

MINISTERUL EDUCAȚIEI  
UNIVERSITATEA PETROL – GAZE DIN PLOIEȘTI  
FACULTATEA: LITERE ȘI ȘTIINȚE  
DEPARTAMENTUL: INFORMATICĂ, TEHNOLOGIA INFORMAȚIEI,  
MATEMATICĂ ȘI FIZICĂ  
PROGRAMUL DE STUDII: INFORMATICĂ  
FORMA DE ÎNVĂȚĂMÂNT: IF

# LUCRARE DE LICENȚĂ

Conducător științific: Absolvent:  
conf. dr. ing. Monica Vlădoiu Stroe Marian Eduard

PLOIEȘTI

2024



Anexa 6

**MINISTERUL EDUCAȚIEI  
UNIVERSITATEA PETROL – GAZE DIN PLOIEȘTI  
FACULTATEA DE LITERE ȘI ȘTIINȚE  
DEPARTAMENTUL: INFORMATICĂ, TEHNOLOGIA INFORMAȚIEI,  
MATEMATICĂ ȘI FIZICĂ  
PROGRAMUL DE STUDII: INFORMATICĂ  
FORMA DE ÎNVĂȚĂMÂNT: IF**

Vizat	Aprobat,
Facultatea de Litere și Științe	Director de departament, Lector dr. fiz. Anca Baciu

## **LUCRARE DE LICENȚĂ**

Aplicație software mobilă pentru divertisment

- Jocul de tip roguelike Slime Survival

Conducător științific: conf. dr. ing. Monica Vlădoiu	Absolvent: Stroe Marian Eduard
---	-----------------------------------

**PLOIEȘTI  
2024**



UNIVERSITATEA PETROL - GAZE DIN PLOIEȘTI

FACULTATEA DE LITERE ȘI ȘTIINȚE

DOMENIU: INFORMATICĂ

PROGRAMUL DE STUDII: TEHNOLOGII AVANSATE PENTRU PRELUCRAREA

INFORMAȚIEI

FORMA DE ÎNVĂȚĂMÂNT: IF

Anexa 7

Aprobat, Director de departament, Lector dr Anca Baciu	Declar pe propria răspundere că voi elabora personal lucrarea de disertație și nu voi folosi alte materiale documentare în afara celor prezentate la capitolul „Bibliografie”.  Semnătură student(ă):
<b>DATELE INITIALE PENTRU LUCRARE LICENȚĂ</b>	
Proiectul a fost dat studentului/studentei: Stroe Marian Eduard	
1) Tema lucrării Aplicație software mobilă pentru divertisment – Jocul de tip roguelike Slime Survival	
2) Data eliberării temei: septembrie 2023	
3) Tema a fost primită pentru îndeplinire la data: noiembrie 2023	
4) Termenul pentru predarea lucrării: iulie 2024	
5) Elementele inițiale pentru lucrare: bibliografie și materiale educaționale la discipline de programare/dezvoltare de aplicații mobile, precum și tutoriale de pe YouTube	
6) Enumerarea problemelor care vor fi dezvoltate: elemente introductive, istoria jocurilor mobile și roguelike, concepte și tehnologii utilizate, dezvoltarea aplicației mobile – jocul roguelike Slime Survival, optimizarea jocului, discuții privind dezvoltarea și problemele întâlnite	
7) Enumerarea materialului grafic (acolo unde este cazul):-	
8) Consultații pentru lucrare, cu indicarea părților din proiect care necesită consultarea: periodic	
Conducător științific: conf. dr. ing. Monica Vlădoiu	Student(ă) Stroe Marian Eduard
Semnătura:	Semnătura:

# Cuprins

Capitolul 1. Introducere .....	6
1.1 Obiectivele principale ale lucrării.....	6
1.2 Motivul alegerii temei .....	6
1.3 Obiectivele principale ale jocului .....	6
1.4 Structura lucrării .....	7
Capitolul 2. Industria jocurilor mobile și roguelike.....	9
2.1 Istoria jocurilor video mobile.....	9
2.2 Jocurile de tip roguelike .....	11
2.3 Impactul jocului Vampire Survivors .....	12
Capitolul 3. Concepte și tehnologii utilizate .....	14
3.1 Motorul Unity .....	14
10) Sprite Renderer .....	17
11) Transform .....	17
3.2 Visual Studio .....	18
Capitolul 4. Dezvoltarea aplicației software mobile. Jocul Slime Survival .....	19
4.1 Ideea jocului și obiectivele principale ale proiectului .....	19
4.2 Derularea proiectului .....	20
4.2.1 Crearea proiectului.....	20
4.2.2 Organizarea proiectului și decuparea imaginilor .....	21
4.2.3 Personajul principal.....	23
4.2.4 Harta jocului.....	28
4.2.5 Înamicii personajului principal .....	31
4.2.6 Echipamentul folosit pe parcursul rundei de joc.....	35
4.2.7 Legarea elementelor prin statistici .....	40
4.2.8 Sistemul de Level up și evoluția armelor.....	45
4.2.9 Scenele de joc și interfețele .....	49
Scena Title Screen .....	49
Scena Menu .....	52
Scena Game.....	53
4.3 Testare .....	58
4.3.1 Testarea script-urilor .....	58

4.3.2 Testarea jocului.....	59
4.3.3 Exemple de erori găsite .....	62
4.4 Optimizare.....	64
4.4.1 Optimizare făcută la finalul dezvoltării.....	65
4.4.2 Optimizări făcute pe parcurs .....	68
4.5 Construirea pentru mobil.....	70
4.6 Contribuția personală .....	76
Capitolul 5. Ghidul utilizatorului .....	80
Capitolul 6. Observații, greșeli făcute și lecții învățate.....	85
Capitolul 7. Concluzii și idei pentru dezvoltări ulterioare.....	87
Bibliografie.....	88
Anexă .....	91
A) Cod sursă .....	91
B) Listă de figuri .....	102

# **Capitolul 1. Introducere**

## **1.1 Obiectivele principale ale lucrării**

Obiectivul acestei lucrări de licență este dezvoltarea unei aplicații software mobile pentru divertisment, mai exact un joc mobil de tip *roguelike*, având ca bază inspirațională jocul Vampire Survivors. Scopul acestui tip de joc este de a supraviețui cât mai mult timp împotriva valurilor de inamici. Jucătorul controlează un personaj care se apără atacând automat, acesta încercând să evite obstacolele și să omoare inamicii. Pe parcursul rundei, jucătorul colectează resurse și recompense pentru a deveni mai puternic și pentru a face față valurilor de inamici, aceștia devenind, și ei, mai puternici pe măsură ce trece timpul.

De asemenea, lucrarea de față are ca obiectiv important și elaborarea unui ghid detaliat, referitor la dezvoltarea aplicației software, pe tot parcursul procesului de elaborare, care poate fi folosit de alte persoane interesate să învețe să dezvolte un joc de acest tip.

Un alt obiectiv al acestei lucrări îl reprezintă crearea unui manual al utilizatorului. Scopul, acestuia este de ajuta orice persoană care îl folosește, indiferent de experiența acesteia cu jocurile video, să se bucure, cu ușurință, de toate elementele jocului.

## **1.2 Motivul alegerii temei**

Am ales această temă datorită pasiunii mele pentru jocuri video. Crearea unui joc mobil de tip *roguelike* reprezintă o provocare interesantă care combină creativitatea cu abilitățile tehnice, oferind oportunitatea de a experimenta și de a învăța.

Am ales să dezvolt un joc mobil datorită accesibilității și popularității pe care o au telefoanele mobile. În prezent, dispozitivele mobile sunt platforma preferată de majoritatea utilizatorilor, fiind disponibile pentru o gamă largă de persoane din diverse medii și categorii de vârstă. Această accesibilitate crescută permite ca jocul să ajungă la un public mai larg și să fie ușor de utilizat în orice moment și loc, sporind astfel potențialul de succes și impactul jocului.

## **1.3 Obiectivele principale ale jocului**

Pentru a oferi jucătorului o experiență plăcută, pe parcursul dezvoltării jocului, trebuie implementate următoarele elemente:

- Un personaj principal ce evoluează pe parcursul rundei de joc;
- O hartă ce încurajează mobilitatea;
- O diversitate de inamici ce devin mai puternici pe parcursul jocului;

- O diversitate de arme și echipament pasiv, pentru a oferi o experiență nouă, ori de câte ori este rejucat;
- Un sistem ce îi permite jucătorului să aleagă ce bucată de echipament îl face mai puternic atunci când nivelul lui crește;
- Un sistem de evoluție a armelor;
- O interfață ușor de înțeles și adaptabilă la orice rezoluție;
- Un ecran ce permite jucătorului să aleagă arma de început dorită;
- Un inventar care îi permite jucătorului să vadă echipamentul ales și starea acestuia;
- Posibilitatea de a pune pauză și de a termina runda de joc mai devreme;
- Posibilitatea de a vedea statisticile în timp real.

Pentru a putea implementa diferite echipamente, inamici și personaje, cu ușurință, este necesară implementarea unui sistem de adăugare a acestora. Prin intermediul acestui sistem, dezvoltatorul poate reutiliza diferite resurse (assets) și poate atribui statistică diferite fiecărui element, într-un mod mai eficient.

## **1.4 Structura lucrării**

### **CAPITOLUL 1 – Introducere**

Acest capitol prezintă motivația alegerii temei împreună cu principalele obiective ale lucrării și ale jocului.

### **CAPITOLUL 2 – Industria jocurilor mobile și roguelike**

Acest capitol reprezintă un mic istoric al jocurilor mobile și al celor de tip roguelike, împreună cu impactul jocului *Vampire Survivors*. Capitolul are rolul de a prezenta starea actuală a industriei jocurilor mobile și cum au evoluat acestea, ieșind astfel în evidență potențialul acestora.

### **CAPITOLUL 3 – Concepte și tehnologii utilizate**

În această parte sunt prezentate tehnologiile folosite, împreună cu uneltele folosite și motivația utilizării acestora. De asemenea, sunt explicați termenii folosiți pe parcursul lucrării.

### **CAPITOLUL 4 - Dezvoltarea aplicației software mobile. Jocul Slime Survival**

În acest capitol, sunt descrise procesele de dezvoltare, testare și optimizare prin care a trecut aplicația. Sunt evidențiați pașii necesari construirii aplicației pentru mobil, împreună cu contribuțiile personale.

## **CAPITOLUL 5 Ghidul utilizatorului**

În această parte, este prezentată starea actuală a aplicației, fiind explicate toate componentele disponibile jucătorului, acest lucru servind ca un manual de instrucțiuni. Utilizatorul este trecut prin fiecare ecran, alături de explicațiile necesare pentru a ști ce fac acestea. După aceea, este explicat modul de joc și obiectivul său.

## **CAPITOLUL 6 Observații, greșeli făcute și lecții învățate**

Sunt prezentate câteva observații, împreună cu greșelile făcute pe parcurs și lecțiile învățate din acestea. Este normal ca procesul de creație să fie întâmpinat de greșeli, iar lecțiile învățate din acestea sunt valoroase.

## **CAPITOLUL 7 Concluzii și idei pentru dezvoltări ulterioare**

În ultimul capitol sunt prezentate concluziile și părerile formate în urmă completării aplicației. Este prezentat planul de viitor al aplicației, împreună cu idei legate de alte dezvoltări din domeniul jocurilor video.

# Capitolul 2. Industria jocurilor mobile și roguelike

## 2.1 Istoria jocurilor video mobile

Începuturile jocurilor mobile sunt modeste și se leagă de primele telefoane mobile, care aveau capabilități limitate. Un moment definitiv a fost lansarea jocului "Snake" pe telefoanele Nokia în 1997, ilustrând acest joc. Acest joc simplu, în care jucătorii controlau un șarpe care creștea în lungime pe măsură ce mâncă puncte, a devenit rapid un fenomen global. "Snake" a demonstrat că telefoanele mobile pot fi platforme viabile pentru jocuri și a pus bazele pentru viitoarele inovații [1].



Fig. 1 Jocul Snake pe telefonul Nokia

În primii ani ai acestui secol, telefoanele mobile au devenit din ce în ce mai sofisticate. Apariția telefoanelor cu ecrane color și capabilități Java a permis dezvoltarea unor jocuri mai complexe.

Lansarea iPhone-ului de către Apple în 2007 a reprezentat un punct de cotitură major în istoria jocurilor mobile. Cu un ecran tactil mare și acces la App Store, iPhone-ul a permis dezvoltatorilor să creeze jocuri mult mai sofisticate și interactive. Jocuri precum "Angry Birds" (Figura 2), lansat în 2009 și "Doodle Jump" au devenit extrem de populare, atrăgând milioane de descărcări. Succesul acestor jocuri a demonstrat potențialul uriaș al platformei mobile și a inspirat o nouă generație de dezvoltatori [1].



Fig. 2 Jocul Angry Birds

În această perioadă, piața jocurilor mobile a crescut exponențial. App Store-ul și Google Play Store au oferit o platformă globală pentru distribuția jocurilor, iar dezvoltatorii independenți au găsit oportunități de a-și lansa creațiile [1].

Astăzi, jocurile mobile continuă să evolueze, profitând de progresele tehnologice și de schimbările în comportamentul consumatorilor. Realitatea augmentată (AR) a fost adoptată cu succes de jocuri precum "Pokémon GO" (2016), care a creat o nouă modalitate de a interacționa cu lumea înconjurătoare prin intermediul telefonului mobil. Jocurile multiplayer online, cum ar fi "Fortnite" și "PUBG Mobile," au adus experiențe de joc competitive și cooperative pe platformele mobile, integrându-se cu succes în ecosistemul global al jocurilor video [1].

De asemenea, serviciile de abonament și de streaming pentru jocuri mobile, cum ar fi Apple Arcade și Google Play Pass, oferă acces la biblioteci extinse de jocuri pentru o taxă lunară. Aceste servicii au redefinit modul în care consumatorii accesează și joacă jocuri mobile, eliminând necesitatea achizițiilor individuale și oferind o experiență de joc fără reclame și microtranzacții. În Figura 3, poate fi văzută opțiunea de a cumpăra Google Play Pass și câteva din jocurile pe care le oferă [1].



Fig. 3 Google Play Pass

Astfel, istoria jocurilor mobile este una de inovație continuă și adaptare rapidă, reflectând progresul tehnologic și schimbările culturale care au transformat telefoanele mobile în platforme de jocuri versatile și accesibile. De la jocurile simple ale anilor '90 până la experiențele complexe și imersive de astăzi, jocurile mobile au evoluat semnificativ, devenind o parte integrantă a vieții cotidiene pentru milioane de oameni din întreaga lume [1].

## 2.2 Jocurile de tip roguelike

Aceste jocuri sunt caracterizate prin niveluri generate procedural, moarte permanentă (permadeath) și adesea o tematică de explorare a temnițelor (dungeons). Jucătorul începe runda prin explorarea unei încăperi de tip temniță, ceea ce înseamnă că este blocat într-o cameră cu diferiți inamici. După ce îi omoară, este deblocată o ieșire, fapt urmat de generarea unui nou nivel, acestea fiind diferite de fiecare dată. Sunt generate procedural, în funcție de deciziile făcute de jucător în camera anterioară. Când personajul principal moare runda este încheiată, moartea fiind permanentă [2].

Istoria jocurilor roguelike începe în 1980 cu lansarea jocului "Rogue" (Figura 4), creat de Michael Toy, Glenn Wichman și Ken Arnold. "Rogue" a fost dezvoltat pentru sistemele Unix și a introdus elemente care au devenit emblematic pentru gen: niveluri generate aleatoriu, moarte permanentă și un sistem complex de luptă și gestionare a resurselor. Grafica ASCII simplă, dar captivantă, a permis jucătorilor să-și folosească imaginația pentru a explora temnițele pline de monștri și comori [2].

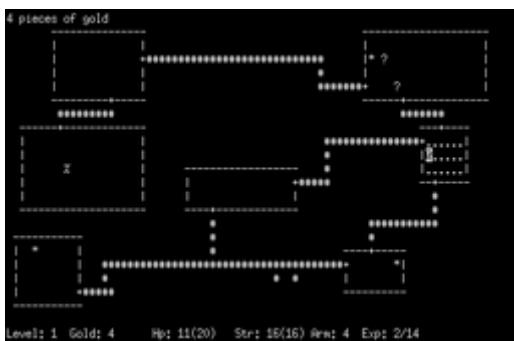


Fig. 4 Jocul Rogue 1980

În anii '90 și 2000, genul roguelike a continuat să evolueze și să se diversifice. Apariția graficii mai avansate și a interfețelor mai prietenoase a permis dezvoltatorilor să creeze jocuri mai accesibile, menținând în același timp complexitatea și rejucabilitatea care definesc genul [2].

Începând cu 2010, jocurile roguelike au cunoscut o adevărată renaștere, devenind din ce în ce mai populare și atrăgând o nouă generație de jucători. Această perioadă a fost marcată de lansarea unor jocuri inovative care au adus genul în atenția mainstream-ului. Jocuri precum "Spelunky" (2008, relansat în 2012), "The Binding of Isaac" (Figura 5), și "FTL: Faster Than Light" (2012) au combinat elementele tradiționale roguelike cu grafica modernă și gameplay-ul accesibil, atrăgând o gamă largă de jucători [2].



Fig. 5 Jocul The Binding of Isaac (2011)

În ultimii ani, jocurile roguelike au continuat să se dezvolte și să experimenteze cu noi idei. Jocuri precum "Hades" (Figura 6) au adus o poveste narativă profundă și personaje bine conturate în genul roguelike, câștigând numeroase premii și recunoaștere critică [2].



Fig. 6 Jocul Hades (2020)

## 2.3 Impactul jocului Vampire Survivors

Vampire Survivors a avut un impact semnificativ asupra industriei jocurilor video, devenind rapid un fenomen cultural și influențând modul în care sunt percepute și dezvoltate jocurile de tip roguelike și bullet hell. Lansat inițial în acces anticipat în decembrie 2021, acest joc indie, dezvoltat de Luca Galante, la studioul independent PONCLE, a captat atenția comunității de gaming și a obținut recunoaștere pe scară largă datorită gameplay-ului său simplu, dar care creează dependență [3].

### Caracteristicile principale

*Gameplay simplu și eficient:* Vampire Survivors se remarcă prin mecanicile sale simple de joc, care se concentrează pe evitarea valurilor de inamici și colectarea de puteri (power-ups). Jucătorii controlă un personaj care se mișcă constant și atacă automat, eliminând nevoia de comenzi complexe și permitând o experiență de joc rapidă și intuitivă [3].

*Progresie și capacitatea de a rejuca:* Jocul este structurat în sesiuni scurte, dar intense, care încurajează rejucarea acestora. Fiecare sesiune oferă o experiență unică datorită

nivelurilor generate procedural și varietății de power-ups disponibile. Aceasta a menținut jucătorii implicați și dornici să revină pentru "doar încă o rundă" [3].

*Estetică retro:* Grafica pixelată și banda sonoră inspirată de jocurile clasice au creat o senzație de nostalgie, atrăgând jucătorii care apreciază stilul retro. În același timp, acest stil a permis dezvoltatorilor să se concentreze pe mecanicile de joc și pe designul nivelurilor [4].

### **Impactul Cultural și Comunitatea**

*Popularitate explozivă:* La scurt timp după lansare, Vampire Survivors a devenit extrem de popular pe platforme de streaming și în comunitățile de jocuri. Simplitatea sa a facilitat accesibilitatea și a permis multor jucători și creatori de conținut să se bucure de joc și să-l promoveze. Aceasta a dus la o creștere rapidă a bazei de jucători și la o vizibilitate crescută în media de gaming [3].

*Influență asupra genului roguelike:* Succesul Vampire Survivors a demonstrat că există o cerere semnificativă pentru jocuri roguelike care sunt ușor de învățat, dar greu de stăpânit. Mulți dezvoltatori indie au început să exploreze mecanici similare, ducând la o revitalizare a genului și la o explozie de noi jocuri care încorporează elemente de bullet hell și roguelike [3].

*Feedback și dezvoltare:* Dezvoltatorul Luca Galante a menținut o relație strânsă cu comunitatea, ascultând feedback-ul jucătorilor și implementând actualizări regulate care au îmbunătățit și extins jocul. Aceasta a creat un sentiment de apartenență și implicare în rândul jucătorilor, consolidând succesul pe termen lung al jocului [4].

### **Impactul pe Termen Lung**

*Inspiratie pentru dezvoltatorii indie:* Vampire Survivors a demonstrat că un joc indie cu mecanici bine gândite și o execuție solidă poate obține un succes uriaș fără un buget mare sau grafică sofisticată. Acest lucru a încurajat mulți dezvoltatori independenti să își urmeze viziunile și să experimenteze cu noi idei [3].

*Expanziune și Evoluție:* Pe măsură ce Vampire Survivors continuă să primească actualizări și conținut nou, este de așteptat ca influența sa să crească. Succesul său poate duce la continuări, spin-off-uri și alte jocuri inspirate de formula sa de succes [4].

În concluzie, Vampire Survivors a avut un impact semnificativ asupra industriei jocurilor video, revitalizând genul roguelike și demonstrând puterea designului simplu și eficient. Succesul său a inspirat o nouă generație de dezvoltatori și a reafirmat importanța comunității și a feedback-ului în dezvoltarea jocurilor [3].

# Capitolul 3. Concepte și tehnologii utilizate

## 3.1 Motorul Unity

Platforma de dezvoltare a jocurilor Unity funcționează prin furnizarea unui mediu de dezvoltare integrat (IDE) și a unui set robust de instrumente și funcții care facilitează crearea și gestionarea jocurilor și aplicațiilor interactive.

Am ales să folosesc acest motor deoarece oferă posibilitatea de a construi jocul pe mai multe platforme (consolă, calculatoare, telefoane mobile), iar versiunea folosită este 2022.3.30f1, aceasta fiind cea mai recentă în momentul începerii dezvoltării aplicației. De asemenea, sunt puse la dispoziție atât uneltele necesare pentru crearea unui joc 2D, cât și a unuia 3D.

În continuare, urmează o prezentare succintă a funcționalității și a principalelor componente ale platformei Unity.

### 1) Limbaj de Programare

Unity utilizează C# ca principal limbaj de programare. Script-urile C# sunt atașate la GameObject-uri sub formă de componente și sunt folosite pentru a defini comportamentele și interacțiunile din joc.

### 2) Scene și GameObjects

În Unity, jocurile sunt construite în Scene, care sunt colecții de obiecte de joc (GameObjects). Aceste obiecte pot fi orice, precum personaje și elemente de mediu. În Figura 7, poate fi observată scena Game, aceasta fiind scena în care are loc acțiunea jocului. De asemenea, sunt prezente două tipuri de obiecte, cele gri fiind obiecte de joc, în timp ce obiectele albastre sunt şabloane reutilizabile [5].

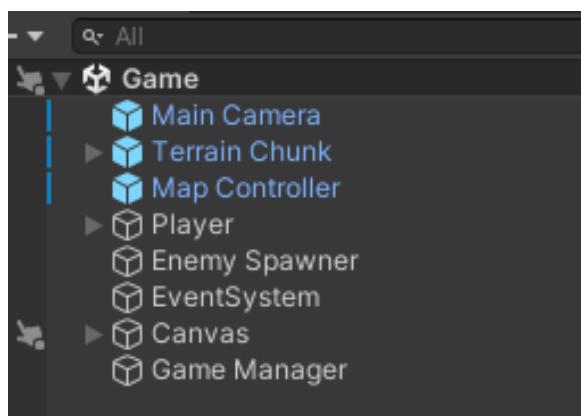


Fig. 7 Scena Game și obiectele sale de joc

### 3) Inspector

Editorul Unity include o fereastră Inspector, unde dezvoltatorii pot edita proprietățile obiectelor de joc, în timp real [6].

