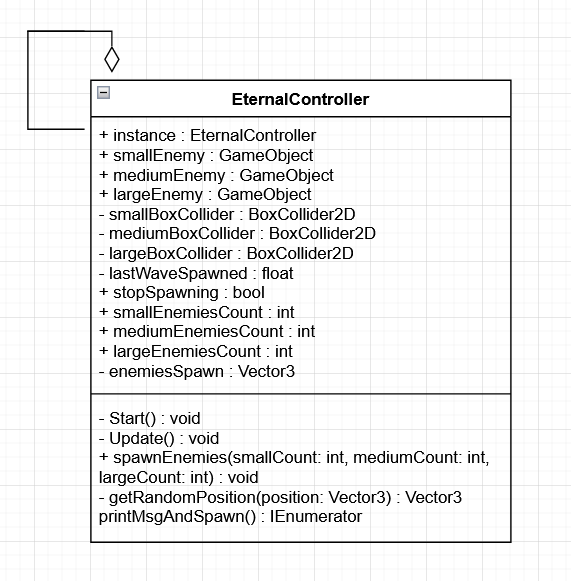
****

Figura 16. Diagrama UML pentru clasa EternalController

1. **Collidable, extinde MonoBehaviour**

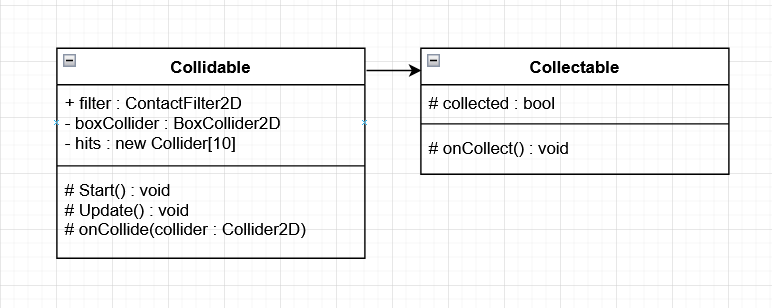
****

Figura 17. Diagramele UML pentru clasele Collidable și Collectable

Clasa Collidable este una din clasele principale din joc, care stă la baza multor alte clase și obiecte. Adaugă funcționalitatea coliziunii obiectelor cărora le este atașat acest script/claselor care o extind. Pentru aceasta, conține atribute pentru BoxCollider-ul pentru care se testează coliziunea, un filtru de tip ContactFilter2D pentru sortarea obiectelor cu care se interacționează, și un tabel unidimensional de tip Collider pentru stocarea obiectelor cu care există coliziuni la momentul verificării.

Metodele cele mai importante din această clasă sunt Update() și onCollide(). Update() este moștenită si suprascrisă din MonoBehaviour, și pentru fiecare cadru se verifică dacă există coliziuni cu alte obiecte, și, dacă există, se apelează metoda onCollide() pentru acestea, pe rând. Metoda onCollide() este metoda de tratare a coliziunii, și este suprascrisă de fiecare clasă care o moștenește pentru a-i personaliza comportamenul.

1. **Collectable, extinde Collidable**

Reprezintă un caz particular de obiecte cu coliziune, care, în plus, oferă funcția de

colectare. Pentru aceasta, există un câmp logic numit collected, care va monitoriza dacă obiectul respectiv a fost colectat sau nu. Metoda onCollide() este suprascrisă pentru a verifica dacă jucătorul este cel care interacționează cu obiectul, caz în care este apelată metoda onCollect, ce va seta variabila collected pe true, marcând obiectul pentru a ști că a fost colectat.

1. **Chest, extinde Collectable**

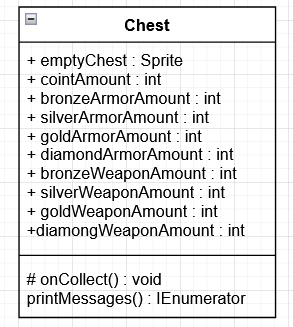
****

Figura 18. Diagrama UML pentru clasa Chest

Este un obiect colectabil, care are rolul de a oferi resurse (bani, arme, armuri) jucătorului când interacționează cu el. Pentru a personaliza conținutul cufărului, există 9 câmpuri de tip intreg pentru resurse, iar pentru a schimba imaginea obiectului la momentul colectării există un atribut de tip Sprite cu imaginea unui cufăr gol.

Metoda onCollect este suprascrisă, având în plus față de cea de bază schimbarea imaginii cufărului gol și afișarea mesajelor de informare asupra bunurilor primite. Pentru aceasta se apelează metoda printMessages(), care afișeasă mesaje despre resursele primite, unul după altul pentru a evita suprapunerea/aglomerarea textelor.

1. **Portal, extinde Collidable**

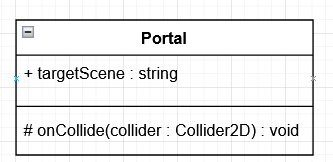
****

Figura 19. Diagrama UML pentru clasa Portal

Clasa aceasta este folosită pentru încărcarea/teleportarea jucătorului spre o nouă scenă în momentul coliziunii cu obiectul. Pentru aceasta există un câmp de tip șir de caractere care reprezintă numele scenei care trebuie încărcată.

Metoda onCollide() este suprascrisă pentru a verifica coliziunea să se realizeze cu jucătorul, se actualizează ultima poziție a jucătorului inainte de intrarea în portal, se salvează progresul și se încarcă scena dorită.

1. **Trigger, extinde Collide**

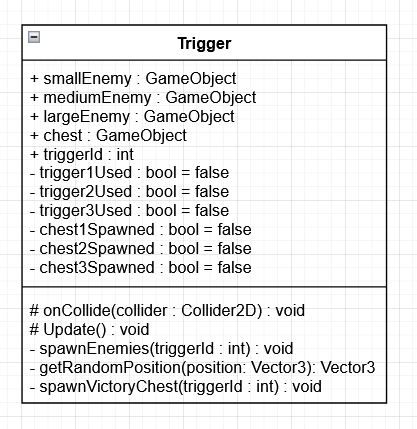
****

Figura 20. Diagrama UML pentru clasa Trigger

Acest script oferă obiectului caruia îi este atașat (un obiect invizibil) comportamentul de declanșator de acțiuni. Pentru primul nivel din joc sunt folosite obiecte de acest tip la intrarea în camerele de luptă, care, la coliziune, declanșează apariția inamicilor într-o locație aleasă. După lupta, când toți inamicii sunt înfrânți apare un cufăr cu resurse.

Pentru aceasta, clasa conține referințe spre prefabricatele inamicilor și cufărului, folosite pentru personalizarea nivelului. Fiecare obiect de acest tip conține un id după care este identificat, pentru a ști comportamentul care trebuie declanșat, și cateva variabile logice care țin evidența obiectelor consumate/cuferelor create, pentru a evita bucle infinite de activare a unui trigger.

Metoda onCollide() verifică dacă interacțiunea se face cu jucătorul, iar, dacă declanșatorul curent nu a fost folosit încă acesta este activat prin apelarea metodei spawnEnemies(), care, în funcție de declanșatorul activat va crea un număr prestabilit de inamici de dificultăți diferite în jurul unei locații predefinite.

Metoda Update() verifică dacă toți inamicii sunt înfrânți, caz în care este momentul de aparție a cufărului cu recompense, care se face prin metoda spawnVictoryChest(). Aceasta verifică id-ul primit ca parametru pentru a ști care cufăr trebuie activat.

1. **EternalTrigger, extinde Collidable**

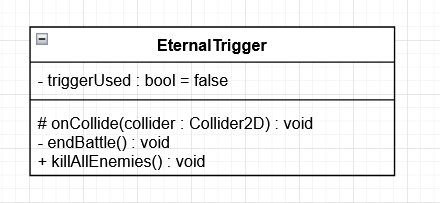
****

Figura 21. Diagrama UML pentru clasa EternalTrigger

Un alt tip de trigger, corespunzător nivelului EternalDungeon. Acesta este plasat în fața portalului de revenire la scena principală, și are rolul de a încheia bătălia și activarea cufărului de resurse. Pentru aceasta există un câmp logic numit triggerUsed, care are rolul de a face obiectul inactiv după activare, pentru a evita formarea de bucle infinite.

Metoda onCollide() este suprascrisă pentru a verifica coliziunea cu jucătorul, iar, dacă obiectul nu a fost încă folosit se apelează metodele endBattle() și killAllEnemies(), după care triggerUsed este marcat true. Metoda endBattle() este folosită pentru a determina numărul de inamici înfrânți, care va determina ulterior cât de bogat este cufărul cu resurse creat. Metoda killAllEnemies() caută toate obiectele de tip inamic și le distruge, actualizând totodată contorul enemiesAlive din GameManager.

1. **NPC, extinde Collidable**

Această clasă este folosită pentru a oferi comportament caracterelor din joc (NPC).

Metoda Update() are rolul de a detecta când jucătorul s-a apropiat destul de obiect, caz în care este afișat un nume deasupra obiectului, care semnalizează că există o interacțiune specială pregătită în momentul coliziunii. Acest nume este afișat sub forma unui mesaj te tip FloatingText, actualizat cat de repede posibil pentru a fi mereu deasupra obiectului. Metoda onCollide() declanșează afișarea de mesaje folosind rutina printMessages. Aceste mesaje sunt transmise ca parametri folosind un vector de string-uri care este inițializat diferit pentru fiecare NPC în parte.

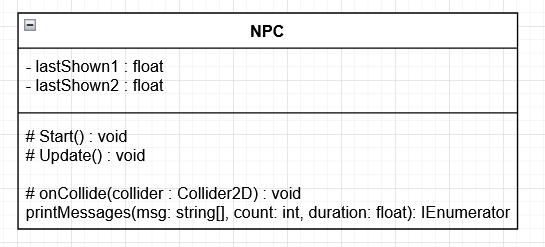
****

Figura 22. Diagrama UML pentru clasa NPC

1. **HealingFountain, extinde Collidable**

Oferă funcționalitatea de vindecare obiectului caruia îi este atașat acest script (un obiect invizibil plasat deasupra fântânii din fundal). Proprietățile acestui obiect sunt puterea de vindecare (healingAmount), frecvența de repetiție (healCooldown) și o variablia float (lastHeal) care păstrează evidența timpului.

La coliziune cu jucătorul, dacă timpul de la ultima vindecare este mai mare decât frecvența de repetiție viața acestuia este incrementată, afișându-se totodată și un mesaj pe ecran pentru informare.

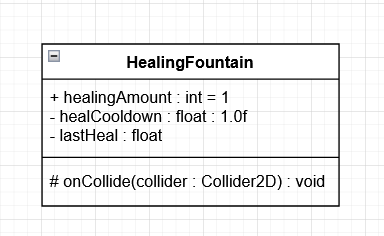
****

Figura 23. Diagrama UML pentru clasa HealingFountain

1. **Weapon, extinde Collidable**

Clasa aceasta este folosită pentru crearea obiectului care constituie arma jucătorului. Are rolul de a oferi acestuia opțiunea de atac, prin apăsarea tastei SPACE. Arma poate fi crescută în nivel pentru putere mai mare de atac, și o altă imagine.

Metodele Awake și Start sunt folosite pentru inițializarea câmpurilor/componentelor din clasă. Metoda Update() monitorizează comenzile introduse de utilizator de la tastatură, iar dacă tasta SPACE este detectată atunci se salvează timpul și se apelează metoda swing(). Aceasta setează trigger-ul de tip String numit Swing din Animatorul anim. Animația constă în rotirea armei în fața caracterului, timp în care Collider-ul acesteia este activat. Dacă sunt detectate coliziuni este apelată automat metoda onCollide(), ce trimite ca mesaj dauna provocată folosind obiecte de tip Damage. Acest mesaj este transmis folosind funcția Unity GameObject.SendMessage() [17].