### 4) Coliziuni și Corpuri rigide

Componentele de coliziune (Colliders) sunt folosite pentru a defini formele fizice ale obiectelor de joc, iar script-urile pot folosi diferite funcții. În Figura 8, poate fi observat un component 2D de coliziune de tip cerc. Funcția „Is Trigger” afectează modul în care acest obiect interacționează cu altele, mai exact când intră în contact direct cu acestea. Raza acestui component poate fi modificată în elementul „Radius”, zona acoperită fiind schimbată [7].

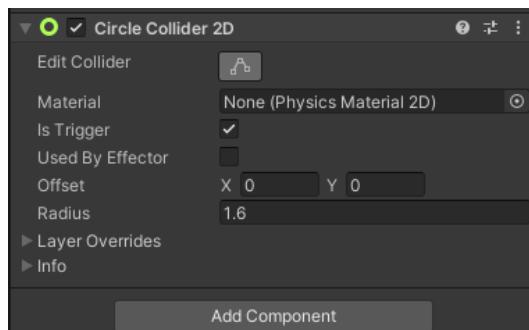


Fig. 8 Circle Collider 2D

Corpul rigid (Rigidbody) este o componentă din Unity care permite simularea fizicii pe un obiect de joc. Adăugarea unui corp rigid la un obiect de joc îi conferă proprietăți fizice, precum masă, gravitație și coliziuni, permitându-i să interacționeze realist cu alte obiecte din joc. În Figura 9, poate fi observat un astfel de corp, acesta fiind de tipul „Kinematic”, tip ce îl face să își păstreze direcția de mers, chiar dacă intră în coliziune cu alte obiecte. Acest tip este folosit pentru arme ce trec prin inamici [8].

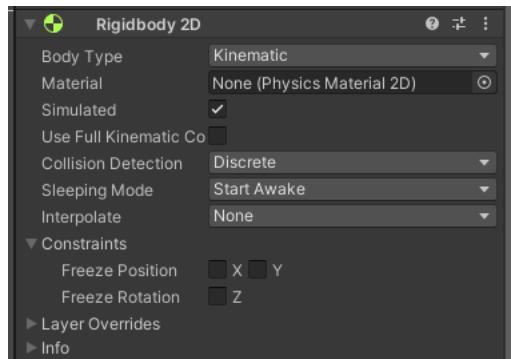


Fig. 9 Rigidbody 2D

### 5) Canvas și UI Elements

Unity include un sistem de elemente UI care permite crearea de interfețe grafice, utilizând obiecte precum butoane, texte și imagini, toate plasate pe un obiect pânză (Canvas). În Figura 10, este afișat un obiect de tip pânză, împreună cu copiii acestuia. Fiecare copil afectează aspectul final, acesta fiind văzut în Figura 11 [9].

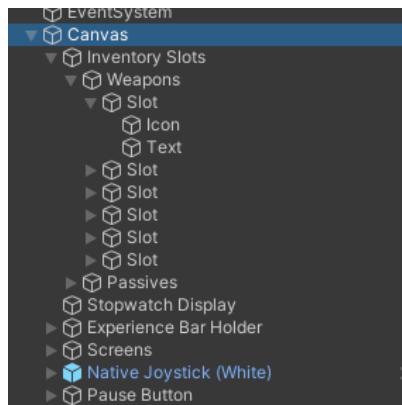


Fig. 10 Obiectul Canvas și obiectele derivate din el (copiii săi)

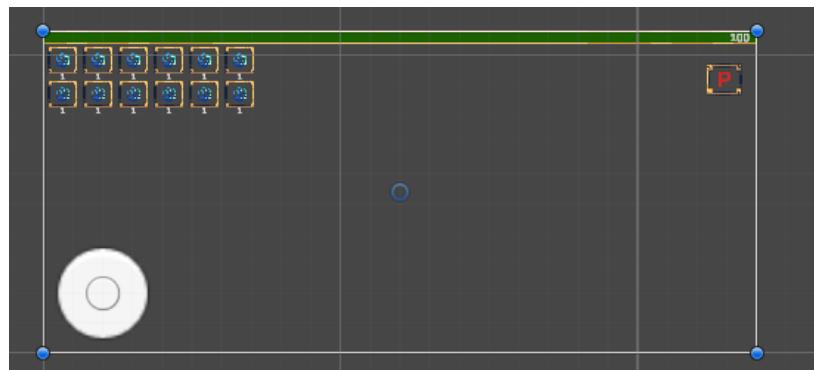


Fig. 11 Aspectul obiectului Canvas în scena de joc

## 6) Event System

Sistemul de evenimente permite gestionarea interacțiunilor utilizatorilor cu elementele UI, cum ar fi click-urile pe butoane și intrările de la tastatură sau joystick virtual [10].

## 7) Asset Store

Magazinul Unity de resurse (Unity Asset Store) oferă o piață unde dezvoltatorii pot cumpăra și vinde resurse precum modele 3D, texturi, sunete, script-uri și instrumente, accelerând dezvoltarea jocurilor.

## 8) Sprite și Spritesheet

Sprite-ul este o imagine 2D folosită în jocuri pentru a reprezenta personaje, obiecte, fundaluri și alte elemente vizuale. Sunt esențiale pentru jocurile 2D, permitând dezvoltatorilor să creeze grafici ușor de gestionat și de animat. În Figura 12, poate fi observată o astfel de imagine 2D, aceasta fiind folosită pentru reprezentarea personajului principal [10].



Fig. 12 Slime Sprite

Spritesheet este o imagine mare care conține o colecție de imagini 2D mai mici, aranjate într-un grilaj. Listele de imagini (Spritesheet) sunt utilizate pentru a eficientiza gestionarea și încărcarea resurselor grafice, reducând numărul de fișiere necesare și

permítând animarea de imagini 2D prin schimbarea cadrului afișat. Figura 13 conține o lista de imagini 2D, câteva dintre acestea fiind folosite pentru crearea animației de mișcare a personajului principal [11].



Fig. 13 Slime Spritesheet

### 9) Tile și Tile Palette

Un tile este o piesă grafică mică folosită pentru a construi niveluri și medii de joc. Jocurile 2D se bazează pe acestea pentru a crea terenuri, drumuri, obstacole și alte elemente ale hărții. Fiecare piesă grafică reprezintă o porțiune mică a graficii jocului și poate fi repetată pentru a crea modele complexe. Această piesă este reprezentată de o singură bucată din Figura 14, fiecare element fiind un fragment diferit.

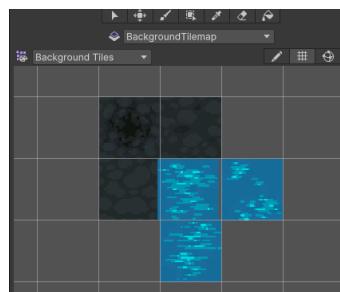


Fig. 14 Tile Pallet-ul folosit pentru crearea fundalului hărții

Paleta de piese grafice (Tile Palette) este o unealtă din Unity care permite dezvoltatorilor să organizeze și să gestioneze tile-urile într-o interfață grafică. Cu această paletă, piesele grafice pot fi plasate pe o hartă sau o grilă pentru a construi niveluri într-un mod eficient .

### 10) Sprite Renderer

Este o componentă care permite afișarea imaginilor pe ecran. Atașarea unui Sprite Renderer la un obiect de joc face posibilă redarea grafică imaginii 2D în scenă, controlând cum arată acesta în joc [12].

### 11) Transform

Este o componentă esențială pentru toate obiectele de joc din Unity. Aceasta definește poziția, rotația și scara unui obiect de joc în spațiu 3D sau 2D al scenei. Transformurile sunt esențiale pentru organizarea și manipularea obiectelor în joc. În Figura 15, se poate observa cum prin intermediul acestuia poate fi modificată poziția (position), rotația (rotation) și mărimea (scale) obiectului [13].

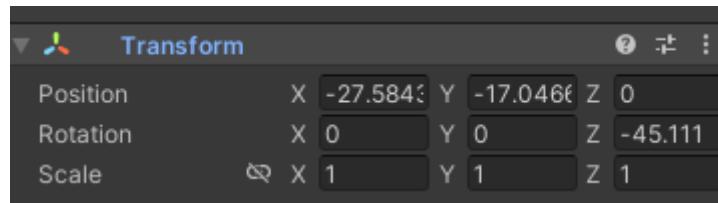


Fig. 15 Componenta Transform

## 12) Animator

Animator este o componentă și o unealtă din Unity care gestionează animările obiectelor de joc. Animatorul folosește un Animator Controller pentru a defini și controla tranzițiile între diferite stări de animație, permitând crearea de secvențe de mișcare fluide și complexe [14].

## 13) Prefab

Un Prefab este un şablon reutilizabil al unui obiect de joc, inclusiv toate componentele și setările sale. řabloanele permit dezvoltatorilor să creeze obiecte care pot fi instantiată de mai multe ori în scenă, economisind timp și asigurând consistență în tot jocul [15].

## 14) Scriptable Object

Obiectul „scriptabil” (ScriptableObject) este un tip de obiect care permite stocarea datelor partajate într-un mod eficient și modular. Spre deosebire de obiectele de joc normale, cele „scriptabile” nu sunt atașate direct la scene. Ele sunt utile pentru stocarea datelor care nu se schimbă frecvent, cum ar fi setările de joc, configurațiile echipamentelor sau datele inamicilor [16].

## 3.2 Visual Studio

Visual Studio este un mediu de dezvoltare integrat (IDE) dezvoltat de Microsoft, care suportă multiple limbaje de programare și tehnologii. Este unul dintre cele mai populare instrumente folosite pentru dezvoltarea aplicațiilor C#, inclusiv pentru dezvoltarea jocurilor în Unity.

Am ales să folosesc Visual Studio pentru editarea script-urilor deoarece:

- Oferă instrumente pentru îmbunătățirea codului sursă (re-factorizing), facilitând renumirea variabilelor, extragerea metodelor și alte modificări de structură a codului;
- Suportă fragmente de cod sursă reutilizabile care pot fi inserate rapid, pentru a accelera procesul de dezvoltare.

## **Capitolul 4. Dezvoltarea aplicației software mobile. Jocul Slime Survival**

### **4.1 Ideea jocului și obiectivele principale ale proiectului**

Jocul este inspirat din *Vampire Survivors*, un joc de tip roguelike în care jucătorul încearcă să supraviețuiască cât mai mult. În Figura 16, se poate vedea un moment din joc, jucătorul fiind reprezentat de personajul din mijloc. Pentru a se apăra de inamicii săi, personajul îi atacă în mod automat, dând albă reprezentând metoda lui de atac. După ce sunt uciși, inamicii lasă în urma lor o recompensă, mai exact un cristal, ce mărește experiența personajului.



Fig. 16 Jocul Vampire Survivors

Slime Survival imită câteva elemente din jocul sursă precum:

- *Supraviețuirea*: scopul principal este să rămâi în viață cât mai mult timp, apărându-te de inamicii ce te înconjoară;
- *Sistemul de level up și echipamentele*: omorând inamici, personajul principal devine mai puternic și își crește nivelul, astfel reușind să aleagă noi echipamente sau le face mai puternice pe cele vechi;
- *Evoluția armelor*: după înndeplinirea unor condiții, armele pot evolua devenind mai puternice.

Crearea unui joc video este un proces îndelungat aşa că, pentru început, stabilirea unor obiective cheie este importantă.

Iată câteva dintre aceste obiective și cerințele care trebuie înndeplinite pentru a le atinge:

- O1 - Implementarea unui sistem ce permite dezvoltatorului să introducă cu ușurință noi elemente precum: personaje, inamici, valuri de inamici și arme:
  - C1 - Crearea a cel puțin un şablon pentru fiecare element. În felul acesta, sunt refolosite aspectul și comportamentul necesare pentru crearea unui element;
  - C2 - Crearea a cel puțin un obiect „scriptabil” pentru fiecare element. Pentru a refolosi statisticile acestora este necesar un obiect „scriptabil” ce reține toate informațiile necesare, inclusiv şablonul elementului.
- O2 - O interfață adaptabilă pentru a putea lansa jocul pe mai multe platforme, în viitor. Fiecare scenă a jocului conține mai multe elemente de interfață precum:
  - C3 - Un joystick virtual cu care jucătorul mișcă personajul;
  - C4 - Un inventar unde jucătorul vede starea actuală a echipamentului;
  - C5 - Un buton prin care jucătorul poate pune pauză;
  - C6 - Informații despre statisticile actuale;
  - C7 - Butoane prin care poate fi aleasă arma de început;
  - C8 - Buton ce sfârșește runda instant.
- O3 - Testarea jocului pentru a rezolva eventualele erori și pentru a nu-l face prea greu sau prea ușor;
- O4 - Optimizarea tuturor elementelor introduse, jocul să consume cât mai puține resurse.

## 4.2 Derularea proiectului

În această secțiune, vor fi detaliate elemente despre organizarea proiectului, implementarea personajului principal, armele acestuia și inamicii săi.

### 4.2.1 Crearea proiectului

Pentru a crea un proiect nou în Unity se folosește aplicația „Unity Hub”, aplicație ce conține toate proiectele., după cum se poate vedea în Figura 17:

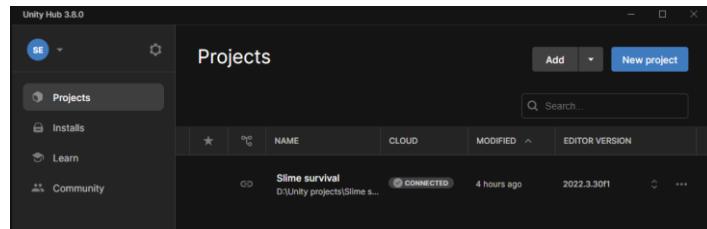


Fig. 17 Unity Hub

Dând click pe „New project” se deschide o nouă interfață. Aici se alege un şablon, în funcție de proiect. După cum se poate vedea în Figura 18, şablonul ales este „Universal 2D”, acesta fiind cel mai potrivit, deoarece, deși jocul este 2D, pentru platforme mobile, un şablon universal face posibilă lansarea sa pe mai multe platforme. Apoi, proiectul este salvat într-un fișier numit „Slime Survival”.

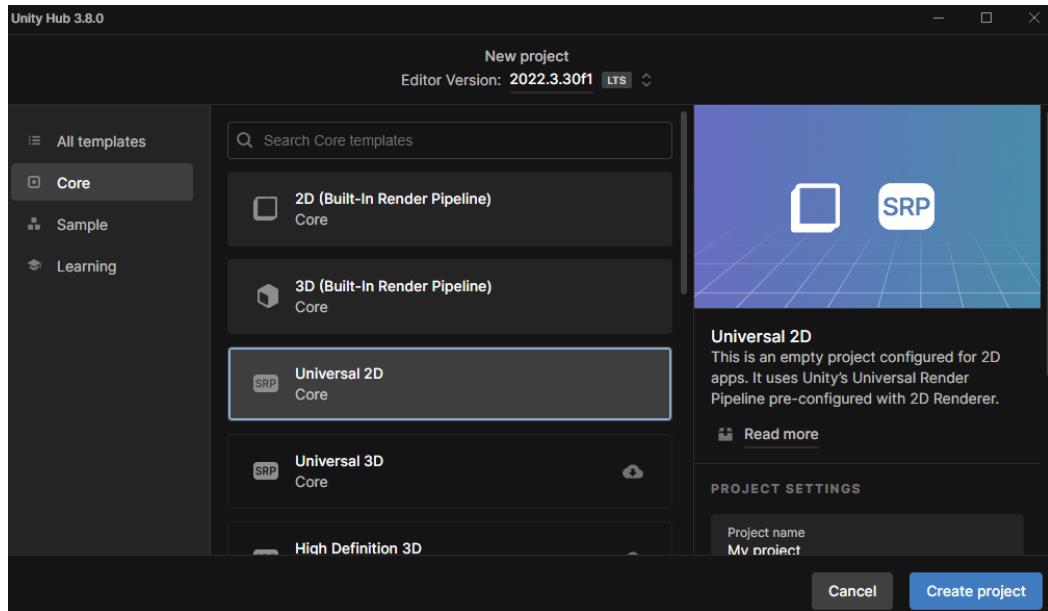


Fig. 18 Alegerea şablonului

#### 4.2.2 Organizarea proiectului și decuparea imaginilor

Cu scopul de a avea o organizare ușoară, au fost create mai multe fișiere. Este important ca acest lucru să fie făcut la începutul proiectului pentru o progresie mai ușoara. Acestea sunt după cum urmează (Figura 19):

- „Art” conține toate imaginile și animațiile folosite în crearea jocului;
- „Plugins” conține plugin-urile folosite;
- În „Prefabs” se află toate obiectele ce nu apar în mod direct pe scena jocului;
- „Scenes” conține toate tipurile de ecran pe care le întâmpină jucătorul;
- „Scriptable Objects” conține obiectele „scriptabile”;
- „Scripts” conține toate script-urile atribuite obiectelor.

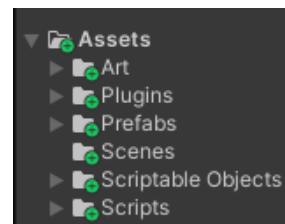


Fig. 19 Fișierele principale folosite pentru organizare

De asemenea, un alt lucru important de fixat la început este mărimea imaginilor folosite. Pe parcursul jocului, o să fie descărcate multe resurse, imaginile venind în

dimensiuni diferite. Din fericire, acest lucru poate fi rezolvat ușor. Pentru a scoate anumite imagini dintr-o listă de imagini, acestea trebuie decupate. În Figura 20, poate fi observată o listă de imagini, mai exact fereastra „Inspector” a acesteia. Înainte de a decupa imaginea trebuie stabilit un element care nu o să fie schimbat pe parcursul proiectului: „Pixels per unit”. Acest element stabilește mărimea imaginii în joc. Pe parcursul dezvoltării jocului o să fie folosită mărimea de 32 de pixeli.

Un alt element important îl reprezintă „Sprite Mode”, care, pentru o listă de imagini, trebuie să fie setat ca „Multiple”. Acest lucru îi transmite editorului faptul că se află mai multe imagini în lista respectivă. După ce au fost făcute modificările, se apasă pe „Apply”, pentru a confirma și pe „Sprite Editor” pentru a decupa imaginile dorite.

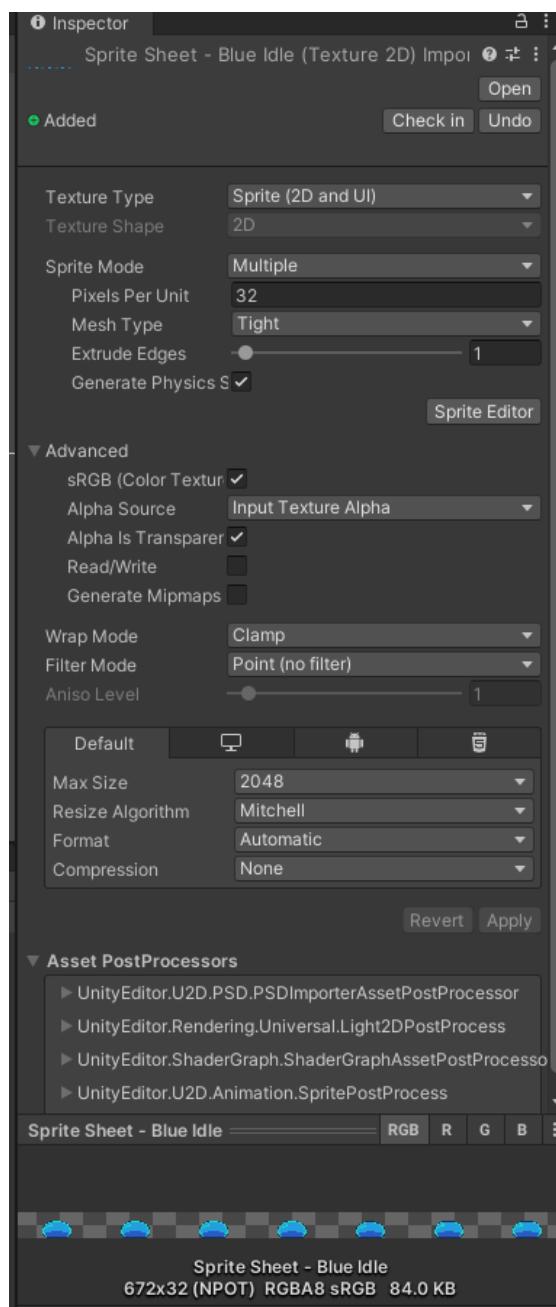


Fig. 20 Inspector listă de imagini

În „Sprite Editor”, mai exact în dreapta sus se află butonul de „Slice”, buton ce decupează imaginile dorite. Acest lucru poate fi făcut în mod automat, după cum se

observă în Figura 21, sau manual dând click și trăgând cu mouse-ul pe ecran, fiind decupată imaginea din interiorul chenarului format.



Fig. 21 Sprite Editor Slice

După cum se poate observa în Figura 22, pentru a salva modificările, se apasă pe butonul „Apply” sau „Revert”, pentru a reveni la starea anterioară, în caz că au fost făcute greșeli.

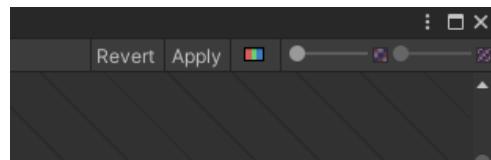


Fig. 22 Sprite Editor Apply

#### 4.2.3 Personajul principal

Ca jucător, scopul tău este să supraviețuiești până la final, iar pentru a face acest lucru posibil este necesar un personaj principal, personaj controlat de utilizator. Sub scena „Game”, scenă în care are loc acțiunea propriu-zisă, în „Hierarchy” se creează un obiect de joc gol. Acesta este personajul și conține diferite componente care îl definsesc aspectul și comportamentul. În Figura 23, pot fi observate toate componentele obiectului, fiecare îndeplinind un rol diferit. Pentru a începe în centrul hărții, componentul „Transform” trebuie resetat. Acest lucru se face dând click pe cele 3 buline din dreapta componentului și apăsând reset (Figura 24).

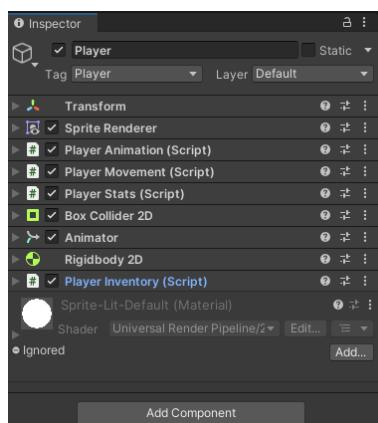


Fig. 23 Fereastra Inspector a obiectului Player

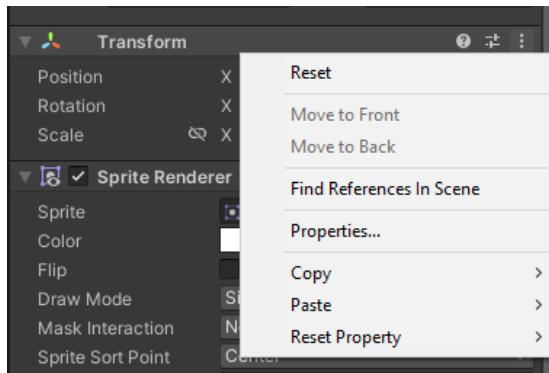


Fig. 24 Resetare componența Transform

După creare, trebuie aleasă o imagine 2D care să îl reprezinte. Imaginea a fost descărcată din „Unity Asset Store”, împreună cu animații. Au fost descărcate 3 variații de culoare, celelalte două reprezentând inamicii. Animațiile descărcate nu oferă și varianta de mers, obiectele sărind când se mișcă. Din acest motiv a fost creată o animație nouă folosind părți din cea originală.

Pentru crearea unei animații se selectează mai multe imagini și se adaugă în scenă. După aceea se salvează în fișierul dorit și este adăugat un controller și un clip de animație, lucru ce poate fi observat în Figura 25.

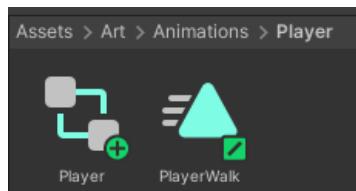


Fig. 25 Controller de clip și animație

Pentru a atribui animația obiectului „Player” sunt necesare un „Sprite Renderer” ce conține imaginea 2D care îi dă aspectul și un „Animator” prin intermediul căruia se atribuie animația dorita.

În Figura 26, poate fi observat componentul care redă imaginea obiectului. Poate fi modificată culoarea (Color) sau răsucită imaginea (Flip). De asemenea, selectarea stratului (Sorting Layer) este importantă, deoarece personajul trebuie să fie mereu pe cel mai înalt strat pentru a putea fi vizualizat. Ordinea în care apare (Order in Layer) ar trebui să fie una cât mai târzie, pentru a fi deasupra altor obiecte care apar în același strat, fiind astfel vizibil în mod clar.

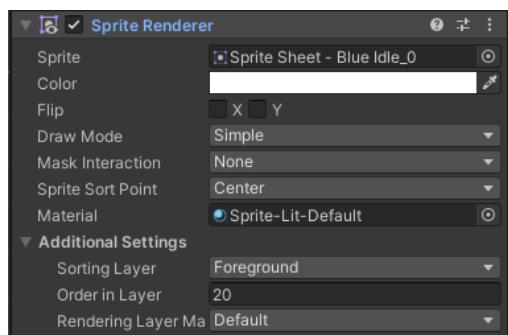


Fig. 26 Sprite renderer pentru Player

Figura 27 reprezintă Animator-ul obiectului. Se poate vedea cum a fost selectată animația dorită (Player), aceasta fiind cea creată mai devreme.

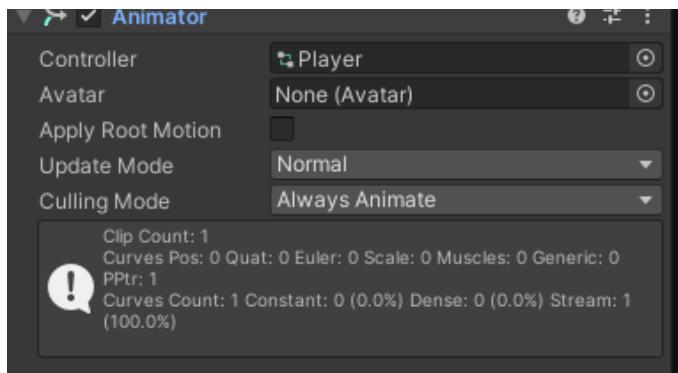


Fig. 27 Fereastra Animator pentru Player

Pentru a ajusta viteza animației sau pentru a adăuga / șterge sprite-uri din aceasta, se folosește fereastra „Animation”, aceasta fiind vizibilă în Figura 28. Un număr mai mare de „Samples” semnifică o animație mai rapidă, iar un număr mic reduce acest lucru. Bulinele albe reprezintă un sprite, iar dând click dreapta pe acesta se poate face modificarea dorită.

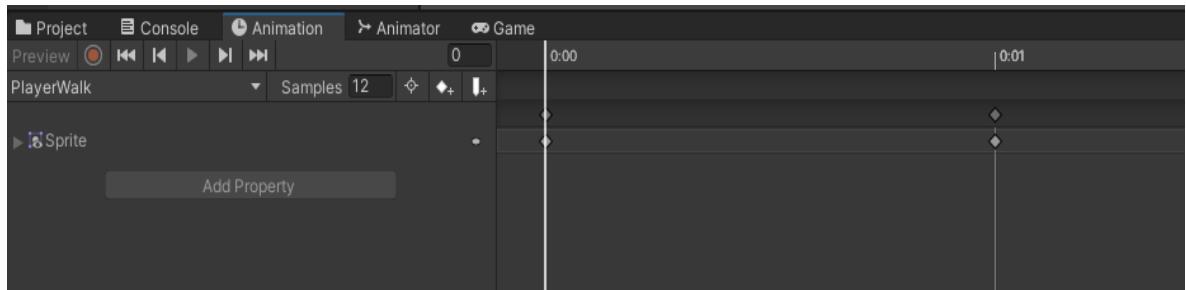


Fig. 28 Fereastra Animation

După ce este terminată animația, urmează ales momentul în care aceasta să pornească și când să se oprească. Pentru acest lucru se folosește fereastra „Animator” (Figura 29).

În această fereastră se editează controller-ul responsabil pentru animație. Momentan sunt importante doar două elemente: „Idle” și „Player Walk”. Animăția din „Idle” este activată prima, atunci când începe jocul și când personajul stă pe loc, iar cea din „Player Walk” este activă doar când obiectul se află în mișcare.

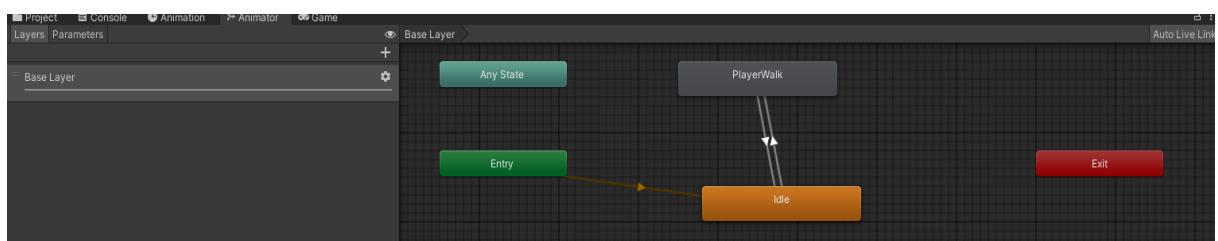


Fig. 29 Fereastra Animator

În Figura 30, pot fi observate elementele „Motion” și „Speed” ce semnifică când are loc animația, respectiv viteza cu care se mișcă personajul.

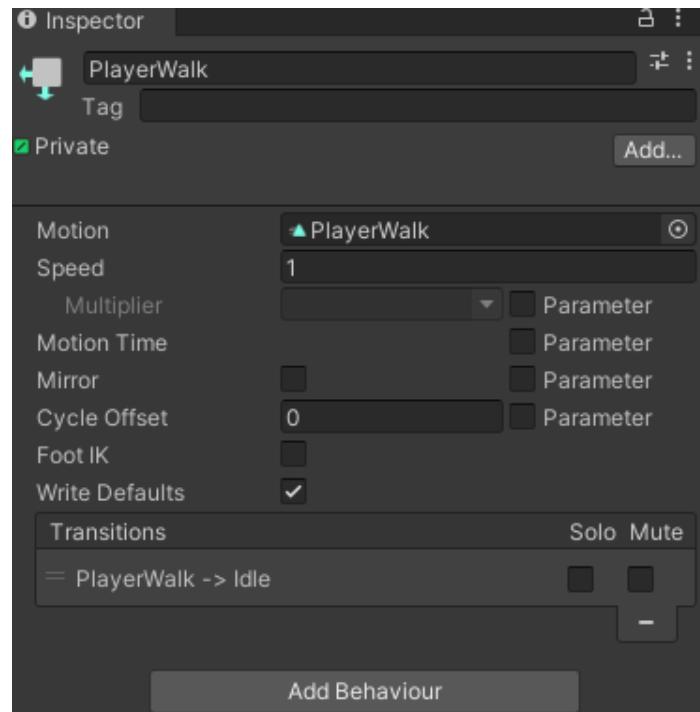


Fig. 30 Acțiunea PlayerWalk

Ca personajul să nu cadă prin hartă este nevoie să fie făcută o modificare asupra gravitației obiectului, dar pentru că utilizatorul privește de deasupra, se setează gravitația proiectului la 0, fiind dezactivată. După cum se observă în Figura 31, acest lucru se face apăsând Edit> Project Settings > Physics2D >General Settings > Gravity.

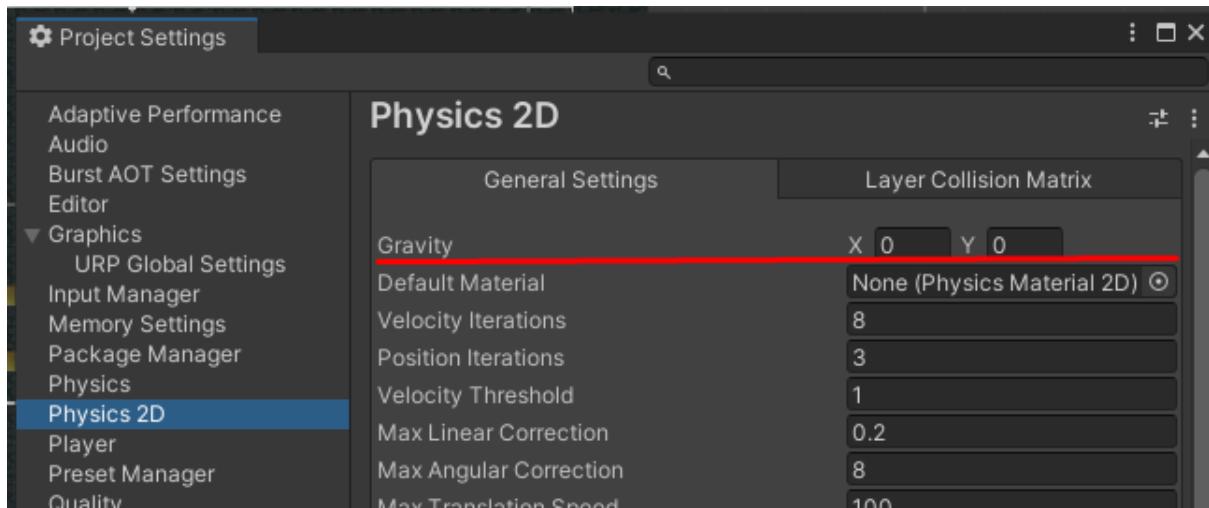


Fig. 31 Setări gravitație

Pentru a face obiectul să se miște este necesar un script de mișcare și un corp rigid 2D. A fost atribuit un script din tutorialul aflat la următoarea adresă:

<https://youtu.be/EIJk5KYzSJM?si=domNNZrhl0JNVoMo>.

Acesta poate fi văzut în următoarea figură (Figura 32):

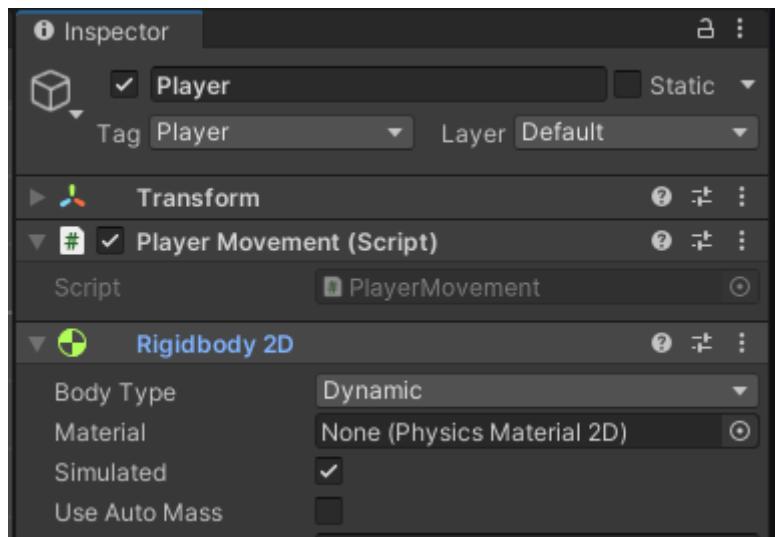


Fig. 32 Partea legată de mișcare din Inspector a obiectului Player

În funcția „ FixedUpdate ” este apelată funcția „ Move ” ce are rolul de a face obiectul să se miște, aceasta fiind observată în codul sursă de mai jos:

```
void Move()
{
    rb.velocity = moveDir * DEFAULT_MOVESPEED *player.Stats.moveSpeed;
}
```

Funcția „ FixedUpdate ” este apelată în mod constant, mai exact de 50 de ori pe secundă. În acest fel, personajul se poate mișca mereu.

Funcția „ Move ” este responsabilă pentru actualizarea vitezei (velocity) corpului rigid (rb) asociat obiectului. Proprietatea „ rb.velocity ” primește un vector care reprezintă direcția și magnitudinea vitezei. Vectorul „ moveDir ” este esențial pentru determinarea direcției de mișcare a jucătorului. „ DEFAULT\_MOVESPEED ” este o constantă (are valoarea de 5 cadre pe secundă) ce definește viteza de mișcare implicită a jucătorului. „ player.Stats.moveSpeed ” este o referință la un atribut specific jucătorului care indică un factor de multiplicare suplimentar pentru viteza de mișcare.

Personajul se mișcă, dar camera trebuie să fie fixată pe el pentru o experiență bună de joc. Acest lucru va face ca acesta să fie mereu în mijlocul ecranului.

Când un proiect nou este creat cu şablonul 2D universal, este atribuit un obiect de joc de tip cameră „ Main Camera ”. Aceasta este responsabil pentru ce vede jucătorul pe ecran cât timp se află în această scenă.

Fixarea camerei poate fi făcută în două moduri, prezentate în continuare:

1. Obiectul „ Main Camera ” poate fi făcut un copil al obiectului „ Player ”, astfel camera se află mereu la aceleasi coordonate față de personaj;
2. Poate fi atribuit un script responsabil cu urmărirea personajului.

Am ales opțiunea 2 deoarece prima limitează posibilitățile camerei. Este un script simplu ce actualizează poziția camerei în funcția „ Update ”, după cum se observă:

```

void Update()
{
    transform.position = target.position + offset ;
}

```

Pozitia camerei este inlocuita cu pozitia obiectului + distanta aleasa in „Inspector”.

Dupa cum se poate vedea in Figura 33, script-ul a fost atribuit obiectului „Main Camera”, iar inta este personajul, camera fiind deasupra acestuia ( $z = -10$ ).

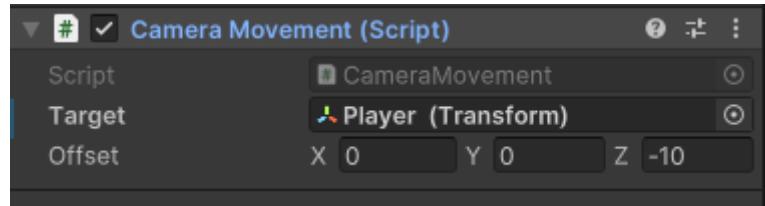


Fig. 33 Script-ul Camera Movement

#### 4.2.4 Harta jocului

Acum ca personajul se poate misca, este nevoie de o hartă pe care să poată face acest lucru. Pentru a oferi o mobilitate mai mare, harta o să fie generată „la infinit”. Indiferent de cât de mult merge jucătorul într-o direcție, o să fie generate părți noi de hartă.

Se începe prin a construi o bucată de hartă ce o să fie generată de câte ori este nevoie. Din magazinul Unity, a fost descărcată o listă de imagini ce conțin texturi pentru construirea unei hărți. Se adaugă imaginile dorite în „Tile Palette” (Figura 15) și se desenează fundalul.

Rezultatul final poate fi vizualizat în Figura 34:

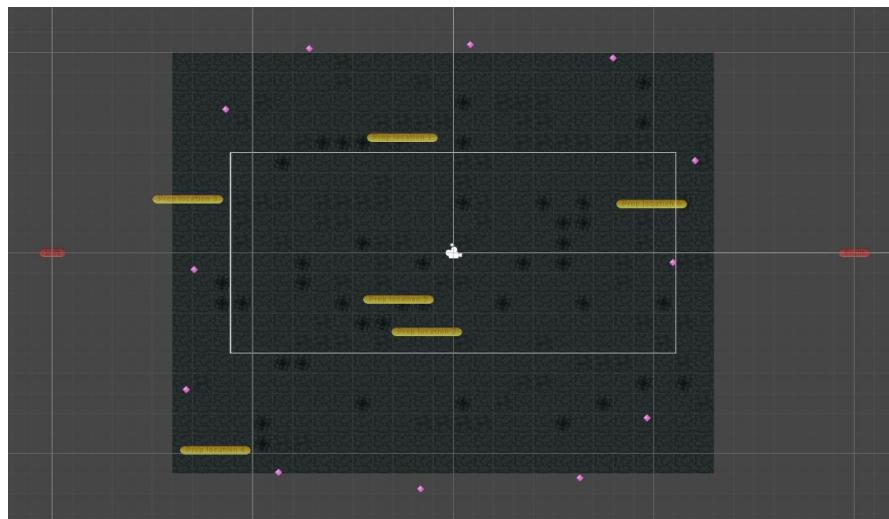


Fig. 34 Fundalul unei bucăți de hartă

Pentru a putea genera hartă, obstacole și inamici, sunt necesare 3 tipuri de obiect de joc folosite ca punct de referință. Bucata făcută este salvată ca şablon, clonată și îi este modificat puțin aspectul pentru a diversifica harta.

Acestea sunt punctele de referință prezente mai sus:

- *Punctele galbene*: reprezintă locurile unde pot apărea obstacole;
- *Punctele roșii*: sunt reperele folosite pentru a genera alte bucăți de hartă;
- *Punctele roz*: reprezintă locul unde poate apărea un inamic.

Pentru a genera obstacole este necesar un script ce alege aleator din opțiunile introduse în Inspector. Acesta a fost luat din tutorialul aflat la adresa :

<https://youtu.be/QN8dm0RD3mY?si=cFcuL6xtCrTaz5L7> .

Acest script se bazează pe funcția „SpawnProps” care alege aleator un obstacol și îl generează pe harta jocului într-un punct galben (Figura 34). Unele dintre acestea au un „Polygon Collider 2D” pentru a nu permite jucătorului să treacă peste ele (Figura 35). Funcția poate fi observată în caseta următoare:

```
void SpawnProps()
{
    foreach (GameObject sp in propsSpawnPoints)
    {
        int rand = Random.Range(0, propsPrefabs.Count);
        GameObject prop = Instantiate(propsPrefabs[rand], sp.transform.position,
Quaternion.identity);
        prop.transform.parent = sp.transform;
    }
}
```

Aceasta este apelată în funcția Start, care este la rândul ei apelată la începutul jocului, o singură dată, înainte ca primul cadru să înceapă.

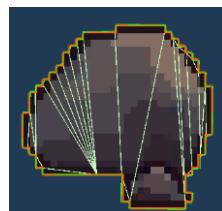


Fig. 35 Obstacol cu un Polygon Collider

În Figura 36, poate fi vizualizat script-ul „Props Spawn Points” conținând punctele de referință pentru obstacole, iar „Props Prefabs” conține toate obstacolele care pot fi generate.

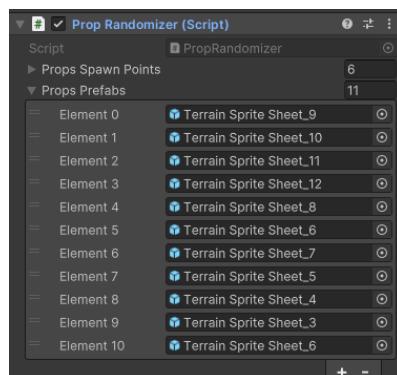


Fig. 36 Script-ul Prop Randomizer

Pentru generarea bucăților de hartă este creat un obiect de joc numit „Map Controller”, ce are un script cu același nume, script obținut din tutorialul aflat pe URL-ul următor:

[https://youtu.be/QN8dm0RD3mY?si=H7IQmd0bq\\_4IG0lf](https://youtu.be/QN8dm0RD3mY?si=H7IQmd0bq_4IG0lf)

Se poate observa în Figura 37, cum sunt introduse bucățile de hartă, personajul în jurul căruia se generează harta, stratul în care este generată și optimizarea acestora.

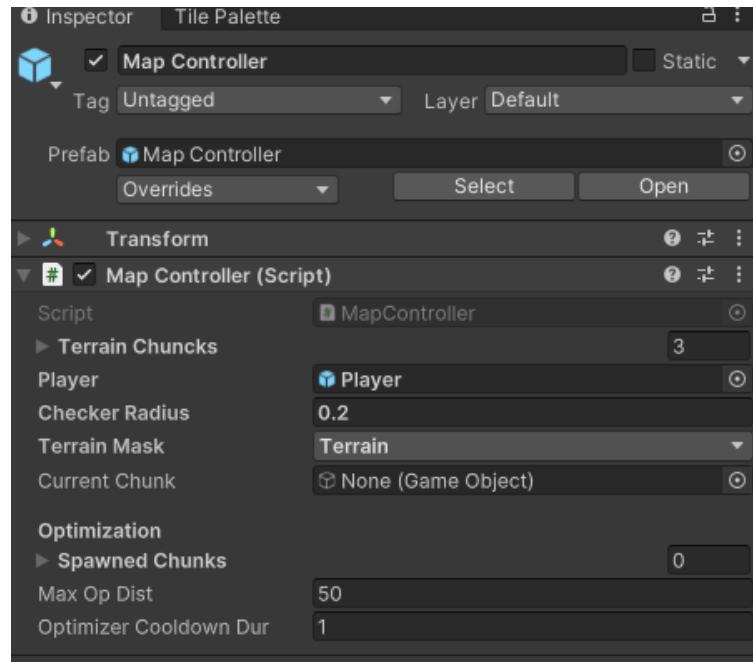


Fig. 37 Obiectul Map Controller

Pentru că harta este generată pe un anumit strat, este posibil să ascundă diferite obiecte aşa că fiecărui obiect îi este atribuit un strat, în funcție de rolul său. Acest aspect este evidențiat în Figura 38.

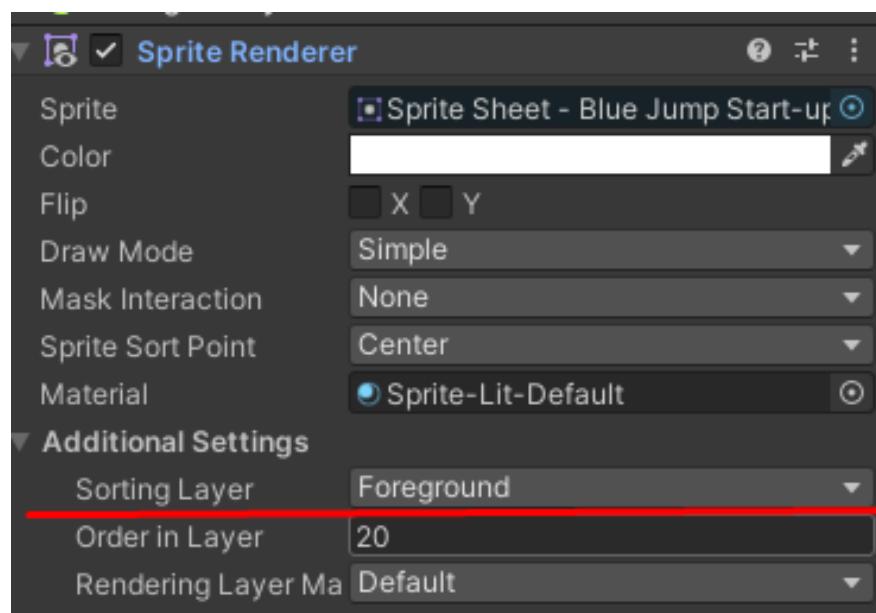


Fig. 38 Stratul pentru obiecte

Principalele funcții ale script-ului sunt:

- „ChunkChecker”, care verifică direcția mișării jucătorului și determină dacă sunt necesare fragmente de teren noi în direcția respectivă;
- „SpawnChunk” (căsuță de text următoare), generează un nou fragment de teren la poziția specificată și îl adaugă în lista de fragmente generate;
- „ChunkOptimizer”, care verifică distanța dintre jucător și fiecare fragment de teren generat. Dacă distanța depășește „maxOpDist”, fragmentul de teren este dezactivat pentru a economisi resurse.

```
void SpawnChunk(Vector3 spawnPosition)
{
    int rand = Random.Range(0, terrainChuncks.Count);
    latestChunk=Instantiate(terrainChuncks[rand], spawnPosition,
    Quaternion.identity);
    spawnedChunks.Add(latestChunk);
}
```

Momentan, bucățile de hartă sunt generate și salvate „la infinit” lucru ce poate ajunge să consume multe resurse. De evitarea acestui lucru se ocupă funcția „ChunkOptimizer”, ea afișând doar câteva bucăți, celelalte fiind reafisate când personajul se întoarce în locația respectivă.