Există, de asemenea, metode de îmbunătățire armă (upgradeWeapon), setarea nivelului armei (setWeaponLevel) sau schimbarea imaginii folosite ca armă (setWeaponSprite), care arată vizual nivelul curent al armei.

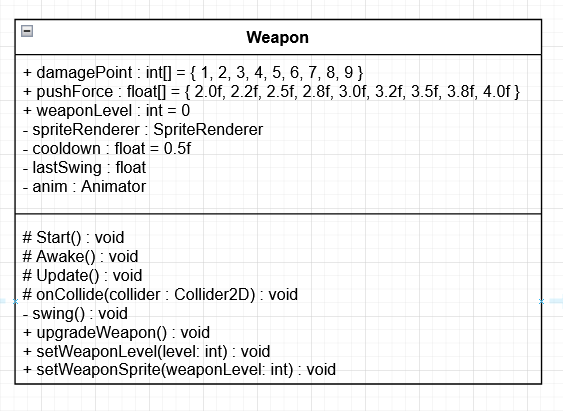
****

Figura 24. Diagrama UML pentru clasa Weapon

1. **Fighter, extinde MonoBehaviour**

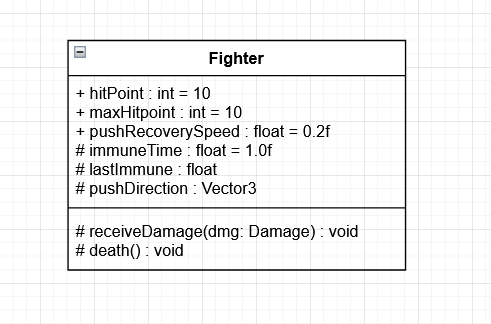
****

Figura 25. Diagrama UML pentru clasa Fighter

Clasa aceasta este baza obiectelor participante în luptă (jucătorul, inamicii, cutiile).

Oferă ca proprietăți nivel curent de viață (hitPoint), nivel maxim de viață (maxHitpoint) și rezistență la împingere (pushRecoverySpeed).

Toate obiectele de acest tip sunt capabile să sufere daune, care se realizează folosind metoda receiveDamage() și să moară, apelănd metoda death() când viața obiectului ajunge la 0. Metoda receiveDamage() este apelată când un mesaj de tip Damage este transmis către acest obiect. Dacă a trecut un timp mai mare decât perioada de imunitate după lovituri (immuneTime) de la ultima daună suferită (lastImmune) atunci obiectul este eligibil să fie lovit din nou, se salvează noul timp și afișează un mesaj de informare pe ecran.

1. **Mover, extinde Fighter**

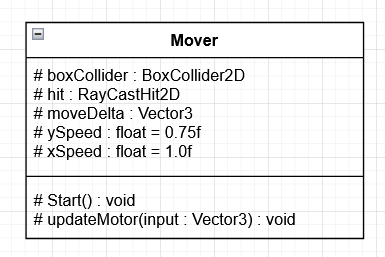
****

Figura 26. Diagrama UML pentru clasa Mover

Reprezintă un caz particular de combatanți, și anume care se pot mișca în jur (jucator, inamic). În plus, față de Fighter, conține metoda updateMotor(), care primește ca parametru un Vector3 ce reprezintă comanda de mișcare (ce poate fi atât inputul de la jucător, cât și algoritmul de control al inamicilor). Aceasta metodă gestionează mișcarea obiectulu. Dacă direcția de mers se schimbă, atunci se rotește imaginea obiectului pentru o mai ușoară percepție a mișcării executate. Pentru verificare se incearcă crearea unei “cutii” la poziția dorită raportată la poziția actuală a obiectului, iar dacă această încercare este cu succes atunci deducem că locația respectivă este liberă, validă pentru mutarea obiectului.

1. **Crate, extinde Fighter**

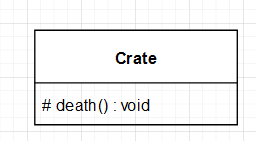
****

Figura 27. Diagrama UML pentru clasa Crate

Clasa aceasta definește Cutia, un combatant special cu rol de blocare a căii de acces, ce poate suferi daune și fi distrus dacă este lovit destul de către jucător. Atunci obiectul dispare iar trecerea prin locația respectivă este permisă. Pentru aceasta se suprascrie metoda death() cu instrucțiunea de distrugere a obiectului.

1. **Player, extinde Mover**

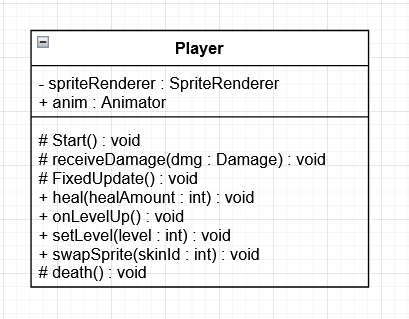
****

Figura 28. Diagrama UML pentru clasa Player

Clasa care reprezintă jucătorul. Conține o componentă Animator care controlează meniul de moarte. Această clasă tratează comenzile jucătorului(WASD sau săgețile sus-jos-stânga-dreapta), interacțiunile cu viața jucătorului(vindecare, solicită actualizarea barei de viață din HUD variabila de viață și-a schimbat valoarea), precum și setările de joc, cum ar fi schimbarea avatarului sau setarea nivelului.

Metoda FixedUpdate() este responsabilă cu monitorizarea comenzilor de mișcare, folosind metoda Input.GetAxisRaw() [24]. Această metodă poate fi apelată atât pentru axa orizontală cât și pentru cea verticală, și valorile returnate de aceasta (în funcție de tastele alese) pot fi schimbate din setările proiectului. La detectarea comenzilor de mișcare, este apelată metoda updateMotor() cu noua destinație, care va executa mișcarea daca este validă, fiind moștenită de la clasa Mover.

Metoda heal() este folosită pentru a crește valoarea vieții jucătorului. Aceasta poate fi apelată din mai multe surse, cum ar fi poțiunile sau fântâna de vindecare. Viața jucătorului este crescută cu cantitatea cerută, cu limitarea la nivelul maxim de viață ce poate fi atins la momentul respectiv. De asemenea, se afișează un text de informare pe ecran.

Metodele onLevelUp() și setLevel() sunt folosite pentru gestionarea nivelului caracterului. Dacă nivelul este incrementat, viața maximă este crescută cu 1, iar caracterul este vindecat cu o cantitate de viață prestabilită, cu limitare la viața maximă la momentul respectiv. Această metodă este apelată de asemenea la încărcarea primei scene, pentru a se asigura atribuirea corectă a vieții la începerea jocului dacă există progres salvat.

Metoda swapSprite() schimbă avatarul jucătorului în funcție de alegerea din meniul de pauză prin schimbarea imaginii atribuite componentei SpriteRenderer a obiectului.

La moarte, se setează trigger-ul de tip string Show al componentei Animator (anim), care declanșează apariția meniului de moarte, iar apoi obiectul este setat inactiv.

1. **Enemy, extinde Mover**

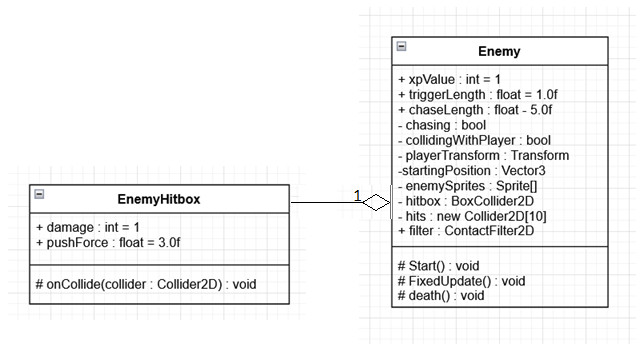


Figura 29. Diagrama UML pentru clasele Enemy și EnemyHitbox

Această clasă este baza tuturor inamicilor din joc. Script-ul acesta este atașat unui obiect de joc care reprezintă inamicul, și adaugă funcționalitatea de mișcare, lovire sau moarte. Inamicul este construit dintr-un obiect ce conține o imagine și un BoxCollider2D peste ea, care reprezintă “trupul” inamicului, adică unde poate fi lovit de către jucător. Conține, de asemenea un copil de tip BoxCollider2D, care reprezintă “hitbox”-ul inamicului, adică dimenisiunea care poate provoca daune înapoi. Acestui copil BoxCollider2D îi este atașat script-ul EnemyHitbox, evidențiind relația dintre cele două clase.

Pentru controlul mișcării este folosită metoda FixedUpdate(), care la fiecare apelare verifică dacă jucătorul este în raza de urmărire a inamicului (chaseLength). Daca se află și în raza de activare inițială (triggerLength), atunci variabila logică chasing este setată și se caută coliziune cu jucătorul. Dacă nu există, atunci se apelează updateMotor() pentru a se apropia de jucător mai mult. Dacă acesta se depărtează prea mult, chasing revine la false și inamicul la poziția inițială.

Dacă este lovit de destule ori încât viața scade la 0, atunci metoda death() este apelată, care distruge obiectul, actualizează contorul enemiesAlive din GameManager, jucătorul primește o cantitate de experiență (xpValue) și este afișat un mesaj de informare.

1. **EnemyHitbox, extinde Collidable**

Reprezintă caracteristicile de atac ale inamicului, reprezentate de daunele provocate (damage) și puterea de împingere (pushForce).

Metoda onCollide() este suprascrisă pentru a verifica coliziunea cu jucătorul, iar dacă aceasta este realizată atunci este trimis un mesaj de atac folosind funcția GameObject.SendMessage() [12] cu un obiect de tip Damage cu proprietățile menționate mai sus.

1. **HUD, extends MonoBehaviour**

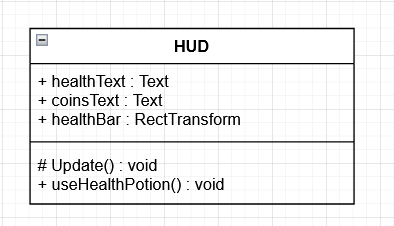
****

Figura 30. Diagrama UML pentru clasa HUD

Clasa HUD gestionează interfața grafică a utilizatorului, care este în permanență pe ecran. Conține referințe spre nivelul de viață și cantitatea de bani la fiecare cadru, precum și acțiunea atribuită butonului de poțiune de vindecare.

Metoda Update() actualizează viața și banii caracterului, prin schimbarea valorilor blocurilor Text (healthText, coinsText), cât și prin schimbarea dimensiunii imaginii de viață pentru reprezentarea vizuală a acesteia.

Metoda useHealthPotion() este apelată când butonul corespunzător ei este apăsat de către jucător. Aceasta verifică daca există în inventar poțiuni. Daca sunt, numarul lor este decrementat, jucătorul vindecat cu cantitatea dorită și un mesaj de informare este afișat pe ecran. În cazul contrar, un mesaj de informare asupra cantității insuficiente este prezentat.

1. **StartMenu, extinde MonoBehaviour**

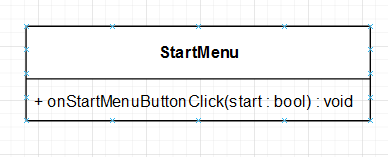
****

Figura 31. Diagrama UML pentru clasa StartMenu

Conține o singură metodă care gestionează acțiunile ce trebuiesc îndeplinite la apăsarea fiecărui buton. S-a folosit o singură metodă care primeste ca parametru o variabilă logică ce este setată true la apăsarea unui buton și false la apăsarea celuilalt. Pentru butonul de Start, acțiunea este încărcarea scenei corespunzătoare primului nivel de joc. Pentru butonul de Quit, acțiunea este închiderea jocului.

1. **DeathMenu, extinde MonoBehaviour**

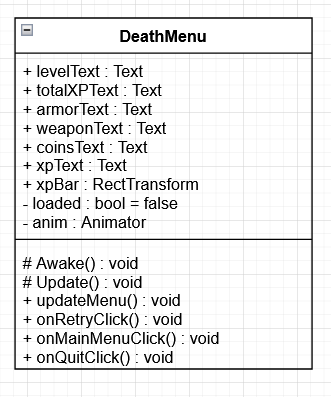
****

Figura 32. Diagrama UML pentru clasa DeathMenu

Acest script reprezintă controller-ul pentru meniul de moarte. Este atașat obiectului DeathMenu care este un copil al HUD-ului, și este activat din cod folosind metoda death() din clasa Player. Conține mai multe câmpuri Text pentru afișarea statisticilor jocului (informații despre level, experiență, armură, armă, bani), precum și cateva metode care tratează acțiunile specifice fiecărui buton prezent.

Metoda Update() este folosită pentru a afla momentul în care meniul este afișat, pentru a actualiza informațiile prin apelarea updateMenu(). După ce s-a facut acest lucru, câmpul logic loaded este setat pentru a se evita rularea la infinit a acestei instrucțiuni.

Metodele onRetryClick(), onMainMenuClick() și onQuitClick() sunt implementarea comportamentului specific fiecărui buton, și sunt apelate prin apăsarea butoanelor din meniu. Acestea redirecționează jocul spre meniul de start, scena principală sau închid aplicația, în funcție de butonul apăsat.

1. **PauseMenu, extinde MonoBehaviour**

Acesta este controller-ul pentru meniul de pauză/opțiuni. Este activat prin click pe butonul din stânga jos în formă de cufăr, și conține informații despre inventar (folosind variabile Text pentru actualizarea informațiilor), opțiuni de configurare a caracterului prin schimbarea avatarului sau a stilului de arme folosit și informații despre caracter cum ar fi level-ul curent, banii, experiența și viața.

Metodele onAvatarArrowClick() și onWeaponArrowClick() sunt metodele ce implementează comportamentul la apăsarea butoanelor de selecție avatar sau armă, care actualizează imaginea selectată prin apelarea metodelor de schimbare onSelectionChanged() și onWeaponChanged(). Acestea schimbă valorile variabilei chosenWeapon din GameManager sau schimbă avatarul jucătorului prin apelarea metodei changeSprite() din clasa Player.

Metoda updateMenu() este apelată în același timp cu activarea meniului, și are rolul de a actualiza informațiile încărcate pe ecran. Metodele onResetClick() și on QuitClick() reprezintă acțiunile pentru butoanele de Reset și Quit, care resetează progresul folosind metoda resetProgress() din clasa GameManager sau închid jocul.

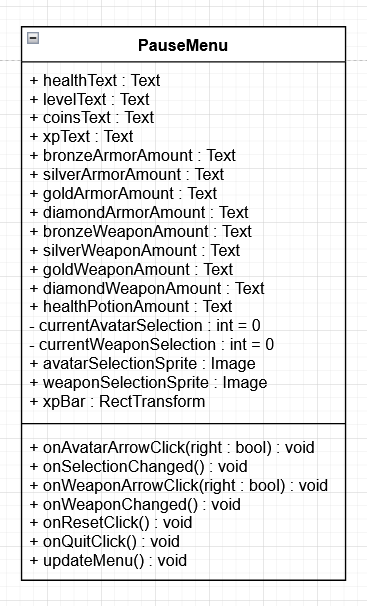
****

Figura 33. Diagrama UML pentru clasa PauseMenu

1. **CraftingMenu, extinde MonoBehaviour**

Script-ul acesta reprezintă controller-ul meniului de craftare. Acesta este activat prin apăsarea butonului din stânga jos în formă de ciocan. Afișează informații despre resursele deținute (folosind campurile Text enumerate), și optiuni de craftare cum ar fi armuri, arme sau poțiuni de vindecare.

Pentru a actualiza informațiile afișate există metoda updateMenu(), care este apelată odată cu încărcarea meniului, dar și din metoda Update() pentru a se actualiza în timp real resursele deținute în cazul în care este selectată vreo opțiune de craftare.

Metodele onCraftArmorClick(), onCraftWeaponClick() și onCraftPotionClick() sunt folosite pentru a declanșa acțiunea selectată prin apăsarea butoanelor, ceea ce presupune verificarea inventarului pentru resursele necesare craftării și afișarea unui mesaj de feedback folosind câmpul Text feedback.

Metodele onArmorArrowClick() și onWeaponArrowClick() sunt folosite pentru a naviga opțiunile de craftare pentru fiecare nivel de armură/armă, care au costuri diferite ce sunt de asemenea actualizate folosind câmpurile armorCostText, weaponCostText, armorCostSprite și weaponCostSprite.

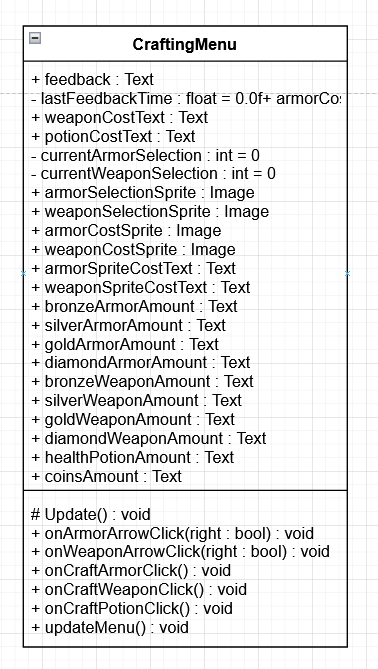
****

Figura 34. Diagrama UML pentru clasa CraftingMenu

1. **Sound Manager, extinde MonoBehaviour**

Este controller-ul pentru fișierele audio folosite în joc. Acesta conține o instanța a sa,

care este folosită pentru a apela metoda playSound(). Acesta conține mai multe câmpuri publice de tip AudioClip, ce reprezintă efectele sonore ce sunt folosite în joc. Acestea sunt ulterior rulate de către un obiect AudioSource.

În metoda Start() instanța instance este făcută de tip Singleton.

Metoda Update() este folosită pentru a crea bucla infinită cu muzica de fundal. Aceasta se repeta la fiecare 64 de secunde (lungimea melodiei), folosind variabila lastPlayed pentru a monitoriza progresul.

Metoda playSound() primeste ca parametru un șir de caractere ce reprezintă numele fișierului audio care trebuie rulat, și rulează fișierul cerut, cu volum prestabilit.

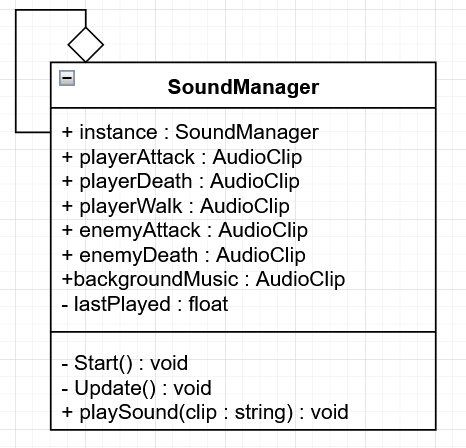


Figura 35. Diagrama UML pentru clasa SoundManager

# IMPLEMENTAREA FUNCȚIONALITĂȚILOR

**5.1 DIAGRAMA USE CASE**

Actorii principali ai jocului sunt jucătorul (player) și inamicii. Există acțiuni comune, precum Mișcarea sau Folosirea Armei (lovire). Actorul mai complex este Player-ul, care are în plus opțiuni de folosire meniuri (de pauza sau crafting), schimbarea setărilor (avatar diferit sau stil de arme scurte/lungi), folosirea poțiunilor de vindecare, sau interacționarea cu alte obiecte din jur, cum ar fi fântâna de vindecare, cuferele cu resurse, portale de teleportare, declanșatoare de acțiuni, NPC-uri de ghidaj. Toate acestea sunt vizibile în Figura 36.

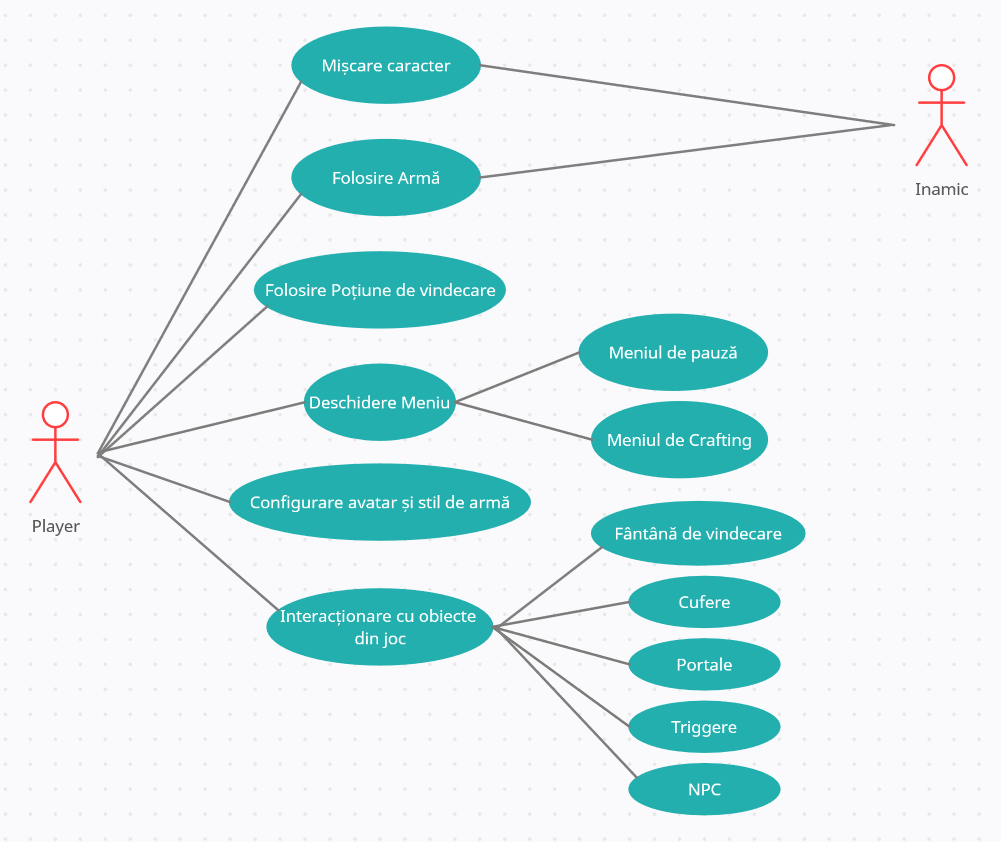


Figura 36. Diagrama Use Cases

**5.2 CONTROLUL MIȘCĂRILOR**

Clasa Mover implementează mișcarea ca funcționalitate generală în joc pentru obiectele care o moștenesc. Pentru aceasta există metoda updateMotor(), care primește ca parametru un Vector3 ce reprezintă comanda de mișcare, și, dacă mișcarea este posibilă atunci obiectul este mutat la noua poziție. Structura logică a acestui algoritm este prezentată în Figura 37 de mai jos, iar implementarea este prezentată in Cod 1.