În Figura 39, se observă cum arată harta inițială, personajul fiind punctul albastru, acesta mergând spre stânga.

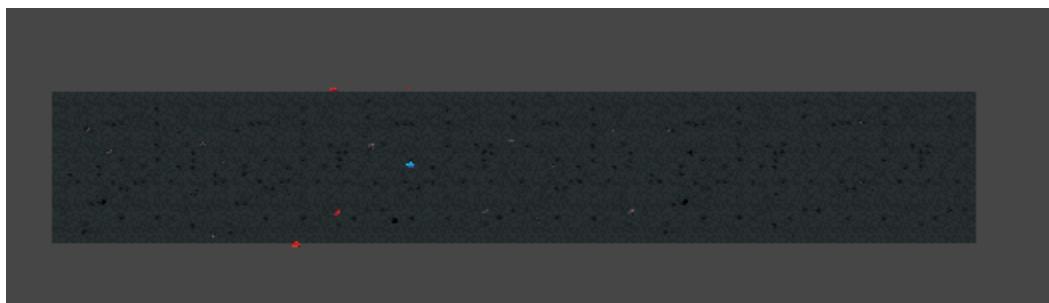


Fig. 39 Harta inițială

Dispariția unui fragment de hartă este remarcată în Figura 40. Acest lucru se întâmplă deoarece personajul a parcurs o distanță suficient de mare.

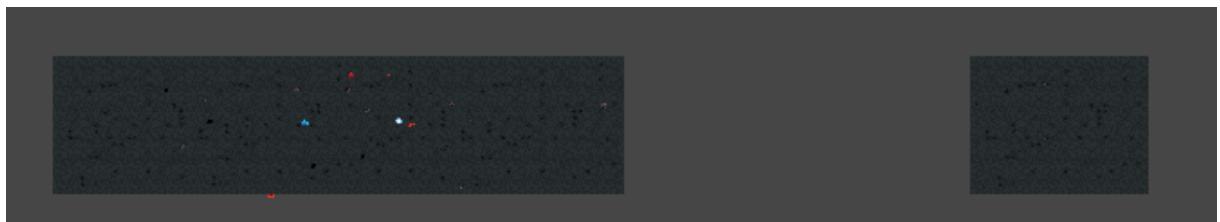


Fig. 40 Harta la o distanță mare față de punctul inițial

#### 4.2.5 Inamicii personajului principal

Următorul pas este generarea inamicilor, aceștia având rolul de a omorî jucătorul.

Implementarea acestora se face în mod similar cu cea a personajului principal, fiind descărcate resurse ce se ocupă de aparență și animație. În Figura 41, sunt toți inamicii ce pot apărea în joc.



Fig. 41 Inamici

De generarea inamicilor se ocupă obiectul „Enemy Spawner” ce are un script cu același nume (Figura 42). Acest script a fost făcut urmând tutorialul aflat la adresa:

[https://youtu.be/h2cg4ucDuWw?si=PvJfu\\_emFnrYc0kd](https://youtu.be/h2cg4ucDuWw?si=PvJfu_emFnrYc0kd).

Prin intermediul lui sunt aleși inamicii și sunt organizați în valuri. Fiecare val generează inamici la un interval introdus în Inspector, următorul val începând când se termină cel precedent.

Pentru a nu consuma prea multe resurse și pentru a face jocul mai ușor, numărul de inamici este limitat la 100, generarea lor fiind oprită până când un inamic moare.

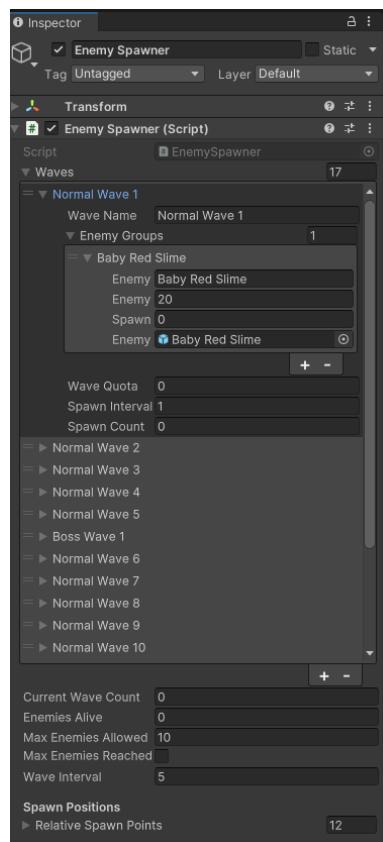


Fig. 42 Enemy Spawner

Functiile folosite pentru a realiza acest lucru sunt urmatoarele:

1. Start()

- Inițializează „player” găsind transform-ul jucătorului și află poziția acestuia;
- Inițializează „CalculateWaveQuota()” .

2. Update()

- Verifică dacă valul curent s-a terminat și începe urmatorul val dacă este cazul;
- Actualizează „spawnTimer” și verifică dacă este timpul să apară urmatorul inamic.

3. BeginNextWave()

- Corutină care așteaptă „waveInterval” secunde înainte de a începe urmatorul val;
- Dacă există mai multe valuri, incrementează „currentWaveCount” și recalculează cota valului.

4. CalculateWaveQuota()

- Calculează și setează cota (numărul total de inamici) pentru valul curent.

5. SpawnEnemies()

- Verifică dacă mai sunt inamici de generat în valul curent și dacă nu s-a atins numărul maxim de inamici activi;
- Generează inamici la poziții aleatorii din „relativeSpawnPoints” (punctele roz din Figura 32) față de jucător;
- Actualizează numărul de inamici generați și de inamici activi;
- Dacă s-a atins numărul maxim de inamici activi, setează „maxEnemiesReached” ca fiind adevărat.

6. OnEnemyKilled()

- Decrementează „enemiesAlive” când un inamic este ucis.
- Resetează „maxEnemiesReached” dacă numărul de inamici activi este sub limita maximă.

Şabloanele inamicilor, ce pot fi văzute în Figura 43, au adăugate elemente de tip „Box Collider 2D” pentru a interacționa cu personajul principal. Prin intermediul acestuia ei îl pot împinge și omorâ. Script-ul „Enemies Movement” se ocupă de mișcarea obiectelor de tip inamic, iar „Drop Rate Manager” se ocupă de asigurarea recompensei oferită când un inamic este ucis de către jucător. Acestea au fost luate din tutorialul ce se găsește la adresa:

<https://youtu.be/qREiQ5vSAn?si=Yk0QTTNQCjU9Tb3P> .

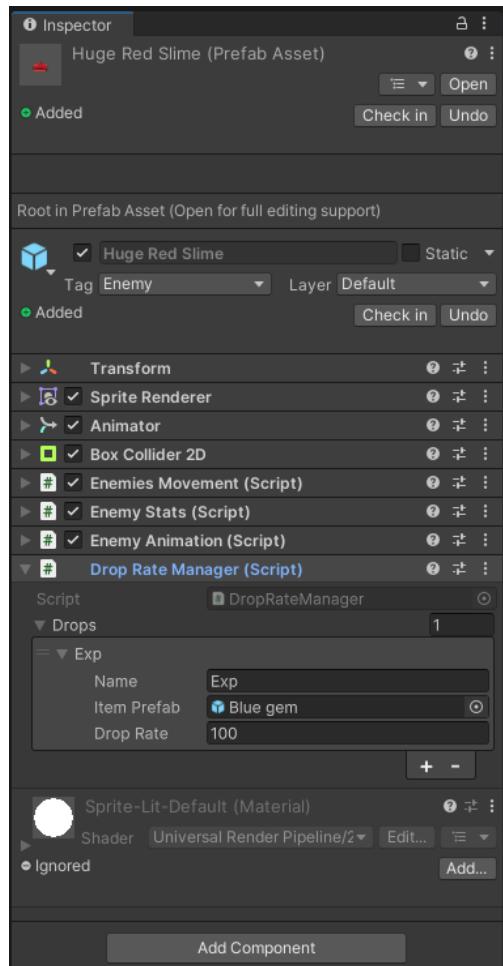


Fig. 43 Fereastra Inspector pentru un şablon de inamic

Funcţia „OnDestroy” este apelată atunci când un obiect este distrus, în cazul de faţă când moare un inamic. În acel moment alege aleator o recompensă din lista introdusă în Inspector şi o generează pe scena de joc, urmând să fie ridicată de jucător.

În următorul Text Box, poate fi observată o parte din funcţia „OnDestroy”, parte ce verifică dacă poate fi oferită o recompensă şi o generează.

```
if(possibleDrops.Count > 0)
{
    Drops drops = possibleDrops[UnityEngine.Random.Range(0, possibleDrops.Count)];
    Instantiate(drops.itemPrefab, transform.position, Quaternion.identity);
}
```

Fiecare recompensă are un element de coliziune, element care atunci când intră în contact cu personajul activează script-ul „Pickup”. El este în şablonul fiecărei recompense şi are rolul de a afecta statisticile jucătorului atunci când recompensa este ridicată. Un asemenea şablon poate fi observat în Figura 44:

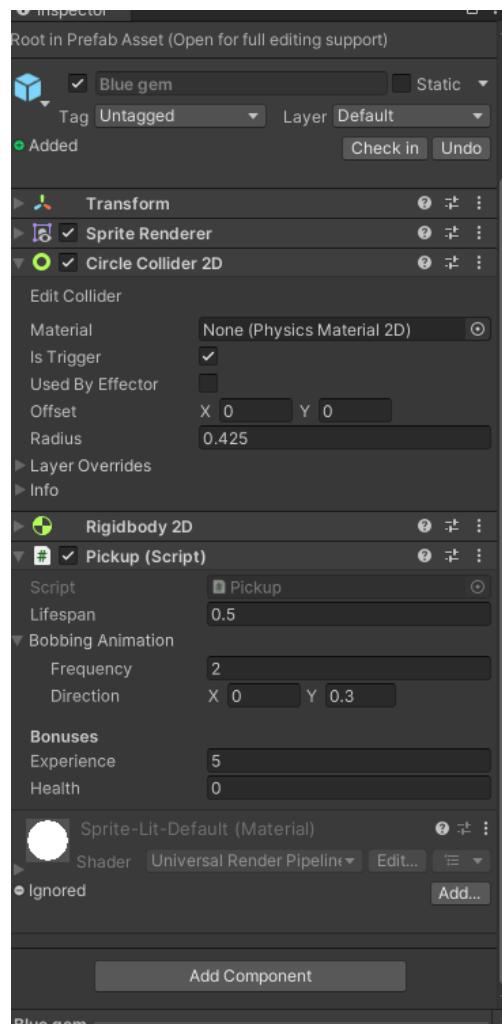


Fig. 44 Fereastra Inspector pentru o recompensă de experiență

Pentru a face jocul să prindă viață, recompensele au o animație de mișcare în sus și în jos. Această animație a fost preluată din tutorialul găsit la adresa:

[https://youtu.be/iiviXsCjHfc?si=D4MftMVg6\\_oPB70w](https://youtu.be/iiviXsCjHfc?si=D4MftMVg6_oPB70w).

Inamicul poate oferi trei tipuri de recompense, după cum urmează:

1. Experiență: cu suficientă experiență personajul trece la nivelul următor și devine mai puternic;
2. Cufăr: cufărul este folosit pentru a evoluă arma;
3. Regenerare viață: personajul își poate regenera viață când ucide un inamic.

#### 4.2.6 Echipamentul folosit pe parcursul rundei de joc

Pe parcursul rundei, jucătorul poate alege între diferite echipamente:

- Active: armele cu care poate ataca;
- Pasive: echipament ce are rolul de a îmbunătăți statisticile personajului principal ( precum viață și armura) și ale armelor( precum puterea de atac, respectiv viteza de atac).

Sistemul de implementare a echipamentului folosește mai multe script-uri și clase abstracte: [https://youtu.be/bbktg7OPyFU?si=Gvcl4myP4\\_CD-BWr](https://youtu.be/bbktg7OPyFU?si=Gvcl4myP4_CD-BWr).

Echipamentul crește în nivel pe parcursul jocului și devine mai puternic. Fiecare superclasă conține date ce sunt refolosite în clasa copil. Diagrama din Figura 45 evidențiază ierarhia dintre clasele folosite în crearea echipamentului.

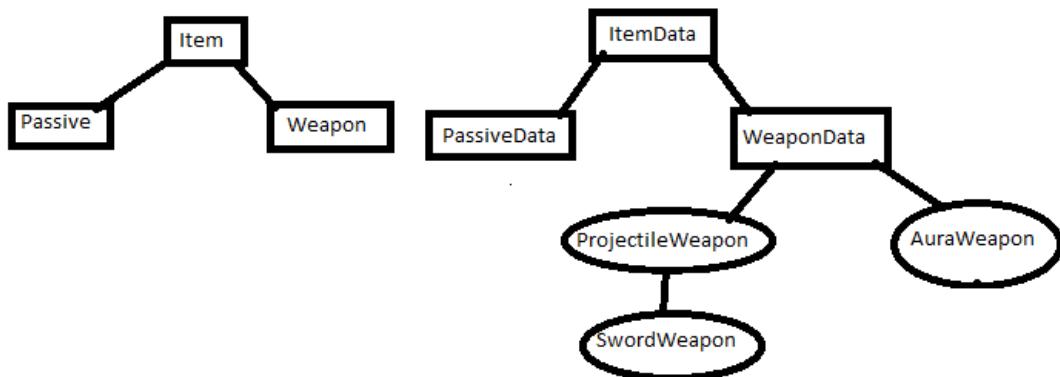


Fig. 45 Diagrama ierarhiei claselor de echipament

Clasa Item este o clasă abstractă ce definește structura de bază pentru orice obiect din joc. Aceasta include atribută și metode esențiale precum:

Atributele acestei clase sunt cele de mai jos:

- currentLevel și maxLevel: Nivelul curent și nivelul maxim al obiectului;
- evolutionData: Date despre evoluțiile posibile ale obiectului;
- inventory: Referință către inventarul jucătorului;
- owner: Referință către caracteristicile jucătorului care detine obiectul;
- data: Date despre obiect, de tip ItemData.

Metodele clasei Item sunt următoarele:

- Initialise(ItemData data): Inițializează obiectul cu datele primite;
- CanLevelUp(): Verifică dacă obiectul poate să își crească nivelul;
- DoLevelUp(): Crește nivelul obiectului;

Clasa PassiveData extinde clasa ItemData și este folosită pentru a defini datele specifice pentru obiectele pasive din joc.

Atributele acestei clase sunt cele de mai jos:

- baseStats: Statistici de bază pentru obiectul pasiv;
- growth: Creșterea statisticilor pentru fiecare nivel.

Metoda clasei PassiveData este următoarea:

- `GetLevelData(int level)`: Returnează datele pentru un anumit nivel al obiectului.

Clasa `Weapon` extinde clasa `Item` și definește comportamentul specific pentru arme.

Atributele acestei clase sunt cele de mai jos:

- `Stats`: O clasă internă care extinde `LevelData` și definește statisticile specifice pentru arme;
- `currentStats`: Statistici curente ale armei;
- `currentCooldown`: Timpul de răcire curent pentru atac;
- `movement`: Referință către mișcarea jucătorului.

Metodele clasei `Weapon` sunt următoarele:

- `Initialise(WeaponData data)`: Inițializează arma cu datele primite;
- `Update()`: Actualizează arma la fiecare cadru;
- `DoLevelUp()`: Crește nivelul armei și actualizează statisticile;
- `CanAttack()`: Verifică dacă arma poate ataca;
- `Attack(int attackCount = 1)`: Atacă dacăarma este pregătită;
- `GetDamage()`: Returnează daunele totale ale armei;
- `GetArea()`: Returnează aria de efect a armei;
- `GetStats()`: Returnează statisticile curente ale armei;
- `ActivateCooldown(bool strict = false)`: Activează perioada de răcire pentru atac.

Clasa `ItemData` este o clasă de bază pentru datele obiectelor, extinsă de `PassiveData` și `WeaponData`. Aceasta definește structura de bază pentru datele de nivel și evoluții ale obiectelor.

Clasa `WeaponData` extinde clasa `ItemData` și este folosită pentru a defini datele specifice pentru arme.

Atributele acestei clase sunt cele de mai jos:

- `behaviour`: Comportamentul armei;
- `baseStats`: Statistici de bază pentru armă;
- `linearGrowth`: Creșterea liniară a statisticilor pentru fiecare nivel;
- `randomGrowth`: Creșterea aleatorie a statisticilor pentru fiecare nivel.

Metoda clasei `WeaponData` este următoarea:

- `GetLevelData(int level)`: Returnează datele pentru un anumit nivel al armei.

Clasa Passive extinde clasa Item și definește comportamentul specific pentru obiectele pasive.

Preia datele și comportamentele de bază din clasa Item și poate adăuga sau modifica specificații pentru obiectele pasive.

AuraWeapon, ProjectileWeapon și SwordWeapon sunt clase derivate din Weapon și adaugă funcționalități specifice pentru armele care trag proiectile și armele de tip sabie.

Crearea unui echipament se face în mai mulți pași, care vor fi prezentate în continuare, aici.

Procesul de creare unei arme este următorul: mai întâi este creat obiectul „scriptabil” al armei intrând în fișierul dedicat acestor obiecte și apăsând click dreapta > Create > Slime Survival Data > Weapon Data. Acest proces este reprezentat de Figura 46.

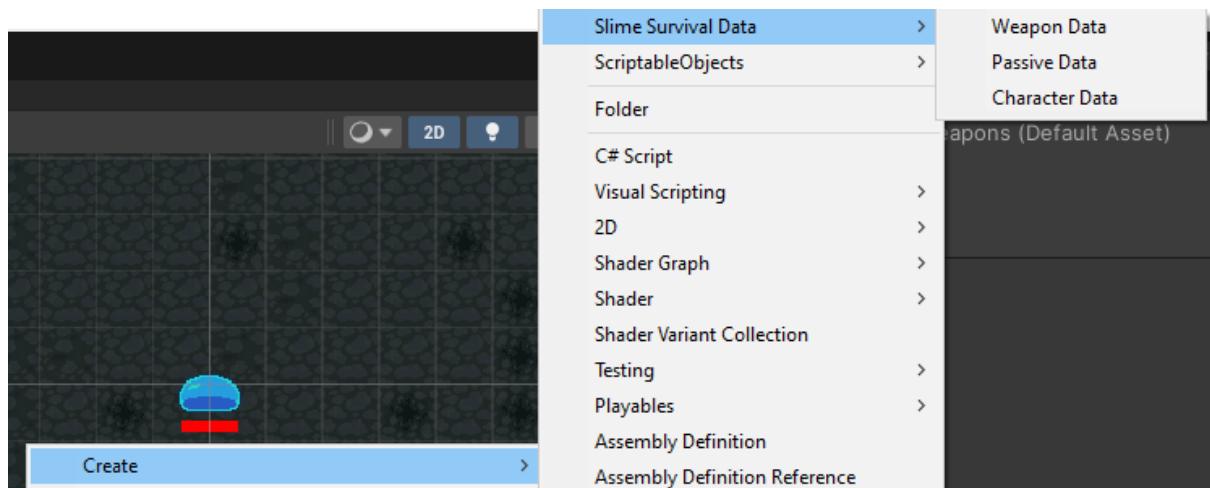


Fig. 46 Crearea unei arme

După aceea se selectează comportamentul armei. Acest lucru se referă la metoda de atac folosită de armă, după cum se poate vedea în următoarea figură (Figura 47).

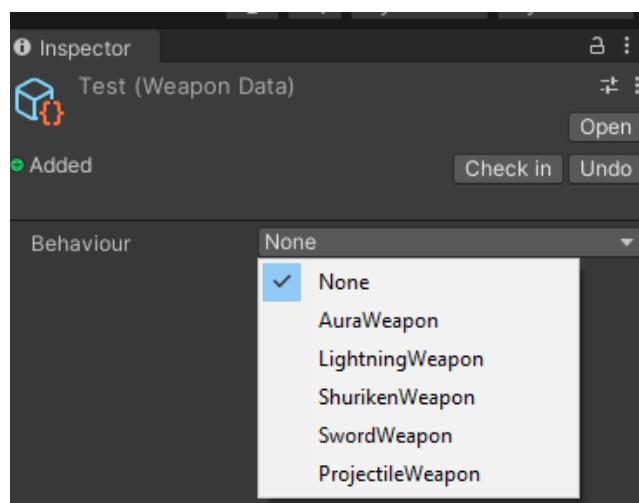


Fig. 47 Alegerea Comportamentului unei arme

Este introdus şablonul armei, iconița și statisticile de bază împreună cu cele obținute atunci când arma crește în nivel.

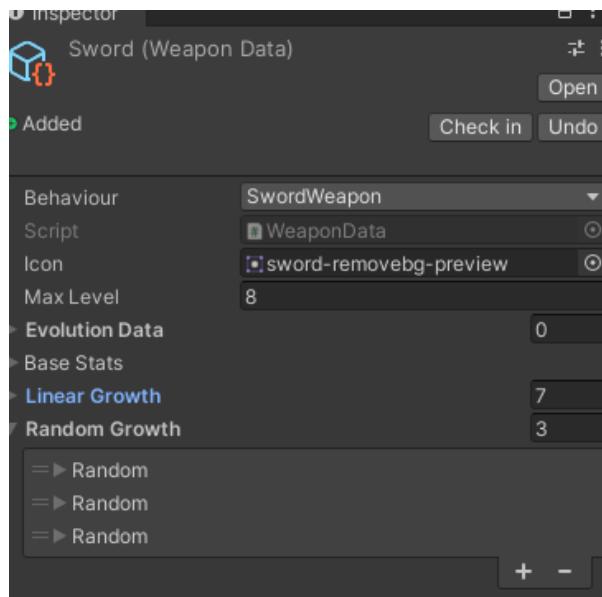


Fig. 48 Fereastra Inspector a unui obiect de tip armă

Şablonul armei conţine element de coliziune, pentru a interacţiona cu inamicii, animator şi corp rigid (sau sistem de particule) pentru animaţii şi un script ce determină tipul de armă. Un astfel de şablon este reprezentat în figura următoare (Figura 49).