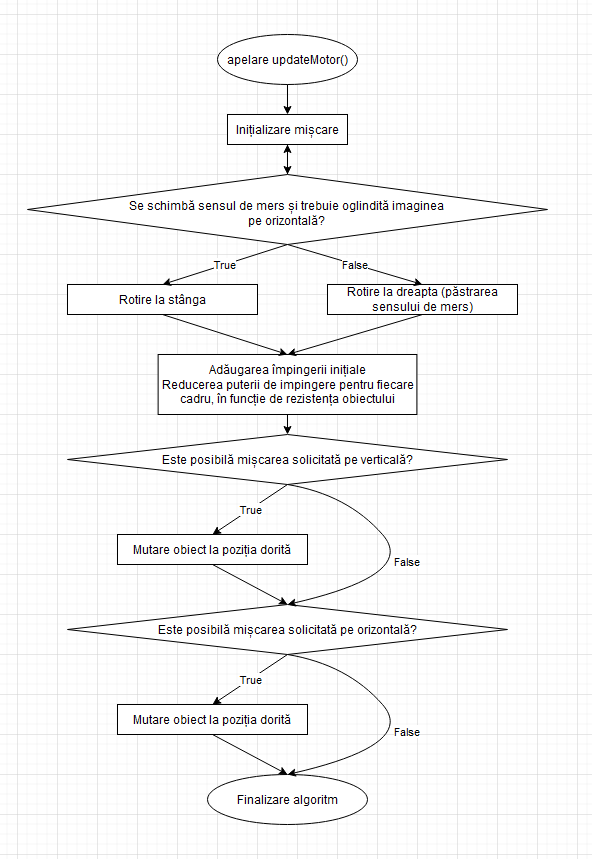
****

Figura 37. Schema logică a algoritmului de mișcare(updateMotor)

Această metodă este apelată de către entitățile cu abilitate de mișcare din joc. În cazul jucătorului, comenzile sale sunt primite de la tastatură folosind tastele WASD sau săgețile. Iar în cazul inamicilor comenzile lor sunt decise de catre algoritmul de decizie (FixedUpdate() din clasa Enemy).

1. protected virtual void updateMotor(Vector3 input) {
2. // reset moveDelta
3. moveDelta = new Vector3(input.x \* xSpeed, input.y \* ySpeed, 0);
5. // swap sprite direction, wether you go right of left
6. if (moveDelta.x > 0) {
7. // flip the sprite to right
8. //transform.localScale = Vector3.one;
9. transform.localScale = scale;
10. }
11. else if (moveDelta.x < 0) {
12. // flip the sprite to the left
13. //transform.localScale = new Vector3(-1, 1, 1);
14. transform.localScale = new Vector3(-1 \* scale.x, scale.y, scale.z);
15. }
17. // Add push vector if there is one
18. moveDelta += pushDirection;
20. // Reduce push force every frame, based off recovery speed
21. = Vector3.Lerp(pushDirection, Vector3.zero, pushRecoverySpeed);
23. // check for vertical collision
24. hit = Physics2D.BoxCast(transform.position, boxCollider.size, 0, new Vector2(0, moveDelta.y), Mathf.Abs(moveDelta.y \* Time.deltaTime), LayerMask.GetMask("Actor", "Blocking"));
26. // move the object if there were no collisions(cast a box on wanted position. If box is successfully casted - null returned - object can be moved)
27. if (hit.collider == null) {
28. transform.Translate(0, moveDelta.y \* Time.deltaTime, 0);
29. }
31. // check for horizontal collision
32. hit = Physics2D.BoxCast(transform.position, boxCollider.size, 0, new Vector2(moveDelta.x, 0), Mathf.Abs(moveDelta.x \* Time.deltaTime), LayerMask.GetMask("Actor", "Blocking"));
34. // move the object if there were no collisions(cast a box on wanted position. If box is successfully casted - null returned - object can be moved)
35. if (hit.collider == null) {
36. transform.Translate(moveDelta.x \* Time.deltaTime, 0, 0);
37. }
38. }

Cod 1. Implementarea metodei updateMotor() din cadrul clasei Mover

După inițializarea mișcării(prin înmulțirea componentelor orizontale și verticale cu vitezele prestabilite, linia 3), se verifică dacă este nevoie rotirea imaginiii obiectului spre dreapta/stânga pentru ca direcția de mers să fie percepută mai ușor (liniile 6 - 15). Se adaugă ulterior direcția de împingere (linia 18) și reducerea puterii acesteia cadru cu cadru (linia 21). Inițial, algoritmul nu știe dacă mișcarea solicitată este validă, de aceea se fac verificări pentru a se asigura că locația unde trebuie mutat obiectul nu este interzisă (blocată de alte obiecte, de pereți, etc). Pentru aceasta se foloseste metoda BoxCast, care încearcă să creeze un obiect la locația dorită, și returnează null dacă această încercare a avut succes.

Deoarce mișcarea conține doua axe, verificarile se fac separat pentru a permite mișcarea doar într-o direcție în cazul în care o axă este blocată. Aceste verificări sunt executate la liniiile 24 și 32, iar dacă au avut succes (liniiile 27, 35) atunci obiectul este translatat pe axa respectivă distanța cerută.

**5.2.1 CONTROLUL CARACTERULUI JUCĂTORULUI**

Controlul caracterului este implementat în fișierul script Player.cs, atașat obiectului de joc Player din ierarhie, și urmează logica unui automat cu stări finite, schema acestuia fiind prezentată în Figura 38 de mai jos.

La încărcarea scenei, sunt inițializate în cod diferite componente ale obiectului (cum ar fi imaginea sa, animatorul, boxCollider-ul necesar coliziunilor) folosind metoda Start().

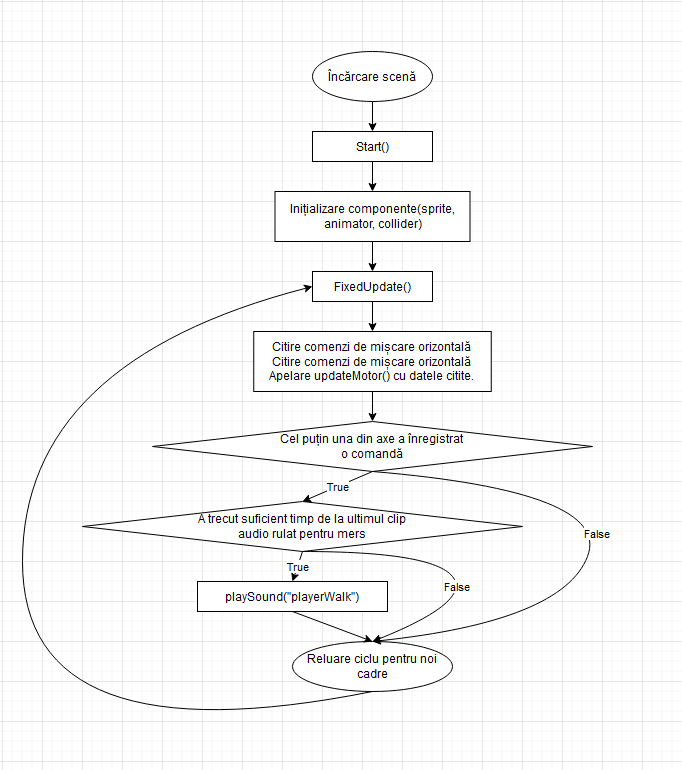


Figura 38. Schema logică a script-ului de controlare a mișcărilor caracterului

După aceasta, folosind FixedUpdate() (Cod 2), s-a creat o buclă infinită de detectare comenzi și actualizare a poziției caracterului pe ecran. Această metodă a fost preferată față de Update() deoarece forțează apelările repetate sa respecte un inverval strict de timp, pentru a evita raspunsuri diferite pe sisteme mai slabe/puternice.

În interiorul acestei metode, se citesc input-urile de la utilizator folosind GetAxisRaw() pe orizontală (linia 9) și pe verticală (linia 12). Acestea sunt stocate in două variabile x, respectiv y, care sunt apoi trimise ca parametri metodei updateMotor() (linia 15). Această metodă este apelată mereu deoarece jucătorul poate fi mișcat și dacă nu introduce comenzi dar este lovit de un inamic, prin împingere inapoi. Dacă a fost detectată cel putin o comandă de mișcare (linia 17), se verifică dacă este necesară rularea efectului audio de mers (linia 18). Pentru aceasta se compară diferența de timp de la ultima rulare a sunetului până la momentul actual, iar dacă este mai mare de 0.4 secunde atunci se rulează sunetul (linia 19) și noul timp se salvează. Această verificare este necesară pentru a evita suprapunerea de efecte audio la fiecare reapelare a metodei FixedUpdate()

1. protected void FixedUpdate()
2. {
3. // GetAxisRaw() returns
4. // -1 : negative input key is pressed
5. // 0 : no key pressed
6. // 1 : positive input key is pressed
8. // read horizontal movement
9. float x = Input.GetAxisRaw("Horizontal");
11. // read vertical movement
12. float y = Input.GetAxisRaw("Vertical");
14. // call move function
15. updateMotor(new Vector3(x, y, 0));
16. if(x != 0 || y != 0)
17. if(Time.time - lastPlayed > 0.4f) {
18. GameManager.instance.playSound("playerWalk");
19. lastPlayed = Time.time;
20. }
21. }

Cod 2. Metoda FixedUpdate() din cadrul clasei Player

**5.2.2 CONTROLUL INAMICILOR**

Inamicii din joc necesită un mod de a decide comenzile necesare mișcărilor, ce sunt apoi trimise spre updateMotor(). Pentru acesta, este folosită metoda FixedUpdate(), care are rolul de a analiza poziția obiectului raportată la jucător, și acționează corespunzător. Logica acestui algoritm este prezentată în Figura 39, fiind construită din o serie de blocuri de instrucțiuni și interogare, iar implementarea codului este prezentată in Cod 3.

La Start() se inițializează variabilele pentru distanța de activare, urmărire, și poziția inițială a fiecărui obiect de tip Enemy, poziția jucătorului, precum și hitbox-ul atașat.

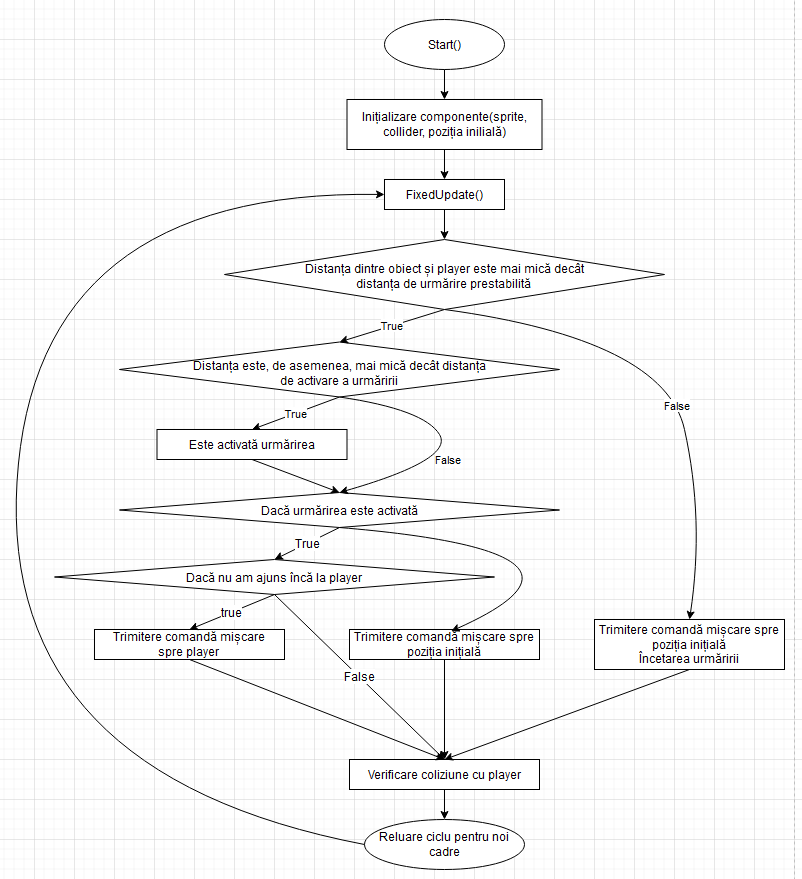


Figura 39. Schema logică a algoritmului de controlare a mișcărilor inamicilor

Dacă jucătorul este în interiorul distanței de urmărire (linia 3), se activează obiectul (linia 5) dacă distanța este mai mică decât cea de activare (linia 4). Dacă obiectul este deja activat (linia 7) și nu s-a detectat încă coliziune cu jucătorul (linia 8), atunci este trimisă comanda de apropiere de acesta (linia 10). Dacă urmărirea este dezactivată (linia 13), atunci este comandată revenirea la poziția inițială (linia 15). Dacă jucătorul se îndepărtează prea mult (linia 18), se dezactivează urmărirea (linia 21) și se revine la poziția inițială (linia 20).