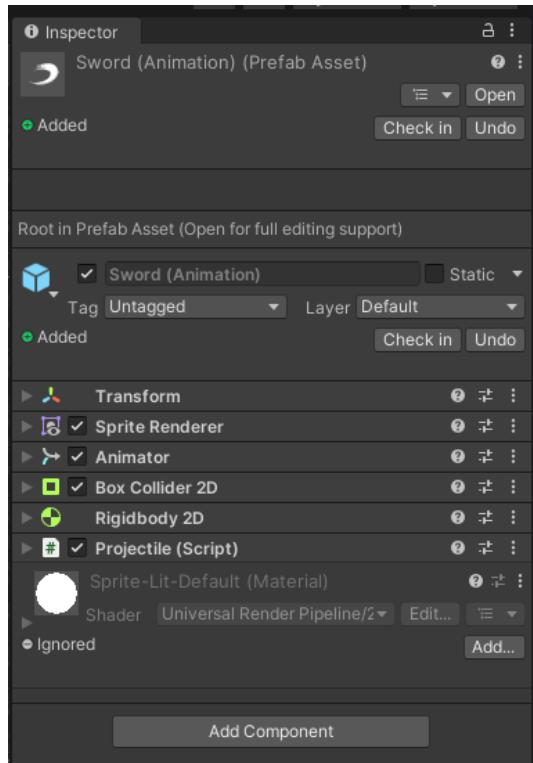


Fig. 49 Şablon pentru un obiect de tip armă

Echipamentul pasiv este creat în mod asemănător, iar în obiectul „scriptabil” sunt introduse statisticile suplimentare pe care le oferă jucatorului. Acestea pot fi observate în Figura 50.

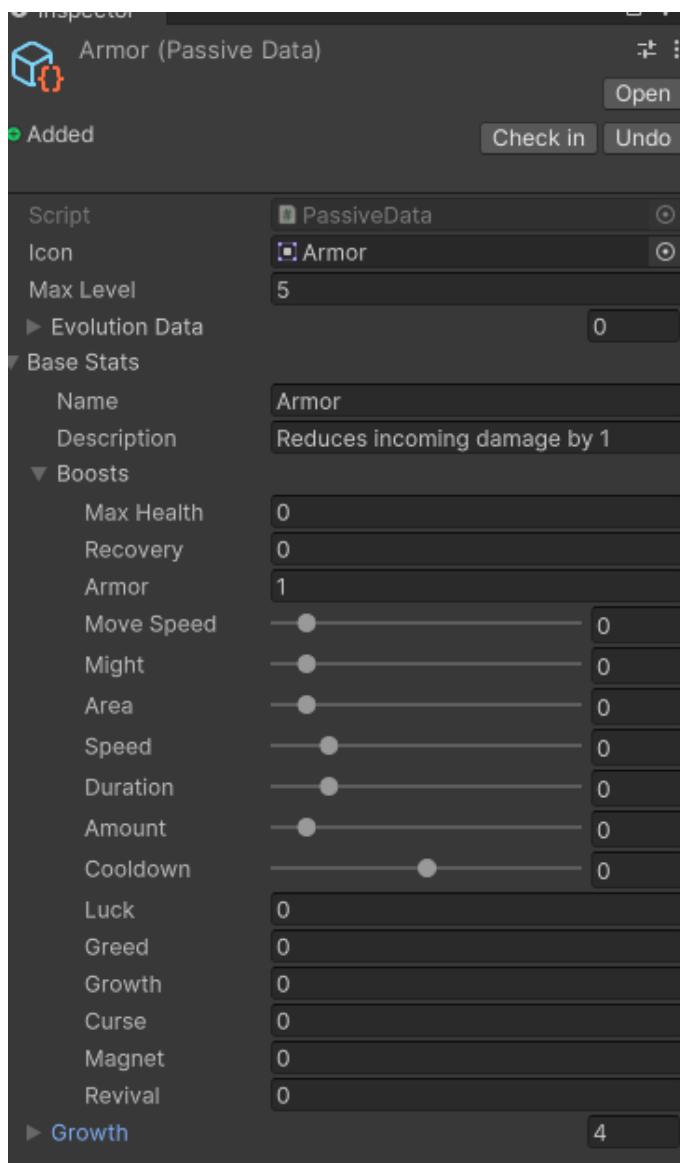


Fig. 50 Fereastra Inspector a unui echipament pasiv

#### 4.2.7 Legarea elementelor prin statistici

Au fost create diferite elemente precum personajul principal, echipamentul său și inamicii acestuia, însă nu se pot afecta între ele.

Rolul personajului este să supraviețuiască, în timp ce rolul inamicilor este de a îl omorî pe acesta. Pentru a se autoapăra, personajul se folosește de echipament pentru a deveni mai puternic și pentru a ucide inamicii.

Statisticile sunt cele care leagă aceste elemente, fiecare având rolul său. Acestea au fost implementate prin intermediul obiectelor „scriptabile”, script-ul necesar fiind obținut din tutorialul aflat pe URL-ul următor:

<https://youtu.be/0J0Jmc2IZYA?si=Mg2U8zoKY9U-mbVS>

Structura „Stats” este responsabilă cu introducerea statisticilor în obiectul „scriptabil” al personajului. Rezultatul implementării acesteia este vizibil în Figura 51.

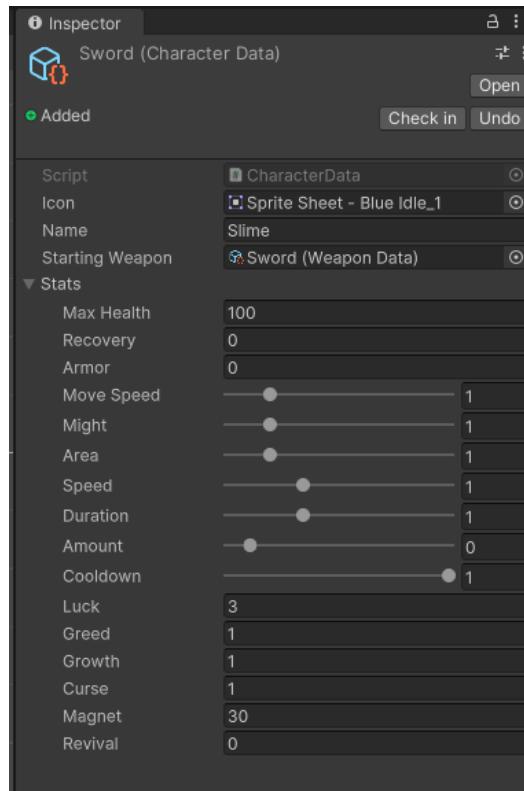


Fig. 51 Obiectul scriptabil Sword

În următorul Text Box, poate fi observat cum, încă din afară de statistici, sunt introduse și alte date precum iconița (Icon) ce reprezintă aspectul personajului, numele (Name) său și arma (Starting Weapon) cu care începe runda.

```
public struct Stats
{
    public float maxHealth, recovery, armor;
    [Range(-1, 10)] public float moveSpeed, might, area;
    [Range(-1, 5)] public float speed, duration;
    [Range(-1, 10)] public int amount;
    [Range(-1, 1)] public float cooldown;
    [Min(-1)] public float luck, greed, growth, curse;
    public float magnet;
    public int revival;
    public static Stats operator +(Stats s1, Stats s2)
    {
        s1.maxHealth += s2.maxHealth;
        s1.recovery += s2.recovery;
        s1.armor += s2.armor;
        s1.moveSpeed += s2.moveSpeed;
        s1.might += s2.might;
        s1.area += s2.area;
        s1.speed += s2.speed;
        s1.duration += s2.duration;
        s1.amount += s2.amount;
        s1.cooldown += s2.cooldown;
        s1.luck += s2.luck;
        s1.growth += s2.growth;
        s1.growth += s2.growth;
        s1.curse += s2.curse;
        s1.magnet += s2.magnet;
        return s1;
    }
}
```

În mod normal, statisticile pot crește sau scade fără limite, însă câteva dintre acestea sunt limitate.

„[Range (x, y)]” atribuie o limită inferioară (x) și una superioară (y), în felul acesta experiența de joc nu este afectată de o schimbare prea mare a statisticilor. De exemplu: statistica „moveSpeed” afectează viteza de mișcare a personajului. Dacă aceasta scade sub -1 jucătorul nu se poate mișca, iar dacă crește peste 10, viteza de mișcare este prea mare și experiența devine neplăcută.

„[ Min (-1)]” atribuie doar o limită inferioară, cea superioară nu fiind necesară.

Inamicii nu au nevoie de multe statistici, aceștia având nevoie doar de viață, putere de atac și viteză de mișcare. De data aceasta nu este folosit un obiect „scriptabil”, datele fiind introduse direct în şablonul acestora prin intermediul script-ului „Enemy Stats”, observabil în Figura 52 și obținut din tutorialul regăsit la adresa:

<https://youtu.be/RCOxhTsbAWo?si=aNMIWGEXN0kwlbSi>.

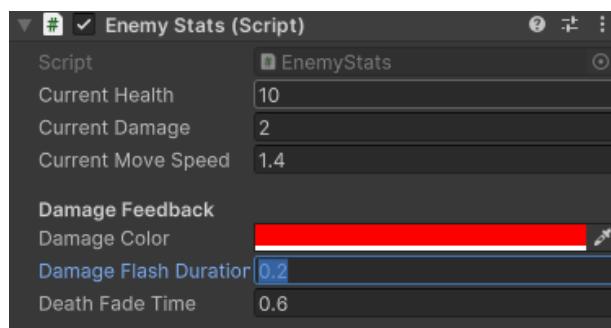


Fig. 52 Script-ul Enemy Stats

Acest script conține și detalii legate de ce se întâmplă când un inamic este lovit. Inamicul devine din ce în ce mai transparent și dispare, lucru de care se ocupă următoarea parte din codul sursă:

```
WaitForEndOfFrame w = new WaitForEndOfFrame();
float t = 0, origiAplha = sr.color.a;
//loop that fires every frame
while(t < deathFadeTime )
{
    yield return w;
    t += Time.deltaTime;
    sr.color = new Color(sr.color.r, sr.color.g, sr.color.b, (1 - t / deathFadeTime)
* origiAplha);
}
Destroy(gameObject);
```

Dupa ce este lovit suficient de mult pentru a muri, această funcție este apelată pentru a distrugă obiectul de joc, acesta reprezentând inamicul. După ce așteaptă un cadru, culoarea inamicului se schimbă, devenind din ce în ce mai palid până când este complet transparent, urmând să fie distrus obiectul de joc.

Echipamentul are, la rândul lui, propriile statistică, cel activ afectând inamicii, iar echipamentul pasiv afectează personajul.

În Figura 53, pot fi observate statisticile armei „Ring”, o armă de tip „Aura”, tip ce atacă o zonă întreagă, nu doar un inamic.

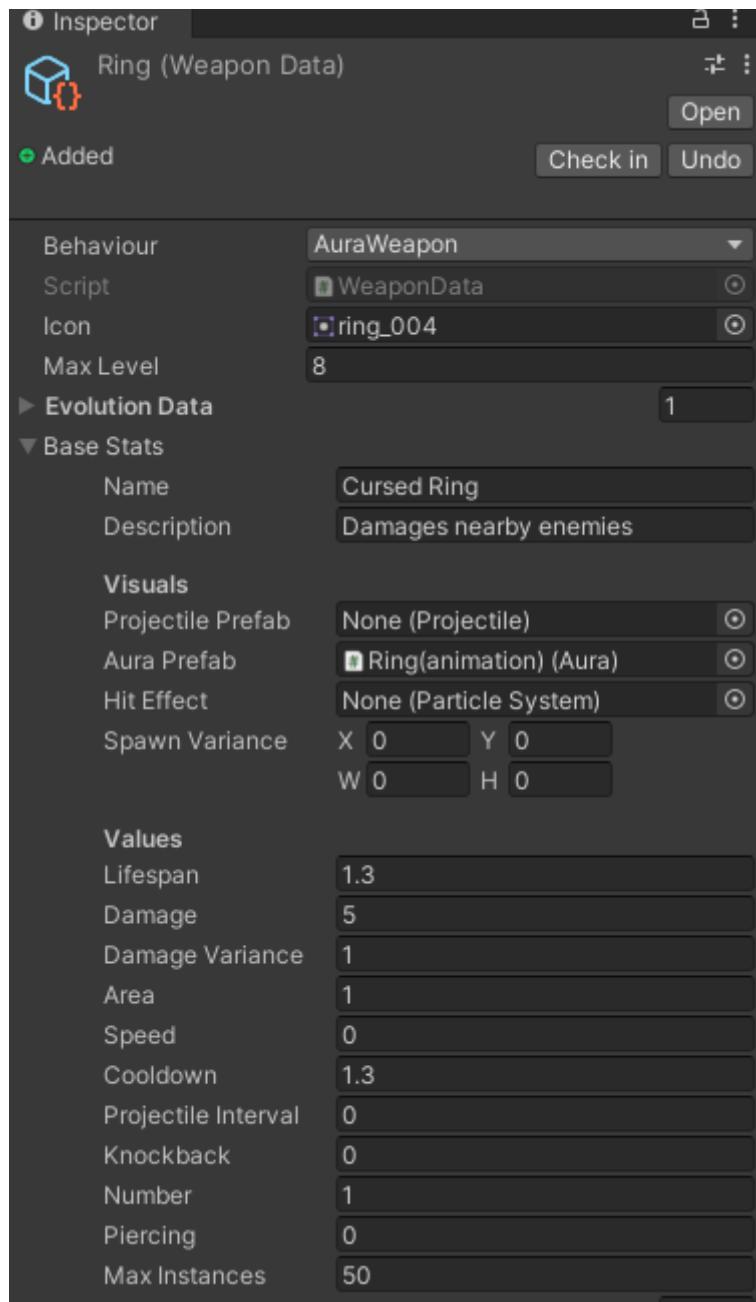


Fig. 53 Statisticile armei Ring

În mod similar cu statisticile personajului, au fost introduse și cele pentru arme. Ele sunt diferite, reprezentând ce poate face fiecare armă, în cazul acesta sunt importante durata de atac (Lifespan), puterea și zona acestuia ( Damage, Damage Variance, Area), durata de răcire (Cooldown) ce reprezintă intervalul dintre atacuri și de cate ori atacă (Number) atunci când este timpul să facă acest lucru.

Echipamentul pasiv afectează direct statisticile personajului, acestea fiind văzute în Figura 54.

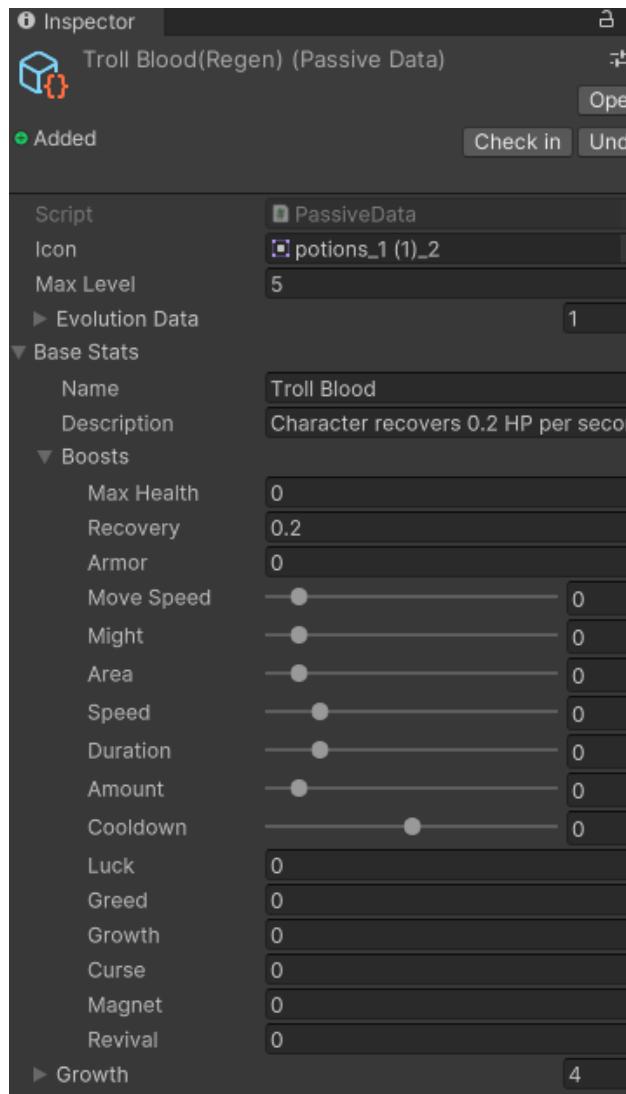


Fig. 54 Statisticile pasive

Aceasta bucată de echipament afectează puterea de regenerare a personajului. În structura „Stats” a personajului, prezentată mai devreme, se observă secțiunea „Stats Operator +”. Acesta are rolul de a aduna statisticile astfel, statisticile personajului sunt adunate cu a echipamentului pasiv, iar rezultatul final este atribuit personajului.

Fiecare statistică are rolul ei, după cum se poate observa:

- **Viață** (Health) reprezintă numărul maxim de lovitură pe care un obiect îl poate încasa înainte de a fi distrus. Numărul variază în funcție de puterea de atac a inamicului, viață fiind redusă până la 0, moment în care obiectul este distrus;
- **Atacul** (Might), împreună cu viteza de atac (Speed), influențează puterea, respectiv viteza, cu care un obiect atacă. Mărirea acestor statistică duce la atacuri mai puternice și mai dese, deci obiectul inamic este distrus mai repede;
- **Viteza de mișcare** (Move Speed) influențează în mod direct viteza cu care un obiect se deplasează pe hartă;
- **Armura** (Armor) reduce puterea de atac a inamicului;

- *Regenerarea* (Recovery) reprezintă câtă viață își regenerează obiectul pe secundă;
- *Suprafața* (Area) semnifică mărimea fiecărui atac;
- *Durația* (Duration) este responsabilă pentru timpul în care un atac se află în scena de joc. Este importantă pentru atacurile de tip proiectil, deoarece acestea traversează mai mult, deci sunt șanse mari să lovească un inamic;
- *Numărul* (Amount) semnifică numărul de proiectile lansate pe atac;
- *Perioada de răcire* (Cooldown) reprezintă intervalul dintre atacuri. Cu cât este mai mică cu sunt mai frecvente atacurile. Atunci când un echipament pasiv mărește acest status, este introdus cu – (de exemplu -0.2), pentru că statisticile curente sunt adunat, apoi actualizate personajului;
- *Norocul* (Luck) influențează șansa de a primi încă o opțiune atunci când personajul avansează în nivel;
- *Magnetul* (Magnet) semnifică raza de atracție a recompenselor. Atunci când o recompensă intră în raza magnetului, este atrasă spre obiect și obținută de acesta. O rază mare este importantă pentru că unii inamici mor la distanță mare, recompensa fiind mai greu de atins;

Greed, Growth, Curse și Revival nu au fost încă implementate, dar acesta este utilitatea planificată pentru ele:

- *Greed* mărește recompensa primită la finalul rundei;
- *Growth* mărește experiența primită de la inamici;
- *Curse* mărește statisicile inamicilor, făcând jocul mai dificil;
- *Revival* oferă posibilitatea de reînvia, obiectul având încă o șansă.

#### **4.2.8 Sistemul de Level up și evoluția armelor**

Personajul are posibilitatea de a deveni mai puternic, însă nu a fost implementat un sistem care să îi ofere această șansă.

Inamicii devin mai puternici pe parcursul rundei de joc, şablonul lor fiind înlocuit de un şablon atribuit unui inamic mai puternic. Obiectul „Enemy Spawner”, prezentat la subcapitolul 4.2.5, ocupându-se de acest lucru. În aceeași parte, a fost prezentat și sistemul de recompense oferite de inamici.

Prin acumularea experienței, personajul își mărește nivelul, fapt denumit și „Level up”. În Figura 55, sunt prezentate limitele fiecărui nivel, limite prezente în script-ul „Player Stats” atribuit personajului.

▼ Level Ranges		4
= ▼ Element 0		
Start Level	1	
End Level	2	
Experience Cap	5	
= ▼ Element 1		
Start Level	3	
End Level	20	
Experience Cap	10	
= ▼ Element 2		
Start Level	20	
End Level	40	
Experience Cap	600	
= ▼ Element 3		
Start Level	40	
End Level	9999	
Experience Cap	2400	
	+ -	

Fig. 55 Condițiile nivelelor

Experiența necesară pentru a trece la următorul nivel crește cu o variabilă ce este reprezentată de „Experience Cap”. Personajul începe cu nivelul 1 (Start Level) și are nevoie de 5 experiență pentru a avansa. Pentru că a atins nivelul 2 (End Level), se trece la următoarea secțiune (Element 1) unde pentru fiecare nivel atins, experiența necesară crește cu 10. În felul acesta, jucătorul trebuie să ucidă cât mai mulți inamici pentru a avansa în nivel. Funcția prezentată în continuare se ocupă de acest lucru:

```
void LevelUpChecker()
{
    if (experience >= experienceCap)
    {
        level++;
        experience -= experienceCap;
        int experienceCapIncrease = 0;
        foreach(LevelRange range in levelRanges)
        {
            if(level >= range.startLevel && level <= range.endLevel)
            {
                experienceCapIncrease = range.experienceCapIncrease;
                break;
            }
        }
        experienceCap += experienceCapIncrease
        UpdateLevelText();
        GameManager.instance.StartLevelUp();
        if (experience >= experienceCap)
            LevelUpChecker();
    }
}
```

Ea verifică dacă a fost atins numărul de experiență necesar, caz în care crește nivelul și mărește limita necesară.

Când jucătorul crește în nivel, are opțiunea de a allege între un echipament nou, sau îl avansează pe cel vechi, aşa cum se poate vedea în Figura 56:



Fig. 56 Alegerea echipamentelor în fereastra Level up

*Vampire Survivors* oferă jucătorilor posibilitatea de a evoluă armele.

Acest element a fost implementat și în Slime Survival, folosind informațiile oferite în tutorialul ce se regăsește la următoarea adresă:

<https://youtu.be/bbktg7OPyFU?si=qFDRDoTF33shfnpZ> .

Pentru a evoluă, sunt necesare următoarele condiții :

- Arma trebuie să atingă un anumit nivel;
- Un echipament pasiv specific trebuie să atingă un anumit nivel;
- Jucătorul trebuie să facă rost de un cufăr.

De înndeplinirea primelor două condiții se ocupă secțiunea de „Evolution Data”, secțiune ce se regăsește în obiectele „scriptabile” de tip armă și echipament pasiv. După cum se poate vedea în Figurile 57 și 58, informațiile introduse trebuie să coreleze, excepție fiind numele, dar se recomandă folosirea aceluiași nume pentru o organizare mai bună.

În exemplul dat, condiția (Condition) aleasă este cufărul, adică arma evoluează doar dacă este colectat un cufăr. „Evolution Level” reprezintă reprezentă nivelul la care echipamentul trebuie să fie ca arma să poată evoluă, în secțiunea „Catalysts” fiind introdus celălalt echipament necesar evoluției și nivelul acestuia. Arma și nivelul acesteia se regăsesc în secțiunea „Catalysts” a echipamentului pasiv și viceversa.

În secțiunea „Outcome” este introdusăarma rezultată după evoluție și nivelul acesteia, iar „Consumes” reprezintă ce se întamplă după evoluție: dispare arma, echipamentul pasiv, ambele sau nu dispare nimic.