În final, se verifică coliziunea cu player-ul, pentru a ști cum trebuie decis comportamentul pentru ciclul următor (liniile 25 – 40).

1. private void FixedUpdate() {
2. // is the player in range?
3. if (Vector3.Distance(playerTransform.position, startingPosition) < chaseLenght) {
4. if (Vector3.Distance(playerTransform.position, startingPosition) < triggerLength)
5. chasing = true;
7. if (chasing) {
8. if (!collidingWithPlayer) {
9. // move enemy towards player
10. updateMotor((playerTransform.position - transform.position).normalized);
11. }
12. }
13. else {
14. // go back to starting position
15. updateMotor(startingPosition - transform.position);
16. }
17. }
18. else {
19. // go back to starting position
20. updateMotor(startingPosition - transform.position);
21. chasing = false;
22. }
24. // check for overlaps
25. collidingWithPlayer = false;
27. // Collision work
28. boxCollider.OverlapCollider(filter, hits);
30. for (int i = 0; i < hits.Length; i++) {
31. if (hits[i] == null)
32. continue;
34. if (hits[i].tag == "Fighter" && hits[i].name == "Player") {
35. collidingWithPlayer = true;
36. }
38. // reset value
39. hits[i] = null;
40. }
41. }

Cod 3. Metoda FixedUpdate() din cadrul clasei Enemy

**5.3 SISTEMUL DE LUPTĂ**

Toate entitățile participante în luptă au la dispoziție acțiuni de suferirea de daune și moartea (cănd nivelul vieții scade la 0). Acestea sunt implementate în clasa Fighter (obiectele de tip Crate sunt de acest fel, de exemplu), prin metodele receiveDamage() și death(). Ulterior, obiecte mai complexe (cum sunt jucătorul și inamicii) au și opțiunea de atac, care se realizează fie folosind arma echipată (în cazul jucătorului) sau prin atingerea jucătorului cu un BoxCollider numit hitbox (în cazul inamicilor). Implementarea acestor metode este prezentată in Cod 5.

Pentru crearea sistemului de rezistețe (viață, rezistență la împingere, timpul de imunitate), sunt folosite mai multe variabile, după cum se observă în Cod 4. Acestea au rolul de a monitoriza pe parcursul jocului nivelul de viață al fiecărui combatant, fiind actualizate de fiecare dată când obiectul suferă daune, este vindecat, este crescut nivelul maxim de viață sau rezistența la împingere prin nivelul mai ridicat al armurii.

1. // health management
2. public int hitPoint = 10;
3. public int maxHitPoint = 10;
4. public float pushRecoverySpeed = 0.2f;
6. // Immunity
7. protected float immuneTime = 1.0f;
8. public float lastImmune;
9. // Push
10. Protected Vector3 pushDirection;

Cod 4. Variabilele folosite in luptă din cadrul clasei Fighter

Pentru a suferi daune există metoda receiveDamage (linia 2), care primește ca

parametru un obiect de tip Damage ce conține daunele provocate, originea atacului și puterea de împingere. Ea este apelată folosind metoda SendMessage() din cadrul Unity, ce permite transmiterea spre obiectul apelant atacul ca paramtetru. Dacă a trecut destul timp de la ultimul atac suferit, prin compararea diferenței dintre timpul actual (Time.time) și ultimul atac (lastImmune) cu perioada de imunitate a obiectului (immuneTime), atunci noul timp este salvat (linia 4), din nivelul de viață sunt scăzute daunele primite (linia 5), și se setează împingerea (linia 6).

Dacă obiectul lovit este jucătorul, atunci este redat și sunetul de atac al inamicilor (liniile 8, 9). Am preferat să aplic aici sunetul atacului inamicilor deoarece aceștia la coliziune trimit atacuri mereu, ceea ce ar fi suprapus foarte multe clipuri audio în același timp.

De asemenea, este afișat un mesaj de informare asupra daunelor suferite pe ecran folosind un obiect de tip FloatingText (linia 10).

După atac, dacă nivelul de viață a scăzut la 0 (linia 12), viața este adusă la 0 (linia 13) și este apelată metoda death() (linia 14), care inițial conține doar un apel de rulare a efectului sonor de moarte a inamicilor (linia 20), dar ulterior poate fi suprascrisă de clasele care o moștenesc pentru a personaliza comportamentul acelor obiecte la moarte.

1. // All fighters can receive damage and die
2. protected virtual void receiveDamage(Damage dmg) {
3. if (Time.time - lastImmune > immuneTime) {
4. lastImmune = Time.time;
5. hitPoint -= dmg.damageAmount;
6. pushDirection = (transform.position - dmg.origin).normalized \* dmg.pushForce;
8. if (name == "Player")
9. GameManager.instance.playSound("enemyAttack");
10. GameManager.instance.showText(dmg.damageAmount.ToString(), 15, Color.red, transform.position, Vector3.zero, 0.5f);
12. if (hitPoint <= 0) {
13. hitPoint = 0;
14. death();
15. }
16. }
17. }
19. protected virtual void death() {
20. GameManager.instance.playSound("enemyDeath");
21. }

Cod 5. Implementarea metodelor receiveDamage() și death() din cadrul clasei Fighter

**5.3.1 IMPLEMENTAREA ACESTUIA PENTRU CARACTERULUI JUCĂTORULUI**

Jucătorul poate ataca obiectele din jur folosind arma echipată. Aceasta este reprezentată de un obiect de tip Weapon ancorat în colțul dreapta sus al obiectului Player. La primirea comenzii de atac, este rulată o animație care rotește arma în fața jucătorului, timp în care componenta BoxCollider2D a armei este activă și poate interacționa cu obiectele din jur. Pentru proprietățile armei sunt folosite mai multe variabile, după cum se observă în Cod 6.

1. // Damage struct
2. public int[] damagePoint = { 1, 2, 3, 4 };
3. public float[] pushForce = { 3.0f, 3.5f, 4.0f , 5.0f};
5. // Upgrade
6. public int weaponLevel = 0;
8. // Swing weapon
9. private float cooldown = 0.5f;
10. private float lastSwing;

Cod 6. Variabilele folosite pentru implementarea proprietăților armelor, din clasa Weapon

Tabelele damagePoint (linia 2) și pushForce (linia 3) sunt folosite pentru a seta puterea armei pentru fiecare nivel, variabila weaponLevel (linia 6) păstrează evidența nivelului curent al armei echipate, iar variabilele cooldown și lastSwing (liniile 9, 10) sunt folosite pentru a defini frecvența cu care pot fi declanșate atacuri.

Metoda Update() este folosită pentru a detecta apăsarea tastei SPACE (linia 3), care apelează swing() (linia 6) dacă este posibil atacul (adică să nu fie prea apropiat de ultimul atac lansat, linia 4).

Metoda swing() este folosită pentru a reda efectul sonor pentru folosirea armei (linia 28) și setează trigger-ul animatorului armei (linia 27, care va rula animația de rotire a armei).

1. protected override void Update() {
2. base.Update();
3. if (Input.GetKeyDown(KeyCode.Space)) {
4. if (Time.time - lastSwing > cooldown) {
5. lastSwing = Time.time;
6. swing();
7. }
8. }
9. }
11. protected override void onCollide(Collider2D collider) {
12. if (collider.tag == "Fighter") {
13. if (collider.name == "Player")
14. return;
15. // Create a new Damage object, and send it to the figher hit
16. Damage dmg = new Damage {
17. damageAmount = damagePoint[weaponLevel],
18. origin = transform.position,
19. pushForce = pushForce[weaponLevel]
20. };
21. // send the message with the damage to the hit fighter
22. collider.SendMessage("receiveDamage", dmg);
23. }
24. }
26. private void swing() {
27. anim.SetTrigger("Swing");
28. GameManager.instance.playSound("playerAttack");
29. }

Cod 7. Implementarea metodelor Update(), onCollide() și swing(), din clasa Weapon

Metoda onCollide() este suprascrisă deoarece este moștenită de la clasa Collidable. Aici se ignoră coliziunile dintre armă și jucător (liniile 12-14), iar dacă sunt detectate coliziuni cu alți combatanți, se crează un nou obiect de tip Damage cu proprietățile curente ale armei (liniile 16-20), care apoi este transmis spre obiectul cu care a fost detectată coliziunea (linia 22) folosind metoda SendMessage(), care va apela metoda receiveDamage() a obiectului în cauză cu structura de daune creată mai sus.

Când jucătorul a suferit daune prea multe și viața a ajuns la 0, metoda death() este apelată. Această metodă este suprascrisă in script-ul Player pentru a schimba comportamentul obiectului la moarte, iar implementarea este prezentată in Cod 8. Se rulează efectul audio corespunzător morții caracterului (linia 2), se setează declanșarea meniului de moarte pe ecran (linia 3), iar obiectul Player este setat ca inactiv (linia 4), pentru a opri orice acțiune din joc.

1. protected override void death() {
2. GameManager.instance.playSound("playerDeath");
3. anim.SetTrigger("Show");
4. gameObject.SetActive(false);
5. }

Cod 8. Implementarea metodei death() din clasa Player

**5.3.2 IMPLEMENTAREA ACESTUIA PENTRU INAMICI**

Pentru a putea ataca, obiectele de tip Enemy trebuie sa aibă un copil de tip BoxCollider2D, care are atașat script-ul EnemyHitbox. Acest copil reprezintă “arma” inamicului, care atacă jucătorul prin coliziunea cu acesta. Implementarea acestei interacțiuni este facută în script-ul EnemyHitbox prin suprascrierea metodei onCollide() (prezentată în Cod 9), deoarece această clasă moștenește clasa Collidable.

Pentru aceasta, se verifică coliziunea să fie efectuată cu jucătorul prin verificarea tag-ului și a numelui obiectului în coliziune cu hitbox-ul (linia 6), iar dacă această verificare este pozitivă, un nou obiect de tip Damage este creat (liniile 8-12) cu proprietățile de atac ale inamicului (variabilele damage și pushForce de la liniile 2 și 3), ce este apoi transmis spre Player folosind metoda SendMessage() (linia 14) .

1. // Damage
2. public int damage = 1ș
3. float pushForce = 3;
4. protected override void onCollide(Collider2D collider) {
5. if (collider.tag == "Fighter" && collider.name == "Player") {
6. // Create a damage object, and send it to player
7. Damage dmg = new Damage {
8. damageAmount = damage,
9. origin = transform.position,
10. pushForce = pushForce
11. };
12. collider.SendMessage("receiveDamage", dmg);
13. }
14. }

Cod 9. Implementarea metodei onCollide() din clasa EnemyHitbox

La înfrângerea unui inamic jucătorul este recompensat cu valoarea de experiență atribuită inamicului respectiv, de aceea este nevoie de suprascrierea metodei death() din clasa Enemy. Implementarea acesteia este prezentată in Cod 10.

Implementarea din clasa Fighter este păstrată prin apelarea base,death(), deoarece conține comanda de rulare a efectului audio pentru moartea inamicului. Obiectul este apoi distrus (linia 3), contorul inamicilor în viață la momentul respectiv(variabila enemiesAlive din obiectul GameManager) este actualizat (liniile 4 și 5), apoi jucătorului îi este adaugată cantitatea de experientă a inamicului (care este reprezentată de câmpul xpValue, linia 6), împreună cu afișarea pe ecran a unui mesaj de tip FloatingText pentru informare.

1. protected override void death() {
2. base.death();
3. Destroy(gameObject);
4. if(GameManager.instance.enemiesAlive > 0)
5. GameManager.instance.enemiesAlive--;
6. GameManager.instance.grantXP(xpValue);
7. GameManager.instance.showText("+" + xpValue + " XP", 30, Color.magenta, transform.position, Vector3.up \* 40, 1.0f);
8. }

Cod 10. Implementarea metodei death() din clasa Enemy

**5.4 ADMINISTRAREA RESURSELOR ȘI A PROGRESULUI**

Pentru a gestiona toate aceste variabile și obiecte există un obiect gol în ierarhie numit GameManager, care are atașat script-ul GameManager și inventarul jucătorului, care este un obiect gol numit Inventory, cu script-ul Inventory atașat. GameManager conține și script-ul DontDestroy, pentru a nu fi pierdute valorile lui la schimbarea scenelor. Acest obiect conține o instanță a sa în interior (care este folosită pentru apelarea metodelor din ea/ accesarea valorilor diferitelor atribute) ce este facută de tip Singleton, pentru a nu permite co-existarea mai multor obiecte de acest tip în timpul rulării.

Conține câmpuri ce reprezintă resursele grafice (avatar-urile disponibile, imaginile armelor, armurilor, a tipurilor de arme), lista prețurilor de craftare și lista costurilor de experiență pentru a fiecare nivel. Există informații despre caracterul jucătorului, cum ar fi tabele cu proprietățile armurilor pe nivele, referințe ale obiectului Player și a armei atașate acestuia (de tip Weapon), nivelul curent al armurii, alegerile cosmetice(avatar, stilul de armă dorit). Există câmpuri de monitorizare a numărul inamicilor în viață, experiența jucătorului precum și ultima poziție a acestuia înainte de părăsirea scenei. De asemenea, conține referințe spre obiectele din ierarhia jocului ce nu se distrug, cum ar fi managerul de texte, managerul de sunete, interfața de joc (hud) cu meniurile (de pauză, crafting, moarte) incluse.

Aici se află metodele cele mai des folosite din joc, cum ar fi showText(), playSound(), grantXP(), onHitpointChange(), onInventoryChange(), resetProgress(), etc. Acestea sunt construite în așa fel să fie cât mai simplu de apelat din afara clasei, folosind instanța de GameManager. Astfel se evită înlănțuiri foarte lungi de obiecte pentru a se ajunge la o metodă dorită(de exemplu, pentru a se reda un efect sonor, se folosește apelul GameManager.instance.playSound(), față de GameManager.instance.soundManager .instance.playSound()).

Pentru gestionarea progresului făcut, există trei metode folosite : loadState(), saveState() și resetProgress().

* **saveState()**

Este apelată la fiecare părăsire de scenă, precum și la închiderea jocului, și are rolul

de a salva progresul făcut, precum și alegerile jucătorului, cum ar fi avatarul sau tipul de armă folosit. Implementarea metodei este prezentată parțial in Cod 11.

1. public void saveState() {
2. string s = "";
4. s += chosenAvatar.ToString() + "|"; // 0
6. // save inventory
7. s += inventory.coins.ToString() + "|"; // 1
8. s += inventory.healthPotions.ToString() + "|"; // 2
9. .....
11. // save armor level
12. s += armorLevel.ToString(); // 16
14. PlayerPrefs.SetString("saveState", s);
15. }

Cod 11. Implementarea metodei saveState() din clasa GameManager

Pentru aceasta, este creat un șir de caractere (linia 2) la care sunt adăugate pe rând

informațiile care trebuie salvate, urmate de caracterul "|", care are rol de separator, pentru a fi mai ușor de procesat la încărcare (liniile 4, 7, 8, etc). După ce toate informațiile au fost adăugate, este folosită metoda SetString() pentru a păstra acest șir de caractere în memorie, cu cheia "saveState" (linia 15), care ulterior va fi căutat și folosit pentru a se încărca progresul.

* **loadState()**

Este folosită pentru a încărca informațiile principale ale jucătorului la încărcarea

scenelor sau a jocului. Implementarea ei este prezentată succint mai jos, în Cod 12.

La apelarea acestei metode, se verifică dacă există în memorie cheia "saveState" (linia 5), caz în care există informații salvate și se pot încărca în joc, în caz contrat fiind terminată execuția acestei metode. Dacă cheia există, pentru a accesa datele asociate ei este folosită metoda GetString() care returnează șirul creat la salvarea datelor, iar pentru a fi mai ușor de lucrat acesta este împărțit în subșiruri delimitate de caracterul "|" folosind metoda Split() (linia 9). După aceasta, fiecare subșir este procesat prin transformarea sa în întreg, și valoarea aceasta folosită pentru a inițializa diferitele câmpuri ale instanței (liniile 11, 14, 15, etc). În final, sunt apelate metodele de actualizare a inventarului (linia 19) și a vieții caracterului (linia 20).

1. // load game state method
2. public void loadState(Scene s, LoadSceneMode mode) {
4. SceneManager.sceneLoaded -= loadState;
5. if (!PlayerPrefs.HasKey("saveState"))
6. return;
8. // split it into "0", "15", "20", "0"
9. string[] data = PlayerPrefs.GetString("saveState").Split('|');
11. chosenAvatar = int.Parse(data[0]);
13. // load inventory
14. inventory.coins = int.Parse(data[1]);
15. inventory.healthPotions = int.Parse(data[2]);
17. .....
19. onInventoryChange();
20. onHitpointChange();
21. }

Cod 12. Implementarea metodei loadState() din clasa GameManager

* **resetProgress()**

Această metodă poate fi apelată atât de butonul de Reset din meniul de pauză, cât și din cod, folosind instanța GameManager. Este folosită pentru a reseta progresul făcut până la acel moment. Implementarea este evidențiată parțial în Cod 13.

Acest lucru include inventarul complet(bani, armuri, arme, poțiuni), viața curentă și viața maximă a jucătorului, experiența, alegerile de avatar și armă, precum și variabila de monitorizare a ultimei poziții a jucătorului. Dupa ce inventarul (linia 12) și viața sunt actualizate (linia 13), este salvat noul progres folosind metoda saveState() (linia 14) și un mesaj de informare este afișat pe ecran folosind FloatingText (linia 15).

1. public void resetProgress() {
2. //// reset inventory
3. // reset coin amount
4. inventory.coins = 0;
6. ......

9. // reset last position
10. lastPosition = Vector3.zero;
12. onInventoryChange();
13. onHitpointChange();
14. saveState();
15. showText("Progress was reset!", 20, Color.red, player.transform.position, Vector3.up, 1.0f);
16. }

Cod 13. Implementarea metodei resetProgress() din clasa GameManager

**5.5 EFECTELE SONORE**

Pentru gestionarea sunetelor în joc, există o clasă specială numită SoundManager. În ierarhie este creat un obiect gol, cu numele SoundManager, căruia acest script îi este atașat, precum și o componentă AudioSource ce are rolul de a prelua clipurile audio și să le redea când este nevoie, având posibilitatea reglării volumului. Declararea câmpurilor și metoda Start() se pot observa în Cod 14.

Fiecare efect sonor folosit trebuie adăugat în această clasă ca un atribut public (liniile 2 – 4), care este apoi folosit de către componenta AudioSource (linia 7) pentru a fi redat. Clasa conține o instanță a sa (linia 1) de tip Singleton (liniile 13 - 15), care este folosită pentru a putea reda sunete de oriunde din cod.

1. public static SoundManager instance;
2. public AudioClip playerAttack, playerDeath, playerWalk;
3. public AudioClip enemyAttack, enemyDeath;
4. public AudioClip backgroundMusic;
6. private float lastPlayed;
7. private AudioSource audioSrc;
9. private void Start()
10. {
11. // make instance a Singleton
12. if (SoundManager.instance != null) {
13. Destroy(gameObject);
14. return;
15. }
16. else {
17. lastPlayed = -64f;
18. audioSrc = GetComponent<AudioSource>();
19. }
20. instance = this;
21. }

Cod 14. Declararea câmpurilor și metoda Start() din clasa SoundManager

Există o metodă playSound() (a cărei implementre poate fi vazută în Cod 15) care

primește ca parametru un șir de caractere ce reprezintă numele clipului audio dorit a fi rulat. Cu ajutorul unui bloc de tip switch (linia 2), folosind componenta AudioSource este rulat efectul sonor dorit, apelând metoda PlayOneShot() (liniile 4, 8, etc). Această metodă permite folosirea unui al doilea parametru de tip float, cu valori între 0 și 1 ce reprezintă volumul cu care trebuie rulat efectul sonor.

1. public void playSound(string clip) {
2. switch (clip) {
3. case "backgroundMusic":
4. audioSrc.PlayOneShot(backgroundMusic, 0.05f);
5. lastPlayed = Time.time;
6. break;
7. case "playerAttack":
8. audioSrc.PlayOneShot(playerAttack, 0.7f);
9. break;
10. ......
11. }
12. }

Cod 15. Implementarea parțială a metodei playSound() din clasa SoundManager

**5.6 ANIMAȚIILE**

Toate animațiile din joc sunt realizate cu ajutorul uneltei Animator [25] din Unity. Aceasta permite crearea de cadre intermediare dacă sunt setate cel inițial și cel final, creând astfel o mișcare fluidă, fără a mai fi nevoie de animație stop cadru (care constă în imagini diferite pentru fiecare cadru în parte, fiind mult mai costisitoare din punctul de vedere al timpului). Interfața acestui editor de animații din Unity se poate vedea în Figura 40.

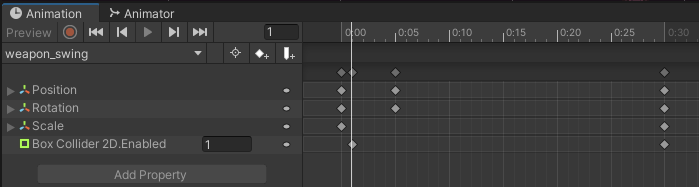


Figura 40. Interfața de creare și editare a animațiilor din Unity. Animația weapon\_swing

Se pot selecta componentele care trebuie sa fie folosite în timpul animației, permițând editarea valorilor acestora(cum ar fi BoxCollider, poate fi activat sau dezactivat, în cazul animației de rotire a armei). Acest lucru face implementarea mai ușoară, deoarece, în cazul armei, aceasta ar trebui să fie inactivă și activată coliziunea doar în timpul animației. În partea dreaptă este editorul de cadre. Pentru armă există inițializate patru cadre(cel inițial, cel imediat următor în care se activează collider-ul armei, un cadru care este extinderea maximă a armei, iar cadrul final aduce arma la poziția inițială, și dezactivează collider-ul din nou. Restul cadrelor sunt generate de către unealta Animator [25].

Toate animațiile trebuie să aparțină unui controller de animații. Acesta este folosit pentru a seta comportamentul animațiilor, tranzițiile dintre ele și durata acestora. Interfața uneltei Animator [25] este prezentată în Figura 41.