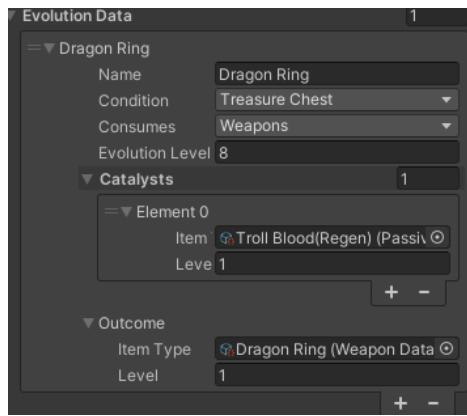


Fig. 57 Evolution Data pentru armă

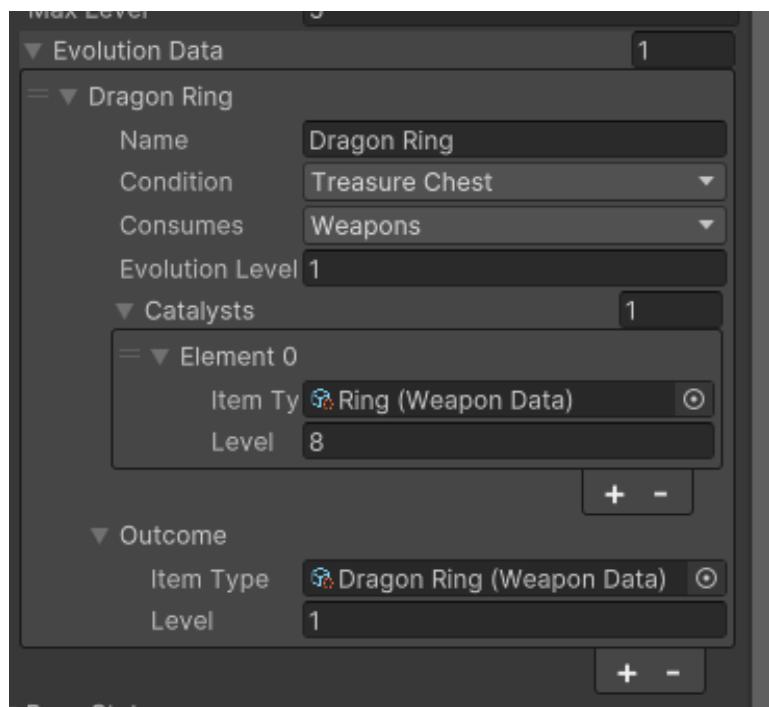


Fig. 58 Evolution Data pentru echipament pasiv

Cufărul este o recompensă oferită de un inamic de tip „Boss” ce este mult mai puternic față de un inamic normal, premiul oferit făcând eliminarea acestuia să merite.

De evoluție se ocupă clasa Item, datele acesteia fiind atribuite fiecărei bucăți de echipament.

În următorul fragment de cod sursă este prezentată funcția care încearcă să evolueze arma și verifică toate condițiile menționate mai sus.

```

public virtual bool AttemptEvolution(ItemData.Evolution evolutionData, int
levelUpAmount = 1)
{
    if (!CanEvolve(evolutionData, levelUpAmount))
        return false;
    //should the passive/weapons be consumed
    bool consumePassives = (evolutionData.consumes &
ItemData.Evolution.Consumption.passives) > 0;
    bool consumeWeapons = (evolutionData.consumes &
ItemData.Evolution.Consumption.weapons) > 0;
    // loop all the catalysts and check if they need to be consumed
    foreach(ItemData.Evolution.Config c in evolutionData.catalysts)
    {
        if (c.itemType is PassiveData && consumePassives)
            inventory.Remove(c.itemType, true);
        if (c.itemType is WeaponData && consumeWeapons)
            inventory.Remove(c.itemType, true);
    }
    if (this is Passive && consumePassives)
        inventory.Remove((this as Passive).data, true);
    else if (this is Weapon && consumeWeapons)
        inventory.Remove((this as Weapon).data, true);
    // add the new weapon
    inventory.Add(evolutionData.outcome.itemType);
    return true;
}

```

#### 4.2.9 Scenele de joc și interfețele

Elementele de interfață au fost create pe parcursul dezvoltării jocului, dezvoltare făcută cu ajutorul unei liste (playlist) de tutorial, acestea regăsindu-se la adresa:

<https://youtube.com/playlist?list=PLgXA5L5ma2Bveih0btJV58REE2mzfQLOQ&si=PYnu2Mg8ipqc2icP>

Toate elementele prezentate până acum fac parte din sena „Game”, aici având loc acțiunea jocului. Jucătorul nu poate intra direct în runda de joc, aşa că sunt necesare încă două scene:

- *Title Screen*: aceasta fiind scena văzută atunci când jocul este deschis;
- *Menu*: scena în care jucătorul își poate alege arma de început.

#### Scena Title Screen

Această scenă are nevoie de un buton care să ducă jucătorul la scena „Menu” și un buton ce oferă instrucțiunile de bază pentru a juca jocul. Obiectele de joc necesare pentru această scenă pot fi observate în Figura 59.

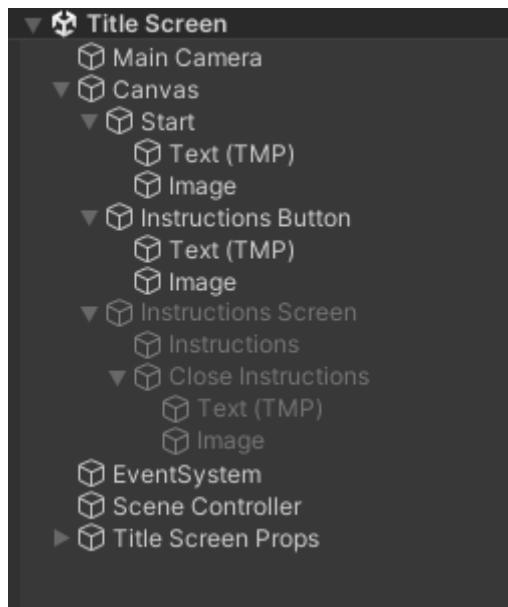


Fig. 59 Scena Title Screen

Poate fi observată o pânză ce conține două butoane: „Start” și „Instructions”.

Ambele butoane se folosesc de acțiunea „On Click”, ce poate fi observată în Figura 60. Aceasta este valabilă pentru toate butoanele, dictând ce se întâmplă atunci când sunt apăsate.

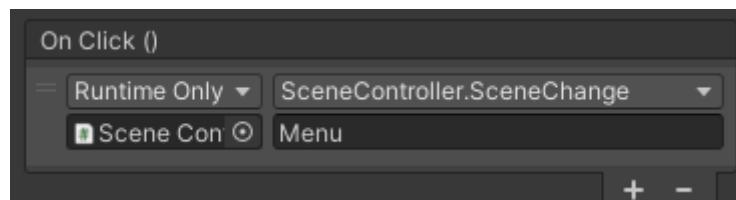


Fig. 60 Acțiunea On Click pentru Start

Atunci când este apăsat, butonul apelează la funcția de schimbarea a scenei, funcție ce se regăsește în script-ul „Scene Controller” introdus în obiectul cu același nume. La apăsare, scena este schimbată cu „Menu”.

Aspectul butoanelor poate fi văzut în Figura 61, butonul „Start” fiind mai mare pentru a fi evidențiat.

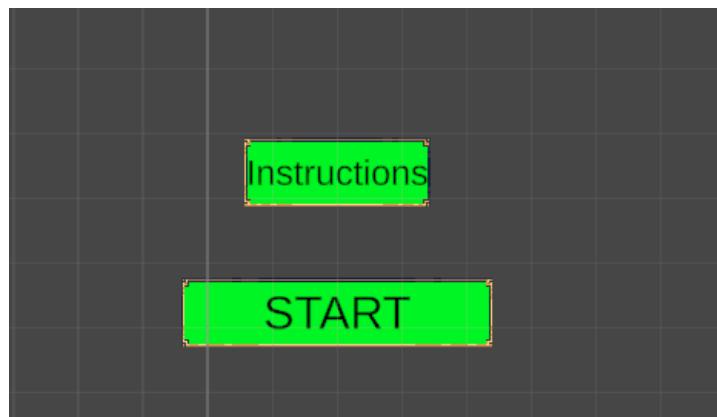


Fig. 61 Aspectul butoanelor Start și Instructions

Pentru afişarea instrucţiunilor este folosită o imagine ce are doar text introdus. Aceasta nu este activă, insă prin apăsarea butonului (Figura 62) setarea acesteia se schimbă și poate fi văzută de utilizator (Figura 63).

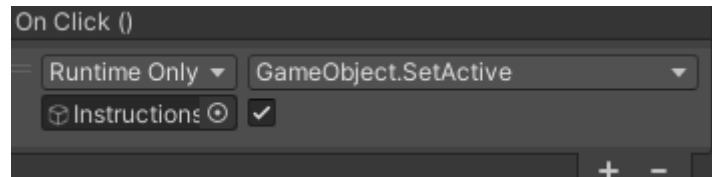


Fig. 62 Acţiunea On Click pentru instructions

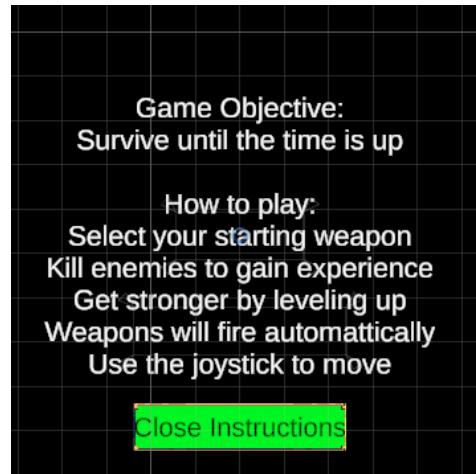


Fig. 63 Imaginea Instructions

De asemenea, este afișat butonul „Close Instructions” ce are rolul de a închide imaginea.

Pentru a avea un aspect mai frumos la deschiderea jocului, poate fi folosit obiectul „Main Camera”. În interiorul camerei pot fi așezate diferite obiecte și un fundal de hartă, aspectul final devenind cel din Figura 64:



Fig. 64 Aspectul scenei Title Screen

Această scenă a fost construită cu ajutorul tutorialului găsit la adresa următoare:  
[https://youtu.be/4U\\_f\\_qjOpZE?si=-CKi7gdjEqHjhlnL](https://youtu.be/4U_f_qjOpZE?si=-CKi7gdjEqHjhlnL).

## Scena Menu

Obiectele scenei pot fi văzute în Figura 65, această scenă bazându-se pe butoanele care aleg arma de început.

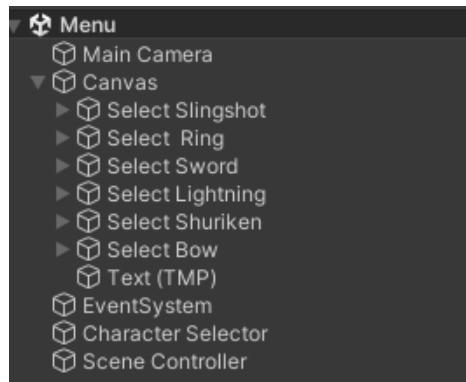


Fig. 65 Obiectele scenei Menu

Similar cu scena anterioară, este folosit „Scene Controller” pentru a schimba scena, dar și „Character Selector ” care are rolul de a alege arma., lucru ce poate fi observat în Figura 66.

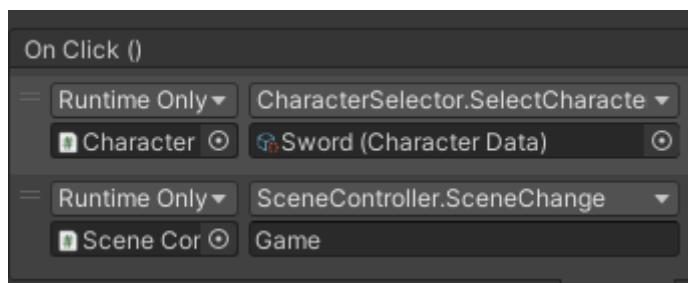


Fig. 66 Acțiunile butonului Sword

Fiecare buton reprezintă un personaj, singurul lucru ce îi diferențiază fiind arma de început. Următoarea clasă este „SceneController”, unde se preia numele scenei (variabila name) prin intermediu butonului, iar jucătorul este trimis în scena respectivă.

```
public class SceneController : MonoBehaviour
{
    public void SceneChange(string name)
    {
        SceneManager.LoadScene(name);
        Time.timeScale = 1;
    }
}
```

În mod similar funcționează și selectarea armei, obiectul armă fiind selectat prin intermediul butonului. Datele sunt returnate și personajul, împreună cuarma aleasă sunt introduse în scena de joc. Acest lucru este realizat prin următoarele funcții din codul sursă:

```

public void SelectCharacter(CharacterData character)
{
    characterData = character;
}

```

```

public static CharacterData GetData()
{
    if (instance && instance.characterData)
        return instance.characterData;
    return null;
}

```

## Scena Game

Pe parcursul dezvoltării jocului au fost introduce diferite elemente în această scenă, iar acum urmează interfață.

În timpul rundei sunt prezente următoarele tipuri de interfețe:

- Interfață normal, aceasta este vizibilă cât timp jocul rulează;
- Interfață de „Level up”, aceasta apare doar când personajul avansează în nivel;
- Interfață de pauză, vizibilă doar când jucătorul pune pauză rundei;
- Interfață de final, vizibilă atunci când se încheie runda.

Pentru a putea schimba interfețele, este necesar obiectul „Game Manager” ce conține un script cu același nume (Figura 67 ).

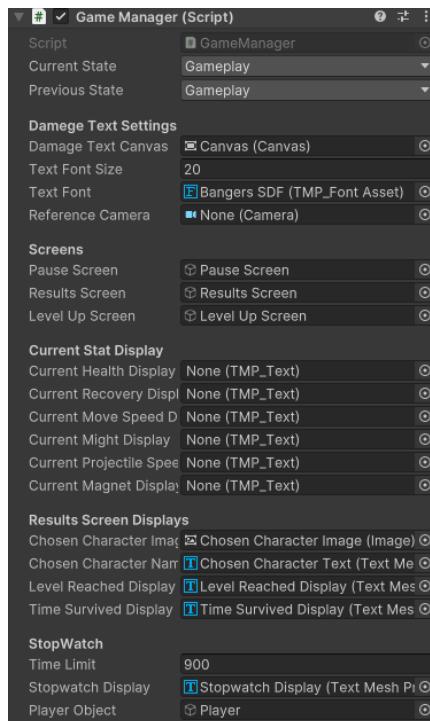


Fig. 67 Script-ul Game Manager

Secțiunea „Screens” conține toate interfețele ce pot apărea, mai puțin cea normal deoarece aceasta este vizibilă mereu. Funcția următoare se ocupă de schimbarea stării de joc, lucru ce schimbă interfața.

```
public void ChangeState(GameState newState)
{
    previousState = currentState;
    currentState = newState;
    OnGameStateChanged?.Invoke(newState);
}
```

Acste stări sunt declarate la începutul clasei și sunt folosite ca sistem de referință pe parcurs, pentru a să cînd este necesară schimbarea interfeței.

Interfața de „Level Up” este activată în momentul în care personajul crește în nivel. Ea conține opțiunile din care poate alege ( Figura 68 ) și design-ul interfeței (Figura 69).

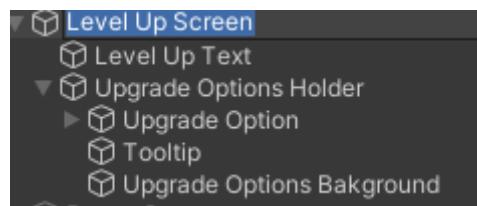


Fig. 68 Ecranul Level Up și componentele acestuia



Fig. 69 Interfața de Level Up

Principalele elemente sunt background-ul și opțiunile ce pot fi alese (Figura 70), acestea repetându-se în funcție de numărul de opțiuni valabile (Figura 56).

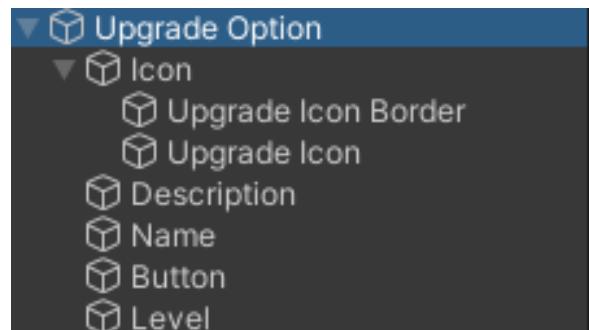


Fig. 70 Upgrade Option

Ecranul de pauză (Figura 71) nu conține multe elemente (Figura 72), dar afișează statisticile bonus actuale ale personajului (cu cât a devenit mai puternic de când a început și până acum). Pentru a face acest lucru, sunt preluate statisticile actuale ale personajului și introduse în cutia text a interfeței.

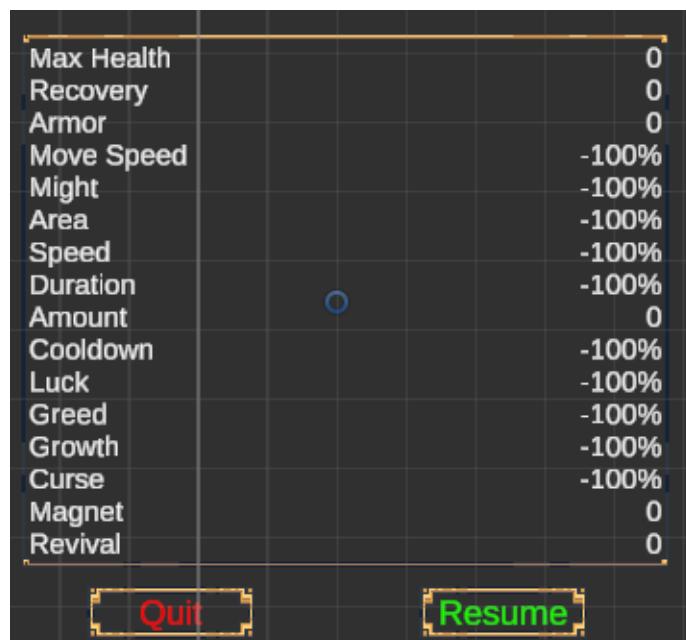


Fig. 71 Ecranul de pauză

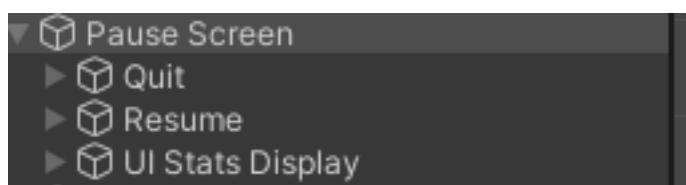


Fig. 72 Componentele Pause Screen

Pentru a nu strica experiența de joc, când această interfață este activă, timpul din joc este oprit. Același lucru se întâmplă când jucătorul crește în nivel, acordându-i timpul necesar pentru a lua decizia corectă.

De acest lucru se ocupă funcția „PauseGame”, ce setează viteza de scurgere a timpului la 0 cadre, în timp ce „ResumeGame” schimbă viteza la un cadru.

```

public void PauseGame()
{
    if(currentState != GameState.Paused)
    {
        ChangeState(GameState.Paused);
        Time.timeScale = 0f; // stop the game
        pauseScreen.SetActive(true);
    }
}

```

```

public void ResumeGame()
{
    if (currentState == GameState.Paused)
    {
        ChangeState(previousState);
        Time.timeScale = 1f; // resume the game
        pauseScreen.SetActive(false);
    }
}

```

Butonul „Quit” întoarce jucătorul în scena „Title Screen”, în timp ce butonul Resume, schimbă interfața în cea normală.

Interfața normală nu are un ecran separat, componentele acesteia fiind introduse în mod direct în pânza scenei (Figura 73).

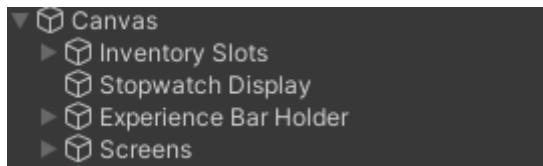


Fig. 73 Pânza scenei Game

Aceasta conține o bară de experiență ce se umple în funcție de experiența curentă, nivelul curent, afișează cât timp a trecut de când a început runda și inventarul ce conține echipamentul curent (Figura 74).

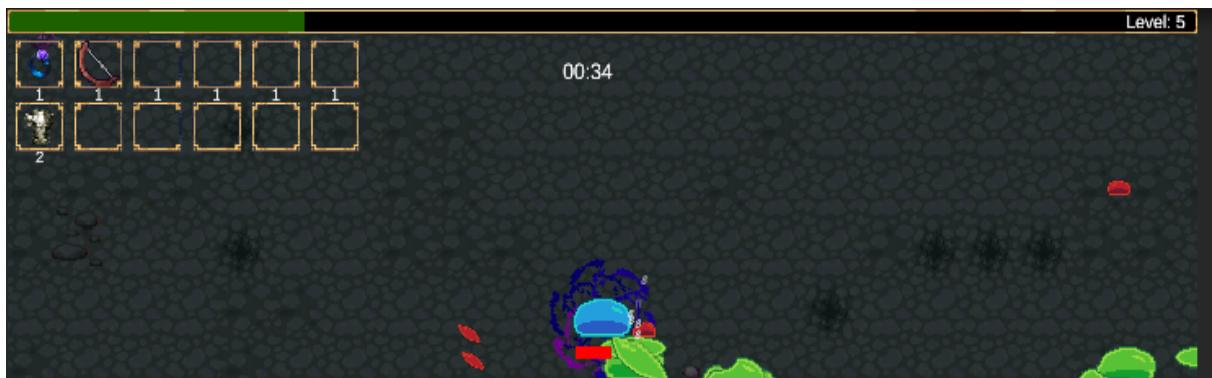


Fig. 74 Interfața normală

Inventarul curent, experiența și nivelul sunt extrase din datele curente despre personaj, în timp ce timpul este reținut de „Game Manager” (Figura 75).

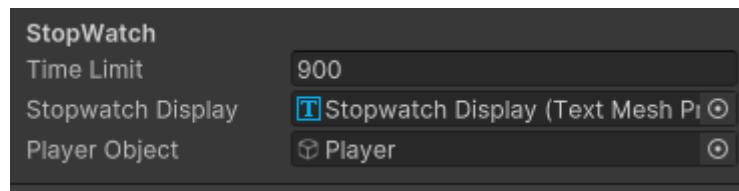


Fig. 75 Timpul în Game Manager

Acesta verifică timpul trecut de când obiectul ales (personajul) a fost introdus în scena jocului și oprește jocul când numărul de secunde a fost atins (Time Limit). De numărarea secundelor se ocupă funcția „UpdateStopwatch”, în timp ce „UpdateStopwatchDisplay” afișează acest lucru.

```
void UpdateStopwatch()
{
    stopwatchTime += Time.deltaTime;
    UpdateStopwatchDisplay();
    if(stopwatchTime >= timeLimit)
    {
        playerObject.SendMessage("Kill");
    }
}
```

```
void UpdateStopwatchDisplay()
{
    // calculate the number of minutes and seconds that elapsed
    int minutes = Mathf.FloorToInt(stopwatchTime / 60);
    int seconds = Mathf.FloorToInt(stopwatchTime % 60);
    //update the stopwatch text to display the elapsed time
    stopwatchDisplay.text = string.Format("{0:00}:{1:00}", minutes, seconds);
}
```

La final de rundă, jucătorul este întâmpinat de o interfață ce îi arată progresul (Figura 76), componentele acesteia fiind vizibile în Figura 77.



Fig. 76 Interfața cu rezultatul final

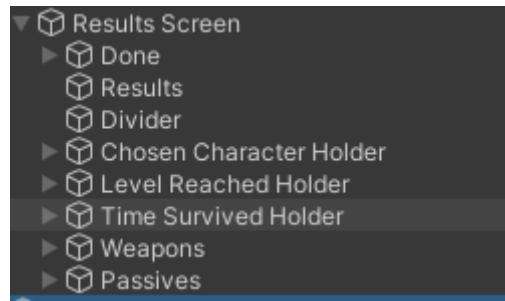


Fig. 77 Componentele ecranului final

„Time survived Holder” și „Level Reached Holder” afișează timpul supraviețuit, respectiv nivelul atins. „Chosen Character Holder” are rolul de afișare a iconiței personajului și arma aleasă la începutul jocului.

Butonul Done întoarce jucătorul la scena „Title Screen” pentru a putea rejuca harta, cu o altă armă sau aceeași, experiență fiind oricum diferită datorită sistemului de progresie, ce oferă diferite opțiuni de fiecare dată când jucătorul avansează în nivel.

## 4.3 Testare

### 4.3.1 Testarea script-urilor

Pe parcursul dezvoltării au fost introduce script-uri, fiecare afectând obiectele de joc în modul lor. Cu cât sunt introduce mai multe, cu atât mai probabil ca erorile să apară lucru ce poate opri procesul de dezvoltare.

Din fericire, consola Unity (Figura 78), oferă informații legate de erori precum motivul pentru care a apărut aceasta și linia de cod sursă, împreună cu script-ul din care provine.

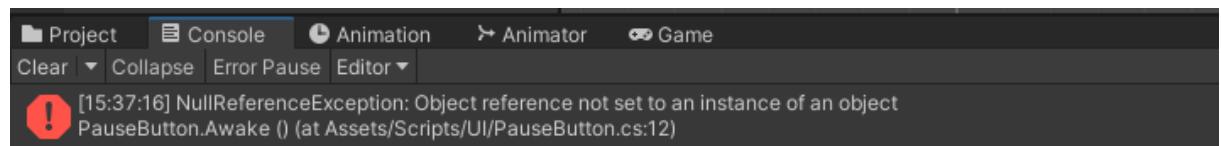


Fig. 78 Consola Unity

Totuși aceste informații nu sunt foarte precise, aşa că este bine să fie introdusă o atenționare în codul sursă. În exemplul de mai jos poate fi observat cum a fost făcut acest lucru: „print (name + "Initialised") ;”.

```
public virtual void Initialise(WeaponData data)
{
    print(name + "Initialised");
    base.Initialise(data);
    this.data = data;
    currentStats = data.baseStats;
    movement = GetComponentInParent<PlayerMovement>();
    ActivateCooldown();
}
```

Acesta afișează în consolă arma atribuită personajului (Figura 79).

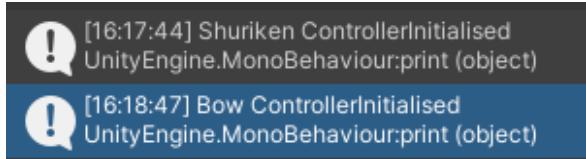


Fig. 79 Consola pentru arme atribuite

Este important să fie notate și potențialele erori, după cum se poate observa în funcția următoare:

```
public override bool DoLevelUp()
{
    base.DoLevelUp();
    // prevent level up if we are at max level
    if(!CanLevelUp())
    {
        Debug.LogWarning(string.Format("Cannot level up {0} to level {1}, max level
of {2} already reached", name, currentLevel, data.maxLevel));
        return false;
    }
    // otherwise, add stats of the next level to the weapon
    currentStats += (Stats)data.GetLevelData(++currentLevel);
    return true;
}
```

„Debug.LogWarning” are rolul de a afișa un mesaj de tip eroare în consolă, în cazul acesta când se încearcă mărirea nivelului unei bucătăți de echipament, peste capacitatea maximă, este afișată o eroare și numele echipamentului respectiv.

#### 4.3.2 Testarea jocului

Pentru testarea jocului, Unity oferă butonul de Play, aflat în partea de sus a ecranului, și fereastra „Game”, unde are loc acțiunea, acestea fiind evidențiate în Figura 80.

În această fereastră dezvoltatorul poate verifica starea actuală a jocului fără a fi nevoie să construiască proiectul și să instaleze aplicația.



Fig. 80 Butonul Play și fereastra Game

De asemenea, poate fi selectată rezoluția în care este afișat ecranul (Figura 81), astfel pot fi făcute ajustările necesare.

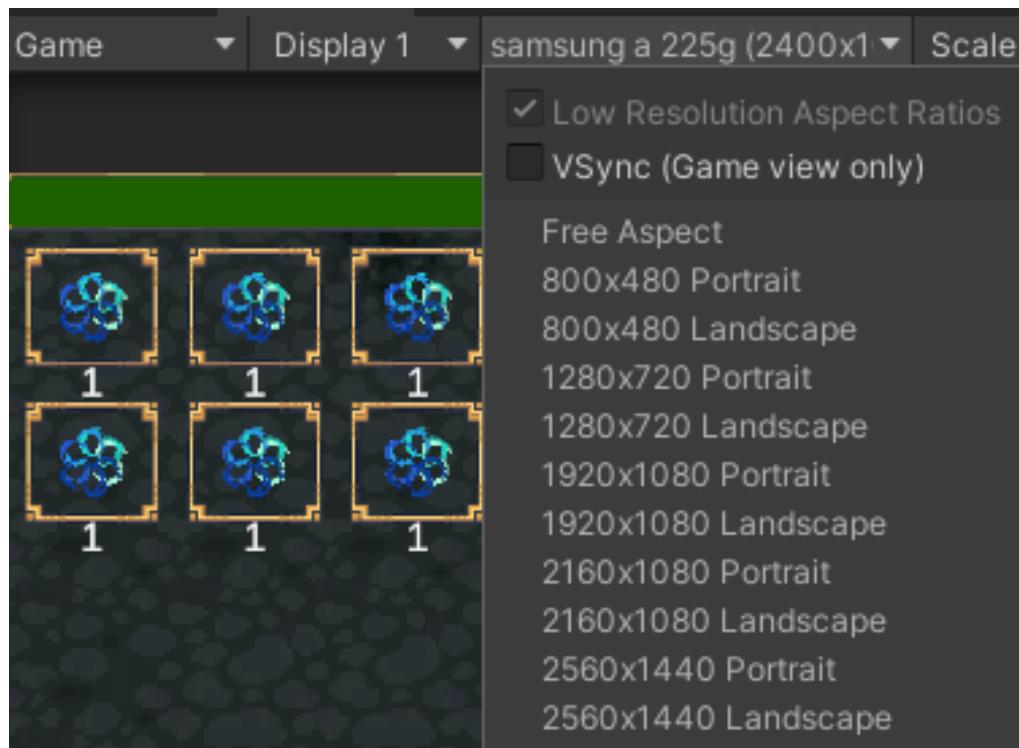


Fig. 81 Selectarea rezoluției

Pe lângă rezoluția dorită, poate fi selectat și dispozitivul, după cum se poate vedea în Figura 82, în dreapta căsuței „Simulator” fiind opțiunea de căutare a dispozitivului dorit.

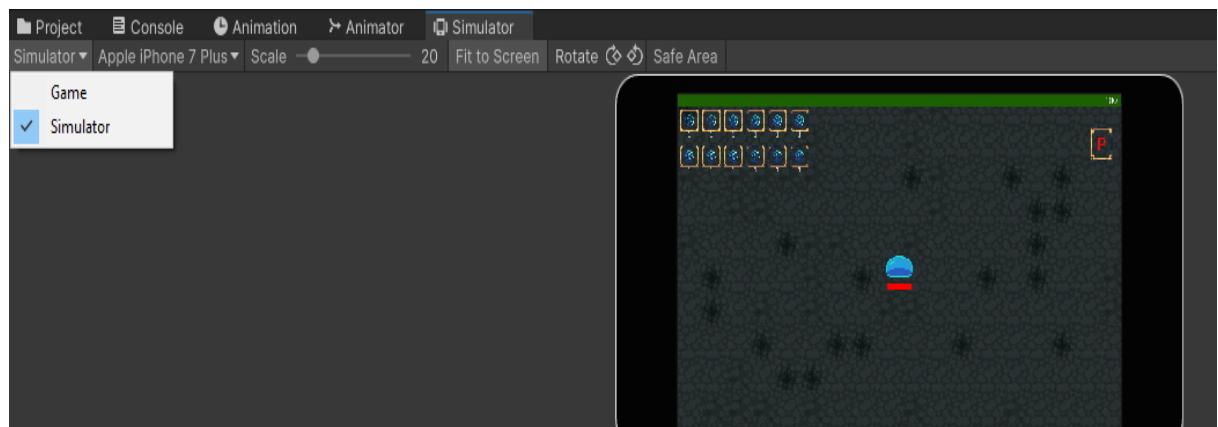


Fig. 82 Selectarea Dispozitivului

Pe parcursul dezvoltării, au fost introduse scene noi, iar după implementarea scenei „Menu”, trebuia schimbată mereu scena, altfel personajului nu îl era atribuită o armă. Pentru a scăpa de această necesitate, a fost introdus următorul fragment de cod sursă în script-ul „CharacterSelector”, mai exact în funcția statică „CharacterData GetData”.

Rolul acestui cod sursă este de a alege aleator o armă, din lista celor disponibile, această parte fiind marcată în Text Box-ul de mai jos.

```

// randomly pick a character if we are playing from the editor
#if UNITY_EDITOR
string[] allAssetsPath = AssetDatabase.GetAllAssetPaths();
List<CharacterData> characters = new List<CharacterData>();
foreach(string assetPath in allAssetsPath)
{
    if(assetPath.EndsWith(".asset"))
    {
        CharacterData characterData
        AssetDatabase.LoadAssetAtPath<CharacterData>(assetPath);
        if(characterData != null)
        {
            characters.Add(characterData);
        }
    }
}
// pick a random number if a character was found
if (characters.Count > 0) return characters[Random.Range(0, characters.Count)];
#endif

```

Pentru a testa cum funcționează sistemul de „Level up” au fost introduse butoane ce avansau direct nivelul echipamentului. Unul dintre acestea poate fi văzut în Figura 83.

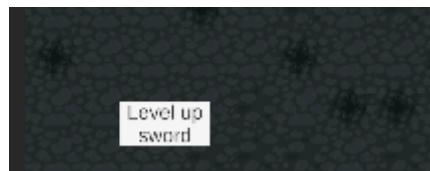


Fig. 83 Buton de Level up

Similar cu celealte butoane, acesta accesează direct funcția de „LevelUp” a script-ului „SwordWeapon”, mărind forțat nivelul acesteia. Este un buton util, în special pentru testarea evoluției armelor, acestea necesitând un anumit nivel.

Evoluția a fost testată și prin reducerea nivelului necesar sau mărirea numărului de experiență oferită de inamic.

Înainte de a implementa cufărul necesar evoluției ca recompensă, acesta a fost un şablon implementat direct în scena jocului, după cum se poate vedea în Figura 84:



Fig. 84 Cufăr în scena jocului

În mod similar au fost implementate diferite şabloane, în mod direct în scena de joc, fără a fi necesară aşteptarea prea lungă.

Pe parcursul procesului de dezvoltare, au fost introduse diferite statistici și date, unele ascunse în fereastra Inspector pentru a nu putea fi modificate. Pentru a le putea vedea,

se apasă pe cele de bulinde aflate în dreapta ferestrei și este selectat modul „Debug”, lucru evidențiat în Figura 85:

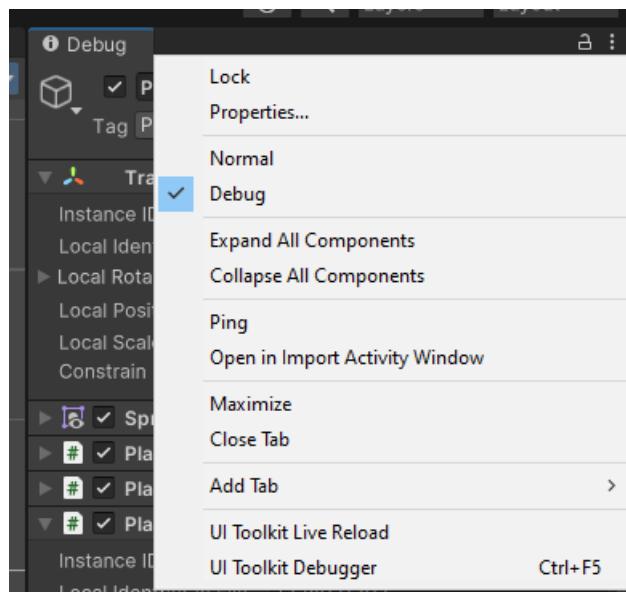


Fig. 85 Debug Inspector

În felul acesta, pot fi văzute statistici ascunse, precum cele din Figura 86:

► Actual Stats	
Health	0
Damage Effect	Player Damage VFX (Particle System)
Blocked Effect	None (Particle System)
Experience	0
Level	1
Experience Cap	0
Invincibility Duration	0.5
Invincibility Timer	0
Is Invincible	<input type="checkbox"/>

Fig. 86 Statistici ascunse

Pentru a oferi o experiență plăcută, testarea aplicației include și jucarea din perspectiva unui utilizator normal. Astfel că pentru a observa cât de dificil este jocul, a fost necesară încercarea acestuia și modificarea pe parcurs a statisticilor.

Ele sunt evidențiate prin culoarea gri și nu pot fi modificate, fără de datele normale.

#### 4.3.3 Exemple de erori găsite

Una dintre primele erori observate a fost legată de inamici, mai exact de animația lor de mișcare.

Script-ul de mișcare al inamicilor, cât și cel al personajului pornește animația atunci când obiectul se mișcă, insă nu este luată în considerare direcția mișcării. În Figura 87, se poate observa cum inamicul din stânga personajului are aceeași direcție de mișcare ca cei din dreapta acestuia.



Fig. 87 Eroare animație mișcare

Deși toți inamicii se îndreaptă spre inamic, direcția în care este îndreptată animația este dreapta.

Această eroare a fost rezolvată în tutorialul aflat la adresa de mai jos, însă soluția a fost doar pentru personaj.

<https://youtu.be/EIJk5KYzSJM?si=iheTmu0ruBRFOAUb>

Pentru a rezolva problema inamicilor, a fost adaptat script-ul „Player Animation”, astfel încât să funcționeze pentru inamici. Funcția următoare preia datele legate de direcția mișcării inamicului din „Enemies Movement” și întoarce animația dacă este cazul, astfel animația inamicilor se îndreaptă spre personaj.

```
void SpriteDirectionChecker()
{
    if (em.lastHorizontalVectorEnemy < 0)
    {
        sr.flipX = true;
    }
    else
    {
        sr.flipX = false;
    }
}
```

O altă eroare observată este legată de generarea hărții. În Figura 88, se poate observa cum o bucată de hartă nu a fost generată.



Fig. 88 Eroare la generarea hărții

Originea acestei erori se află în funcția „ChunckChecker” din script-ul „MapController”. Rolul funcției este de a verifica direcția în care se mișcă personajul, generând o bucată de hartă în direcția respectivă. Partea de mai jos se ocupă doar de direcția de sus, dar fiecare direcție este verificată în mod similar.

```
// check additional adjacent directions for diagonal chunks
if(directionName.Contains("Up"))
{
    CheckAndSpawnChunk("Up");
}
```

Nu este suficient să fie generată o bucată doar în direcția de mișcare, deoarece atunci când personajul se află la marginea unei bucăți nu este generat nimic în jurul acestuia, doar în direcția sa de mers. Pentru a fixa eroarea, a fost modificat codul sursă astfel:

```
// check additional adjacent directions for diagonal chunks
if(directionName.Contains("Up"))
{
    CheckAndSpawnChunk("Up");
    CheckAndSpawnChunk("Right");
    CheckAndSpawnChunk("Left");
}
```

În acest moment, este generată o bucată în jurul personajului, cât timp nu este o bucată în acel loc deja. Există părți similare pentru fiecare direcție cardinală.

## 4.4 Optimizare

Pe parcursul dezvoltării și testării au fost descoperite limitări ale unor sisteme și funcții ce consumă resurse în exces și îngreunează sarcina dezvoltatorului.

Script-urile sunt conectate atât la alte obiecte de joc, cât și la alte Script-uri. Pentru a nu avea probleme pe parcursul înlocuirii și pentru a putea marca mai ușor elementele cheie, fiecare clasă este marcată ca „Obsolete”. În felul acesta, editorul nu le ia în considerare atunci când sunt aplicate, și noile script-uri pot fi testate mai ușor. Codul sursă următor prezintă clasa „Enemy Spawner” ca fiind marcată, în Figura 89, fiind observat aspectul acesta în Inspector.

```
[System.Obsolete("Replaced by the Spawn Manager.")]
public class EnemySpawner : MonoBehaviour
```

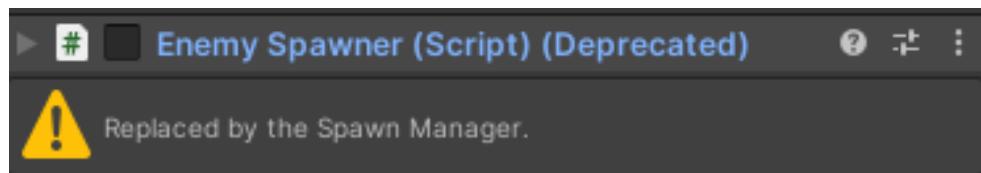


Fig. 89 Enemy Spawner marcat ca Obsolete

#### 4.4.1 Optimizare făcută la finalul dezvoltării

Sistemul de generare a inamicilor este unul limitat. După cum se poate observa în Figura 90, pentru a introduce un val de inamici sunt necesare diferite date.



Fig. 90 Val de inamici în sistemul vechi

Sistemul nu doar că include un alt grup de date (Enemy Groups), dar conține și unele care nu trebuie modificate, altfel nu poate începe valul. „Spawn”, „Wave Quota” și „Spawn Count ” sunt schimbate pe parcurs, ele fiind actualizate în timpul jocului în funcție de câți inamici sunt în viață. După ce au fost omorâți toți, se trece la valul următor, dacă există.

Acest sistem nu este cel mai potrivit, scopul jocului fiind de a supraviețui un anumit timp, sau cât mai mult. Pentru că jocul se oprește când personajul moare sau când timpul s-a scurs, fiecare val de inamici trebuie testat, astfel încât să dureze un anumit număr de secunde, lucru ce nu se poate face ușor, deoarece jucătorul are la dispoziție diferite rute de a avansa. O altă opțiune o reprezintă un număr mare de inamici, dar acest lucru ar strica diversitatea jocului, fiecare tip de inamic fiind generat după ce au fost generați toți inamicii din grupul anterior (Enemy Groups).

Pentru implementarea unui nou sistem, mai eficient, a fost folosit tutorialul aflat pe URL-ul următor:

<https://www.youtube.com/live/ZNJmp2gjDfU?si=q0uDaD3ty2noMAjH>.

Acesta conține o ierarhie de clase (Figura 91), fiecare dintre acestea reprezentând un obiect „scriptabil” ce poate fi refolosit cu ușurință.

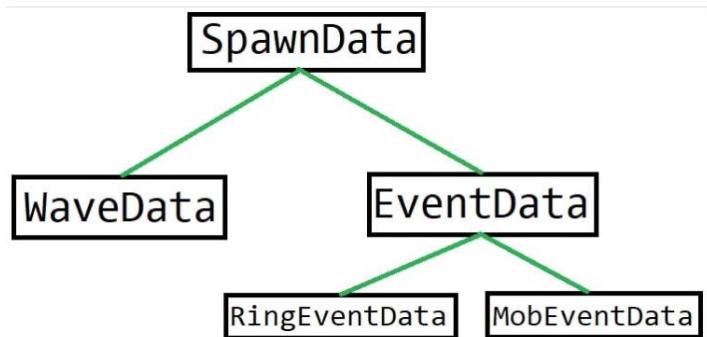


Fig. 91 Ierarhie de clase pentru a genera inamici, obținută din tutorial

În Figura 92, pot fi observate două script-uri, „Spawn Manager” fiind responsabil cu generarea normală a inamicilor, iar „Event Manager” se ocupă de valuri speciale de inamici.

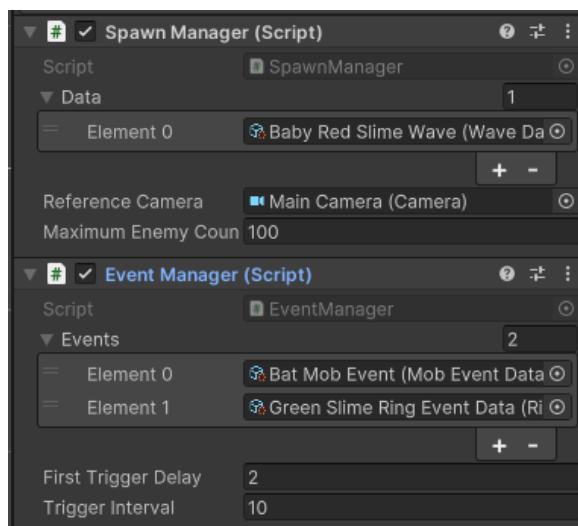


Fig. 92 Spawn Manager și Event Manager

Ambele operează în mod similar, fiind introduse obiectele „scriptable” ale valurilor. În felul acesta nu este necesară reintroducerea datelor unui val, ele fiind reutilizabile.

Obiectul primului val poate fi observat în Figura 93.

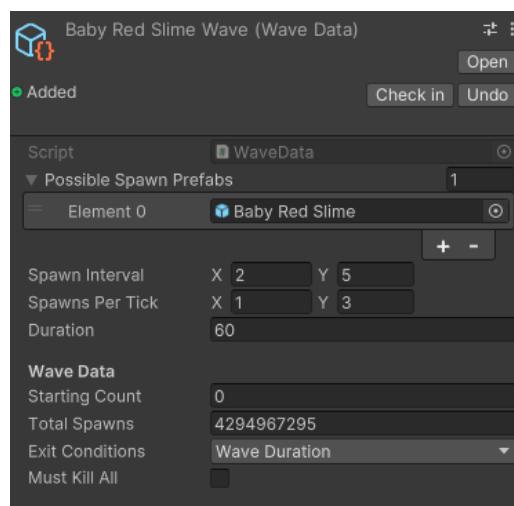


Fig. 93 Primul val de inamici

Acesta alege din şabloanele inamicilor (Possible Spawn Prefabs) şi îi generează, la un interval ales (Spawn Interval). Poate fi introdus şi numărul de inamici generaţi în acelaşi timp (Spawns Per Tick), împreună cu durată valului (Duration). Un element important este „Exit Conditions”, ce permite ca valul să se încheie atunci când expiră timpul sau când mor toţi inamicii (notati în Total Spawns), faţă de sistemul anterior, acesta finalizând valul când au fost generaţi toţi inamicii.

Celelalte tipuri de valuri funcţionează asemănător, în Figura 94, fiind reprezentat un val de tip grup, acesta generând un grup de inamici ( Possible Spawn Prefabs) care merg în direcţia personajului.

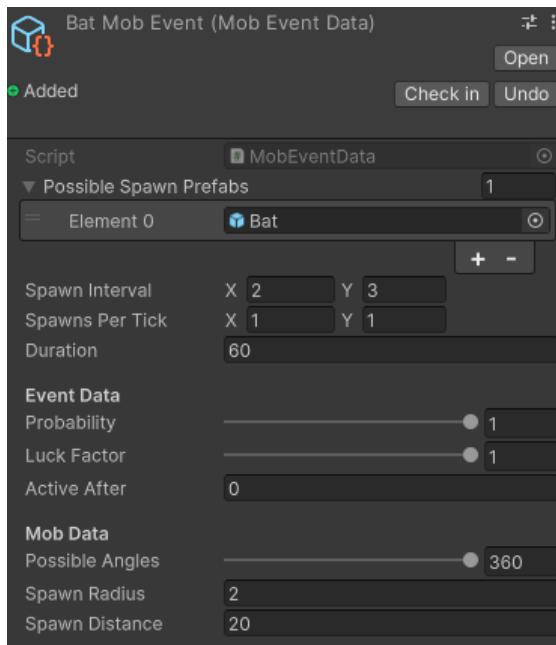


Fig. 94 Val de inamici de tip grup

După cum se poate observa în figurile anterioare, primul set de date se repetă. Inamicul ales, durata, intervalul de generare și numărul este salvat în clasa părinte „Spawn Data”. În felul acesta codul sursă din script poate fi refolosit la crearea unor noi tip de valuri.