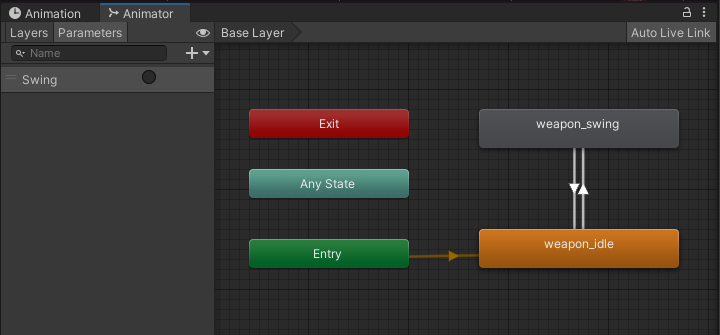


Figura 41. Interfața uneltei Animator din Unity. Controller-ul Weapon

Animatorul pentru armă (Weapon) conține un parametru de tip Trigger, numit Swing. Acesta este folosit pentru a porni animația armei, și este setat în metoda swing() din clasa Weapon, la apăsarea tastei SPACE. Animația standard este cea de weapon\_idle, în care arma este inactivă, și este repetată până când parametrul Swing este setat și solicită tranziția spre animația weapon\_swing. După ce rularea acesteo animații se termină, parametrul Swing este “consumat” și valoarea lui se resetează, solicitând tranziția înapoi spre animația weapon\_idle care își reia ciclul.

Meniurile din joc au animații identice, de acea sunt prezentate doar animațiile și controller-ul pentru meniul de pauză (pauseMenu). Există iarăși doua animații, pauseMenu\_show și pauseMenu\_hidden (Figura 42), și animația creată este cea de apariție a meniului din afara ecranului, intr-o mișcare de sus în jos.

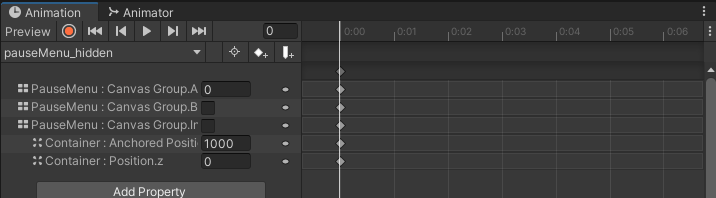


Figura 42. Interfața de editare a animației pauseMenu\_hidden

Proprietățile folosite pentru animație sunt pozitia acestuia (pe axa verticală), și cele din Canvas Group, pentru activarea/dezactivarea meniului, fiind imposibil de selectat dacă nu este activat. Aceste animații nu au decât un singur cadru, cel inițial, deoarece la finalizarea mișcării meniul este static.

Animatorul pentru acest meniu este prezentat în Figura 43, și este identic pentru celelalte meniuri (startMenu, craftingMenu și deathMenu).

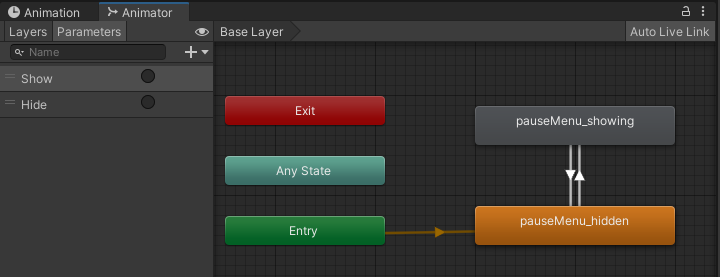


Figura 43. Interfața uneltei Animator din Unity. Controller-ul pauseMenu

Acesta conține doi parametri de tip Trigger, numiți Show și Hide. Show este folosit pentru a activa tranziția spre animația de afișare a meniului (fiind setat la acționarea butonului de meniu din interfața HUD), iar cel de Hide este folosit pentru a face tranziția la animația de ascundere a meniului (fiind setat la click în afara meniului, care este acoperit de un buton de dimensiunea ecranului). Nu este posibilă folosirea unui singur parametru deoarece valoarea parametrului este resetată după ce este folosită, ceea ce ar obliga tranziția spre celalată animație imediat.

Animația standard este cea de meniu ascuns, care rămâne valabilă până când Show este setat, când se face tranziția spre animația de meniu vizibil, care rămâne activă până Hide este setat.

# TESTAREA PE CONFIGURAȚII DIFERITE

Pentru analizarea performanțelor jocului la rulare am decis să export jocul ca

executabil folosind opțiunea de Build din Unity. Aceasta permite compilarea proiectului și generarea unui executabil pentru rulare, precum și folderele cu resurse (audio, grafice, etc) necesare jocului. Platforma aleasă pentru testare este Windows, dar Unity pune la dispoziție și opțiuni pentru MacOS, Linux, Android sau WebGL (joc în browser).

Pentru monitorizarea resurselor sistemului am folosit programul HWMonitor [26], care are o versiune gratuită de descărcare, și permite vizualizarea în timp real a datelor (cum ar fi temperaturile, utilizarea în procente, frecvența de funcționare sau eficiența energetică), precum și a unor statistici de valori minime sau maxime. Interfața acestui program poate fi observată în Figura 44.

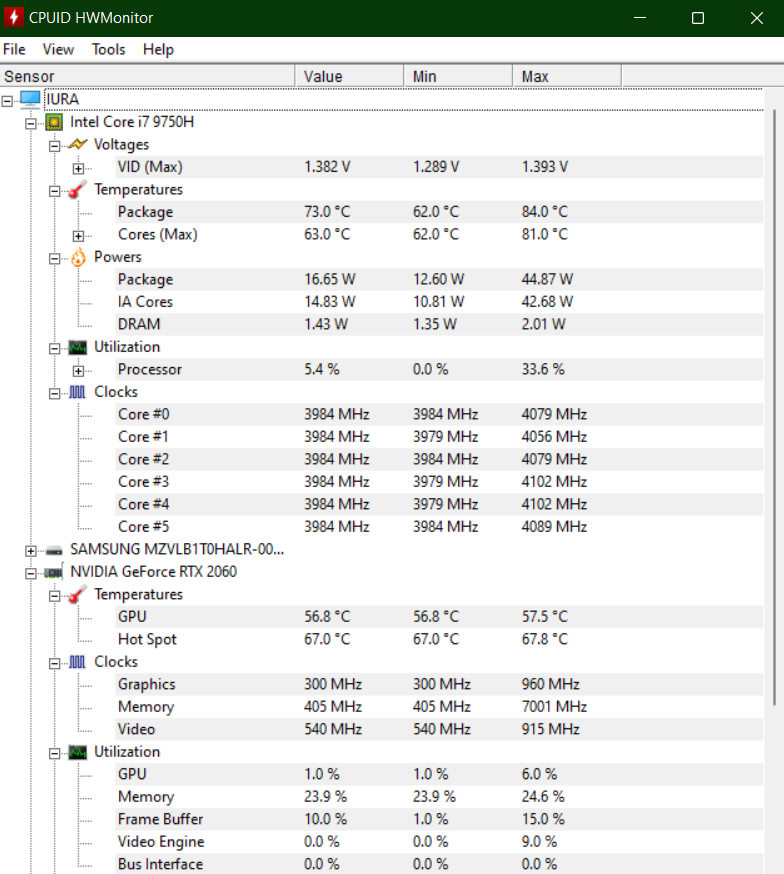


Figura 44. Interfața aplicației HWMonitor, folosită la testare

Jocul a fost testat pe 4 sisteme cu configurații diferite, iar componentele principale ale fiecărui sistem (tipul dispozitivului, modelul de procesor, placa video, memoria RAM, versiunea de Windows folosită) sunt prezentate în Tabel 2.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Tipul dispozitivului** | **Sistemul de operare** | **Procesor** | **Placa video** | **Memoria RAM** |
| LAPTOP Lenovo | Windows 11 | Intel Core i7 -9750H | Nvidia GeForce RTX2060 | 32GB |
| LAPTOP  Dell | Windows 11 | Intel Core i7 -9850H | Nvidia Quattro t2000 | 32GB |
| LAPTOP Lenovo | Windows 10 | Intel Core i7 -4710MQ | Nvidia GeForce 840m | 16GB |
| LAPTOP  Hp | Windows 10 | Intel Core i5 -3360M | AMD Radeon HD 7570M | 4GB |

Tabel 2. Comparație între mediile de test folosite

Pentru a extrage statisticile cu privire la consumul resurselor sistemului, am rulat teste cu dispozitivul în idle, fără alte sarcini executate, acesta fiind mediul meu de control (statisticile standard) cu care am putut apoi să compar datele din timpul rulării jocului. Datele au fost procesate, apoi transformate în diagrame folosind editorul online charts.livegap.com [27], pentru interpretarea mai ușoară a rezultatelor.

În Figura 45 se poate observa grafic diferența de utilizare a procesorului (în procente) între idle și rularea jocului. Primele doua dispozitive au procesoare puternice, aproape identice deci sunt utilizate mut mai puține resurse pentru rulare. Al treilea dispozitiv are un procesor mai vechi, care este folosit mai mult pentru rularea jocului, iar ultimul dispozitiv, fiind cel mai slab dintre cele 4 are rezultatele cele mai slabe, utilizând aproape un sfert din puterea de procesare pentru a rula jocul.

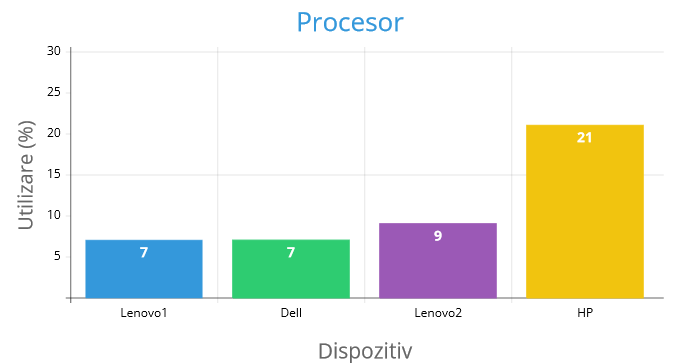


Figura 45. Impactul rulării jocului asupra procesorului

Diferența cea mai mare între sistemele pe care am testat jocul o reprezintă placa video. Aceasta este una dedicată, relativ recentă, foarte puternică pe primul dispozitiv (laptop personal dedicat jocurilor), celelalte fiind fie dedicate randărilor 3D (cum este cea quattro), modele mai vechi (Nvidia Geforce 840m și AMD Radeon). Acest lucru se poate remarca și pe grafic (Figura 46), deși diferența între ultimele doua este surprinzător de mare.

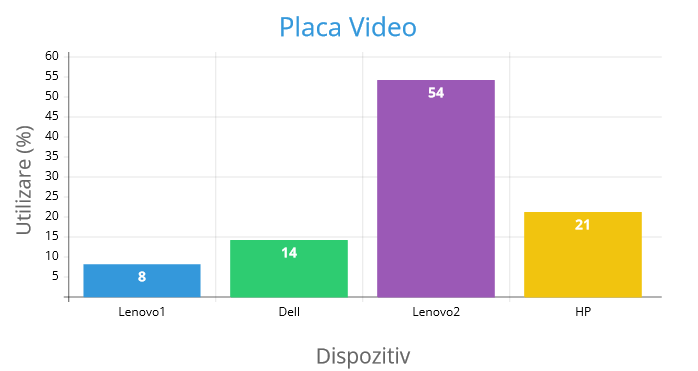


Figura 46. Impactul rulării jocului asupra plăcii video

Dimensiunea totală a jocului este 74MB, ceea ce este foarte mică pentru un joc pe calculator, dar jocul nu este finalizat încă. După adăugarea a mai multor nivele, mai multe resurse grafice pentru caractere, arme și încăperi, mai multe efecte sonore și suport pentru platformele MacOS, iOS și Android aceasta ar crește spre 300-350MB, fiind o dimensiune acceptabilă în majoritatea cazurilor.

Utilizarea memoriei RAM pentru fiecare dispozitiv se poate observa în Figura 47.

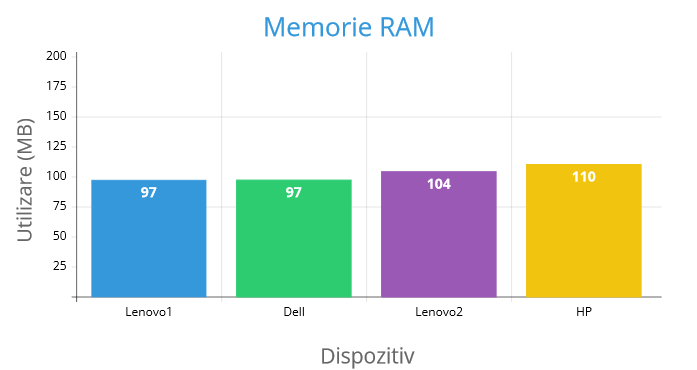


Figura 47. Utilizarea memoriei RAM pentru rularea jocului

Diferențele sunt relativ mici, deoarece dimensiunea jocului este aceeași pentru toate dispozitivele, dar dacă aceste numere sunt analizate din perspectiva memoriei totale atunci diferențele sunt mult mai accentuate. Folosind acest raport se observă o utilizare de 0.3% pentru primele două, 0.6% pentru al treilea și 2.7% pentru ultimul laptop, deoarece are cea mai mică memorie RAM (doar 4 GB).

Impactul jocului asupra temperaturilor procesorului și a placii video este prezentat în Figura 48. Primele două laptopuri, fiind mai noi și mai puternice pot gestiona sarcina fără o diferență mare de temperatură (maxim 2 grade Celsius), dar la celelalte dispozitive diferența este mai pronunțată (în jur de 3-5 grade Celsius).

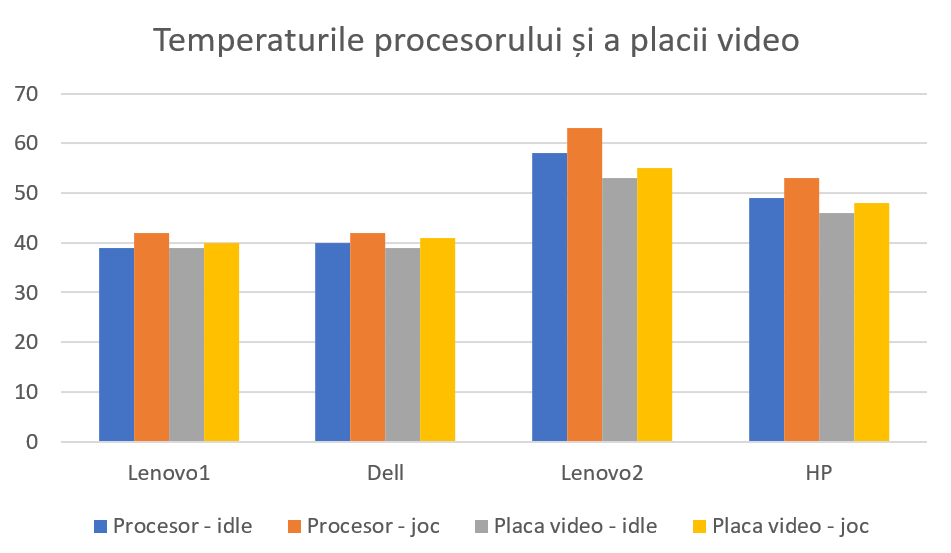


Figura 48. Impactul asupra temperaturilor

În urma testelor, pot spune că jocul nu este foarte solicitant din punctul de vedere al resurselor necesare, putând fi rulat fără probleme pe majoritatea dispozitivelor. Deoarece nu este un joc rapid, care are nevoie de cât mai multe cadre pe secundă pentru o mișcare fluită, impactul pe care îl are configurația sistemului pe care este rulat jocul nu este evidentă la experiența de joc (nu este afectată grafica, sau rapiditatea raspunsurilor la comenzi), ci mai mult prin impactul asupra sistemului prin temperatură mai ridicată/ folosirea mai multor resurse când componentele sunt mai slabe/vechi.

Componentele recomandate de mine pentru a rula jocul fără probleme ar fi :

* Un procesor intel sau amd minim generația a treia,
* Placa video Nvidia generația 8xxm sau AMD Radeon 2xx,
* Minim 4 GB RAM.

Un laptop de gaming sau un sistem configurat pentru aceasta din ultimii 2-3 ani ar rula fără probleme acest joc chiar și după ani de dezvoltare, creșterea complexității prin adaugarea de noi nivele și interacțiuni.

# CONCLUZII

Am pornit în dezvoltarea acesui joc având foarte puține tangențe cu industria de

jocuri video (din perspectiva developer-ului), ci fiind doar consumator. Am învățat foarte multe despre ce constituie un proiect de asemenea magnitudine și rezolvarea de probleme prin folosirea resurselor disponibile pe internet.

Industria jocurilor video a evoluat foarte mult în ultimele decenii, iar ușurința cu care se pot găsi materiale de învățare despre această ramură a programării mi-a stârnit un interes nou, și anume de a crea jocuri simple în timpul liber pentru a învăța și stăpâni procesul. Dorința mea este să progresez încet până creez un joc suficient de interesant din punctul de vedere al experienței de joc și implementarea este cât mai bine realizată încât poate fi lansat pe piață ( Magazin Play/AppStore pentru mobil, și platforma Steam pentru calculator).

Acest joc, în stadiul actual, reprezintă o bază solidă de continuare a dezvoltării. Pentru viitor, vreau să creez mai multe nivele pentru joc și să adaug o poveste jocului, pentru a captiva mai ușor jucătorii. Sistemul de progres poate fi îmbunătățit pentru a cuprinde mai multe nivele de experiență, mai multe arme și armuri cu proprietăți diferite, multiple resurse, cum ar fi mana, stamina și noi resurse de craftare. Sistemul de luptă poate fi dezvoltat pentru a fi mai complex, prin adaugarea de arme cu proiectile pentru atacuri de la distanță (arc, pistol, etc) sau abilități speciale pentru diversitate și crearea de combo-uri.

Jocul poate fi adaptat pentru a oferi suport și pentru platforma mobilă, ceea ce asigură un număr mai mare de jucători, iar dacă se adaugă un mod de joc multiplayer acesta ar oferi competitivitate și captiva interesul jucătorilor de a deveni mai puternici. Dacă jocul are succes și atrage utilizatori, acesta poate fi monetizat prin adaugarea de microtranzacții pentru progres mai rapid prin cumpărarea de resurse cu bani reali, prin adaugarea unui preț de cumpărare pentru platforma Steam, sau prin reclame, folosind serviciul de reclame pus la dispoziție de Unity.

Jocul conține și câteva vulnerabilități la momentul actual, cum ar fi modificarea progresului prin accesarea variabilelor folosind programe stil Cheat Engine, care ar trebui tratată dacă intenționez să îl fac public. Există și câteva bug-uri de producție ce necesită rezolvare, cum ar fi vizibilitatea interfeței HUD la revenirea în meniul principal după moarte, sau accesarea de spații interzise (cum ar fi în pereți) la împingerea inapoi când caracterul suferă daune și este lipit de un zid.

Pentru viitor, îmi doresc să exersez și editarea grafică și audio pentru a fi capabil de crearea resurselor necesare, fără a fi nevoie de cumpărarea acestora (care cresc costul de dezvoltare al jocului) sau folosirea de resurse gratis (precum cele folosite pentru Dungeon Eternal). Acest lucru permite un control mai bun asupra dezvoltării jocurilor, deoarece aș putea să creez singur elementele de grafică, animațiile sau efectele sonore și muzica de fundal de care am nevoie pentru a transforma mai ușor viziunea jocului în realitate.

**8. BIBLIOGRAFIE**

\*\*\*[1] **“Ce înseamnă RPG”** <https://en.wikipedia.org/wiki/Role-playing_game>, accesat la data 20.08.2022;

\*\*\*[2] **„Ce este un NPC?”** <https://www.techopedia.com/definition/1920/non-player-character-npc>, accesat la data 20.08.2022;

\*\*\*[3] **„PixelArt”** <https://en.wikipedia.org/wiki/Pixel_art>, accesat la data 20.08.2022

\*\*\*[4] **“Unity Engine”** <https://en.wikipedia.org/wiki/Unity_(game_engine)>, accesat la data 20.08.2022;

\*\*\*[5] **„Pachetul de sunet folosit”** <https://leohpaz.itch.io/minifantasy-dungeon-sfx-pack>, accesat la 27.08.2022;

\*\*\*[6] **„Atlas-ul DungeonTileset gratuit”** <https://0x72.itch.io/16x16-dungeon-tileset>, accesat la 02.08.2022;

\*\*\*[7] **„World of Warcraft”** <https://ro.wikipedia.org/wiki/World_of_Warcraft>, accesat la data 18.08.2022;

\*\*\*[8] **„Metin2”** <https://en.wikipedia.org/wiki/Metin2> accesat la data 18.09.2022;

\*\*\*[9] **„Albion wiki”** <https://wiki.albiononline.com/wiki/Albion_Online_Wiki>, accesat la data 18.09.2022;

\*\*\*[10] **„Visual Studio 2019”** <https://visualstudio.microsoft.com/vs/>, accesat la data 18.09.2022;

\*\*\*[11] **„Editorul app.diagrams.net”** <https://app.diagrams.net/>, accesat la data 18.09.2022;

\*\*\*[12] **„Editorul app.createlu.com”** <https://app.creately.com>, accesat la data 18.09.2022;

\*\*\*[13] **„PC Gaming to have less growth than Console and mobile in 2022”** <https://twistedvoxel.com/pc-gaming-less-growth-console-mobile-2022/>, accesat la data 21.08.2022;

\*\*\*[14] **„Collider2D.OverlapCollider()”** <https://docs.unity3d.com/ScriptReference/Collider2D.OverlapCollider.html>, accesat la data 21.08.2022;

\*\*\*[15] **„Physics2D.BoxCast”** <https://docs.unity3d.com/ScriptReference/Physics.BoxCast.html>, accesat la data 21.08.2022;

\*\*\*[16] **„Vector3.Distance()”**  <https://docs.unity3d.com/ScriptReference/Vector3.Distance.html>, accesat la data 21.08.2022;

\*\*\*[17] **„GameObject.SendMessage()”** <https://docs.unity3d.com/ScriptReference/GameObject.SendMessage.html>, accesat la data 21.08.2022;

\*\*\*[18] **„Input.GetKeyDown()”** <https://docs.unity3d.com/ScriptReference/Input.GetKeyDown.html>, accesat la data 21.08.2022;

\*\*\*[19] **„Transform.Translate()”** <https://docs.unity3d.com/ScriptReference/Transform.Translate.html>, accesat la data 21.08.2022;

\*\*\*[20] **„PlayerPrefs.SetString()”** <https://docs.unity3d.com/ScriptReference/PlayerPrefs.SetString.html>, accesat la data 29.08.2022;

\*\*\*[21] **„PlayerPrefs.HasKey()”** <https://docs.unity3d.com/ScriptReference/PlayerPrefs.HasKey.html>, accesat la data 29.08.2022;

\*\*\*[22] **„PlayerPrefs.GetString()”** <https://docs.unity3d.com/ScriptReference/PlayerPrefs.GetString.html>, accesat la data 29.08.2022;

\*\*\*[23] **„Clasa MonoBehaviour”** <https://docs.unity3d.com/ScriptReference/MonoBehaviour.html>, accesat la data 29.08.2022;

\*\*\*[24] **„Input.GetAxisRaw()”** <https://docs.unity3d.com/ScriptReference/Input.GetAxisRaw.html> , accesat la data 24.08.2022;

\*\*\*[25] **„Unity AnimatorController”** <https://docs.unity3d.com/Manual/class-AnimatorController.html>, accesat la data 21.08.2022;

\*\*\*[26] **„HWMonitor”** <https://www.cpuid.com/softwares/hwmonitor.html>, accesat la data 02.09.2022;

\*\*\*[27] **„Editorul de diagrame”** <https://charts.livegap.com/app.php>, accesat la data 02.09.2022;