Optimizarea nu este limitată doar la resursele folosite, ea reprezentând întreg procesul de dezvoltare.

Prin optimizarea sistemului de generare a inamicilor, nu doar că a fost eficientizată consumarea resurselor, dar dezvoltarea jocului a devenit mai ușoară prin implementarea mai sistematică și ușor de înțeles a inamicilor. Acest lucru ajută și utilizatorul, acesta putând beneficia de conținut nou.

Procesul de dezvoltare este unul îndelungat, astfel că dezvoltatorul trebuie să treacă în mod repetat prin implementarea unor elemente noi, testarea acestora și optimizarea lor.

#### 4.4.2 Optimizări făcute pe parcurs

Acest tip de optimizări au fost făcute pe parcursul dezvoltării, astfel că, sistemul de echipament, statistici și interfețe, prezentate anterior, sunt la o formă ce a trecut deja prin procesul de optimizare.

În sistemul anterior, pentru a implementa o armă nouă, era necesar un şablon și un obiect „scriptabil” pe nivel de armă, deci o armă care poate atinge nivelul 8 are nevoie de 16 obiecte, în timp ce în noul sistem două sunt suficiente. Figura 95 prezintă un exemplu de astfel de obiect „scriptabil”.

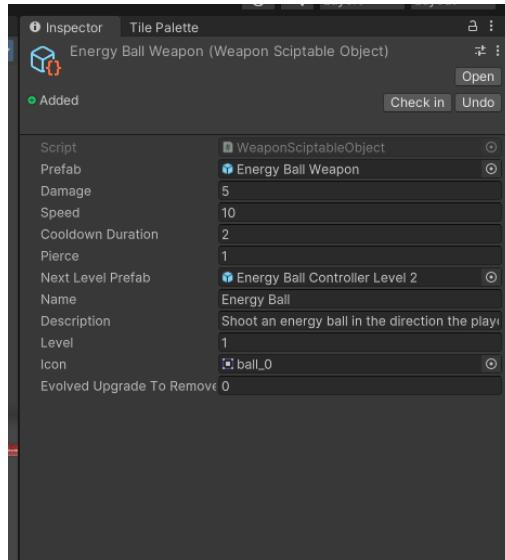


Fig. 95 Obiect scriptabil vechi pentru armă

Pot fi observate atât statisitcile (mai puține față de cele actuale), cât și şablonul pentru nivelul următor al armei, împreună cu nivelul armei curente.

Fiecare obiect era conectat la un controller (Figura 96), acesta având doar rolul de a îi permite personajului să folosească arma.

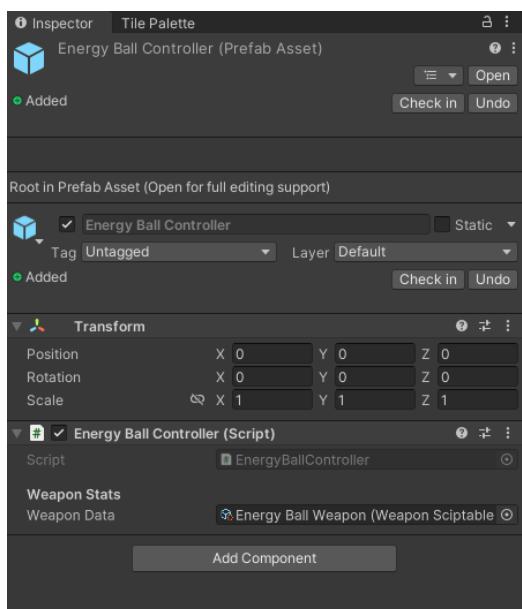


Fig. 96 Controller vechi de arme

Într-un final, este prezentat şablonul unei arme vechi în Figura 97, prin intermediul lui fiind obiectul „scriptabil” al armei fiind aplicat încă odată.

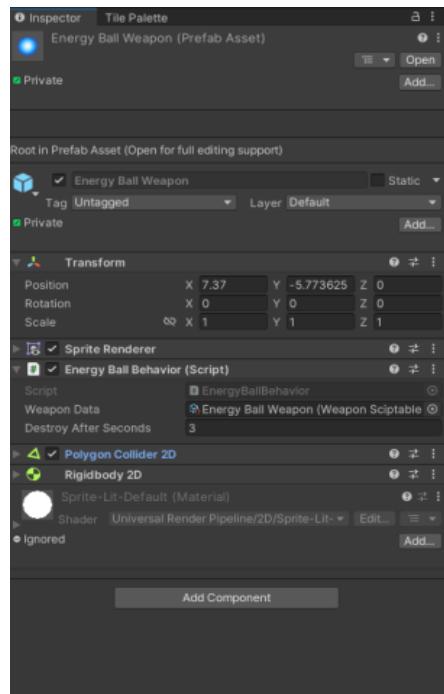


Fig. 97 Şablon armă veche

O altă schimbare semnificativă este inventarul, cel vechi (Figura 98) necesitând foarte multe date.

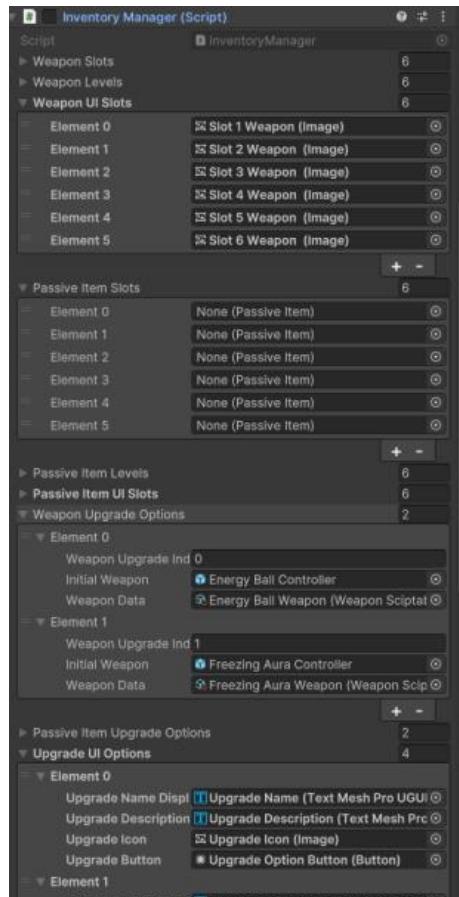


Fig. 98 Inventarul vechi

Cel nou (Figura 99) necesită date doar despre script-urile interfeței (Weapon UI și Passive UI), locația inventarului (Weapon Slots și Passive Slots) și obiectele „scriptabile” ale echipamentului (Available Weapons și Available Passives). Este un inventar mai compact, cel anterior necesitând mai multe date. Acest inventar, împreună cu sistemul nou de echipamente, a făcut implementarea mult mai ușoară și rapidă.

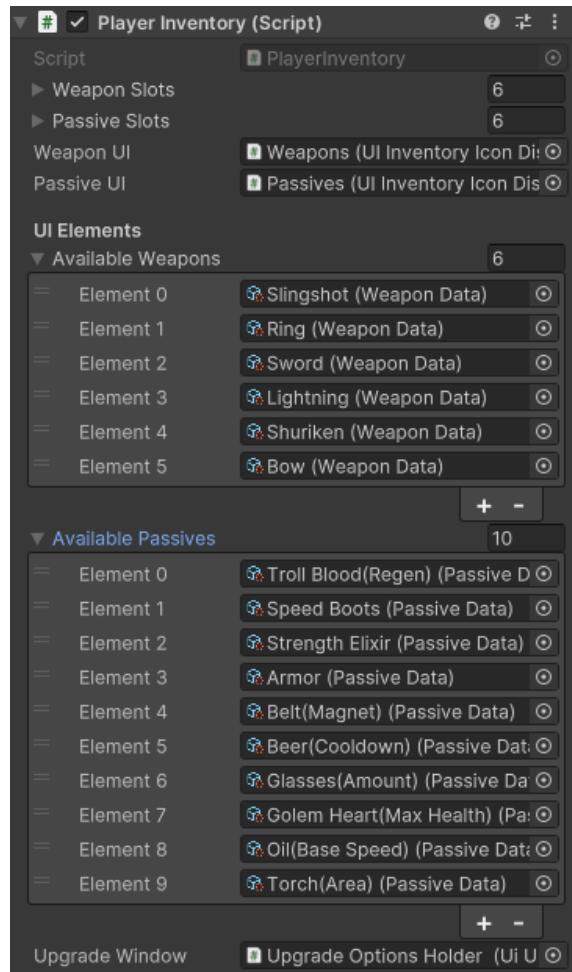


Fig. 99 Inventarul nou

## 4.5 Construirea pentru mobil

Acum că jocul a fost testat și optimizat, este timpul pentru crearea propriu-zisă și dezvoltareasă jocului pentru telefonul mobil.

Pentru a se putea juca, utilizatorul trebuie să poată realiza următoarele două acțiuni: să apese pe butoane și să miște personajul. Din fericire, Unity se ocupă de apăsarea butoanelor atunci când este construit proiectul, deci ce mai rămâne de făcut este partea de mișcare.

Pentru implementarea mobilității pe telefon, este folosit un joystick virtual descărcat din sursa:

<https://assetstore.unity.com/packages/tools/utilities/virtual-joystick-pack-261317>.

După introducerea acestuia în pânza scenei (Figura 100), sunt reglate detaliiile acestuia (Figura 101) a asigura o mișcare fluentă pe ecranul de joc (Figura 102).

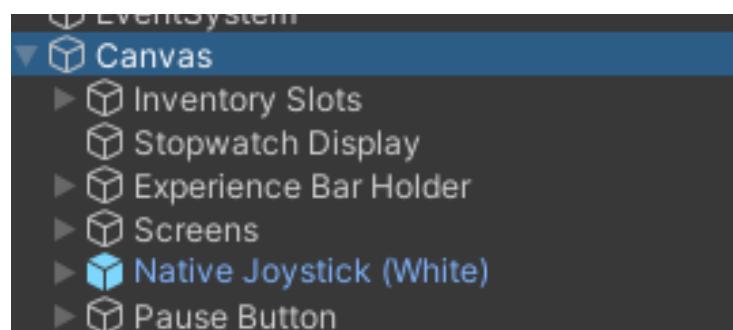


Fig. 100 Pânza scenei cu joystick

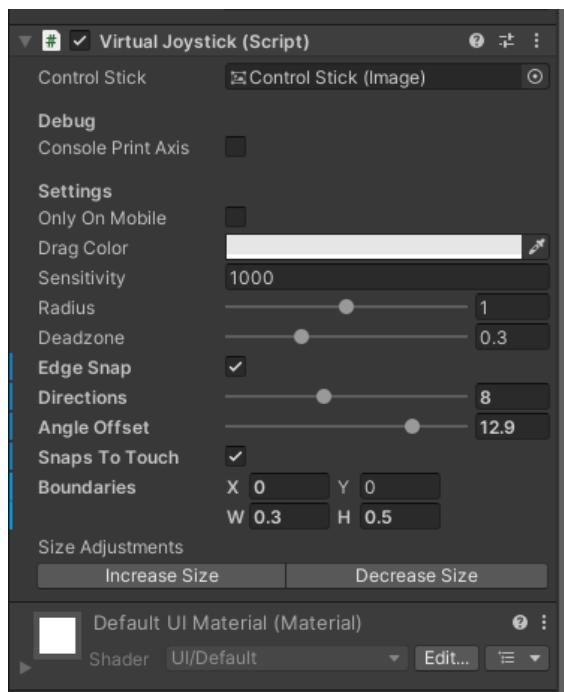


Fig. 101 Script-ul joystick-ului

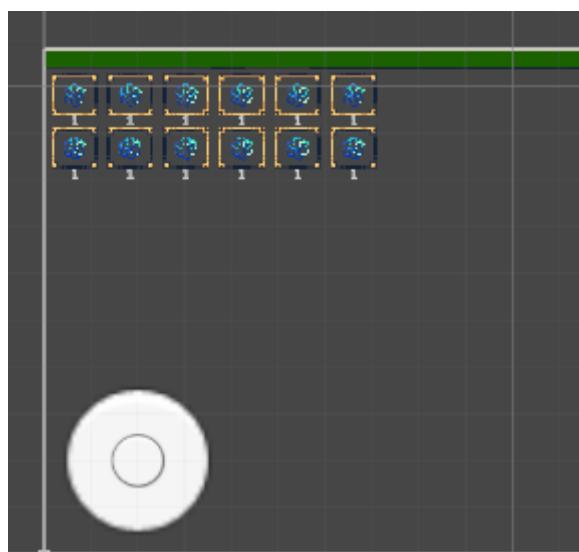


Fig. 102 Joystick-ul pe ecran

Acum că utilizatorul are cu ce să miște personajul, este necesar să fie făcută o legătură între obiect și joystick. În script-ul „Player Movement”, în interiorul funcției „InputManagement” este introdus următorul fragment de cod sursă:

```
float moveX, moveY;
if(VirtualJoystick.CountActiveInstances() > 0 )
{
    moveX = VirtualJoystick.GetAxisRaw("Horizontal");
    moveY = VirtualJoystick.GetAxisRaw("Vertical");
}
else
{
    moveX = Input.GetAxisRaw("Horizontal");
    moveY = Input.GetAxisRaw("Vertical");
}
```

Această secțiune se asigură că, cât timp există un joystick, sunt salvate direcțiile transmise de acesta și sunt folosite pentru a mișca personajul.

Acum că utilizatorul se poate juca, este timpul pentru reglarea interfeței. Industria mobilă este diversificată, fiind disponibile telefoane și tablete de diferite dimensiuni, lucru ce modifică rezoluția jocului.

Pentru a face interfața adaptabilă, este schimbat „Rect Transform-ul” fiecarei componente.

Acesta are o secțiune denumită „Anchors”, vizibilă în Figura 103.



Fig. 103 Rect Transform

Fiecare element din această secțiune reprezintă un colț al obiectului de interfață, colțuri vizibile în Figura 104.



Fig. 104 Colțurile din Rect Transform

Acestea rămân ficate, deci obiectul de interfață se află mereu în aceeași poziție pe ecran, idiferent de rezoluție. Acest lucru se poate observa comparând Figura 105 cu Figura 106:

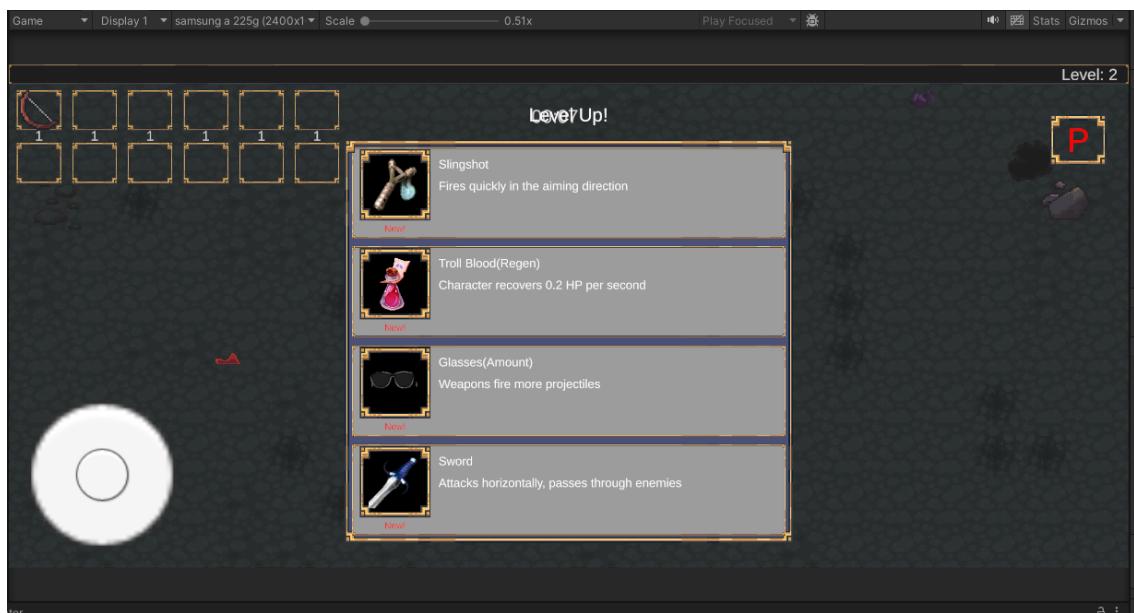


Fig. 105 Ecran cu rezoluție mare

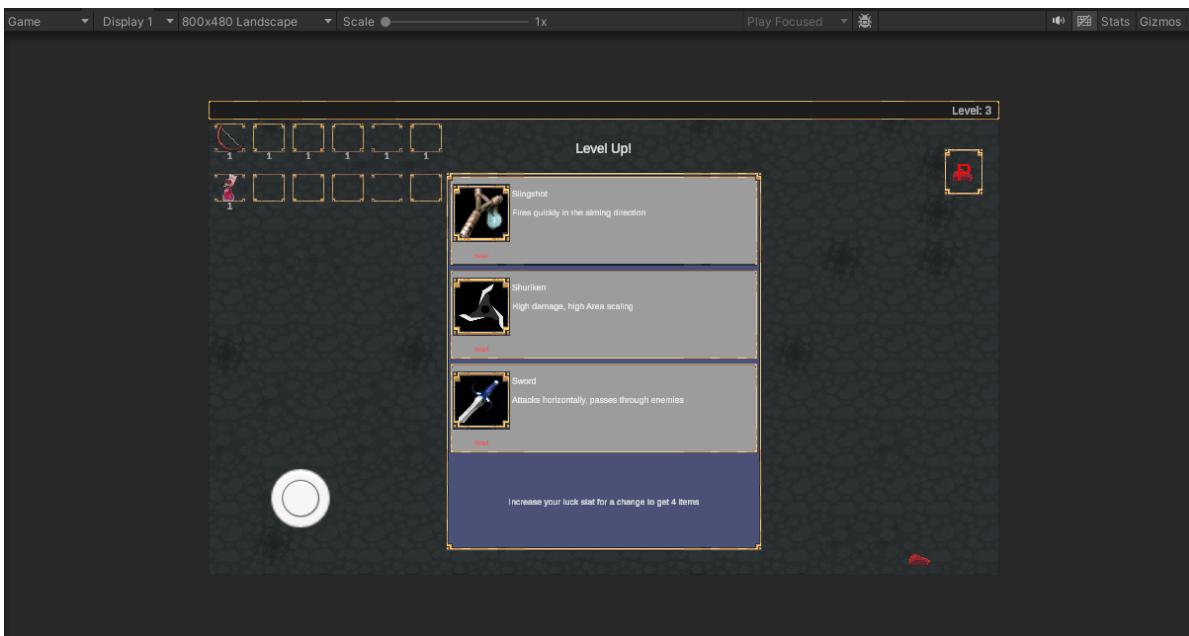


Fig. 106 Ecran cu rezoluție mai mică

Urmează pasul final, construirea aplicației. Pentru a face acest lucru, se apasă în stânga sus, pe butoanele „File” > „Build Settings”, așa cum poate fi văzut în Figura 107:

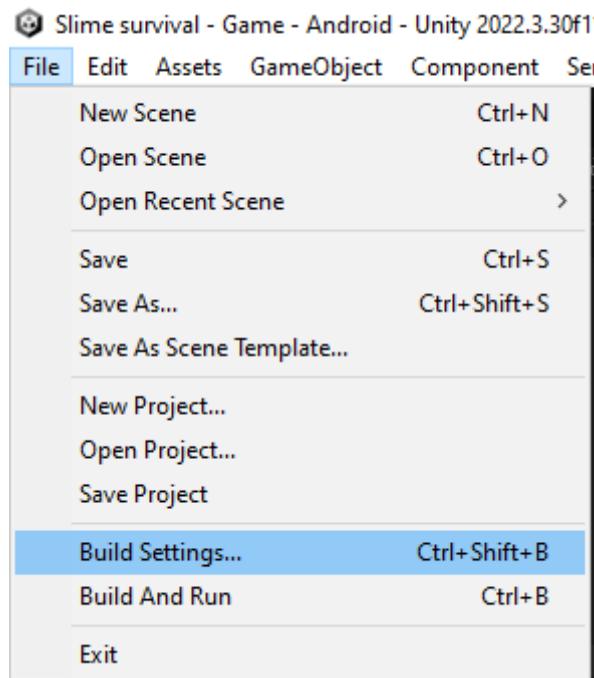


Fig. 107 Butoanele File și Build Settings

În fereastra ce se deschide (Figura 108), se selectează platforma dorită, în acest caz Android și se apasă pe butonul „Switch Platform” (Figura 109) pentru schimba platforma, dacă nu este deja cea de Android. De asemenea, în secțiunea „Scenes în Build” trebuie aranjată ordinea în care sunt construite scenele, prima fiind cea văzută de utilizator când pornește aplicația.

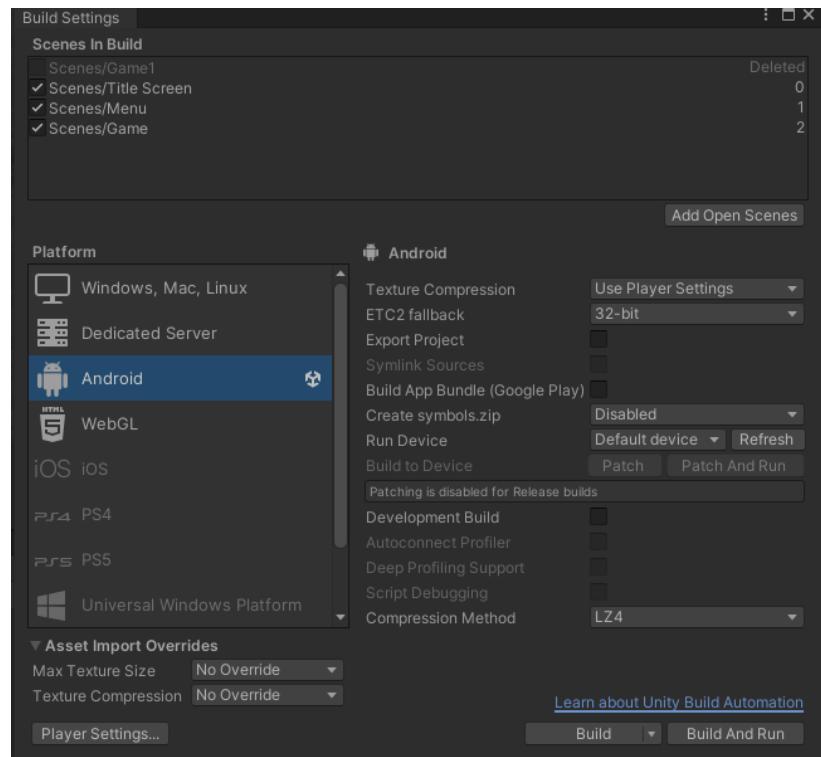


Fig. 108 Fereastra Build Settings



Fig. 109 Butonul Switch Platform

Înainte de a apăsa pe „Build”, este important să fie fixate următoarele două lucruri în „Player Settings” :

- Icoană jocului (Figura 110), care poate fi fixată prin introducerea imaginii dorite în secțiunea „Icon”;
- În secțiunea „Resolution and Presentation”, trebuie dezactivat modul portret (Figura 111), jocul putând fi jucat doar în modul landscape.

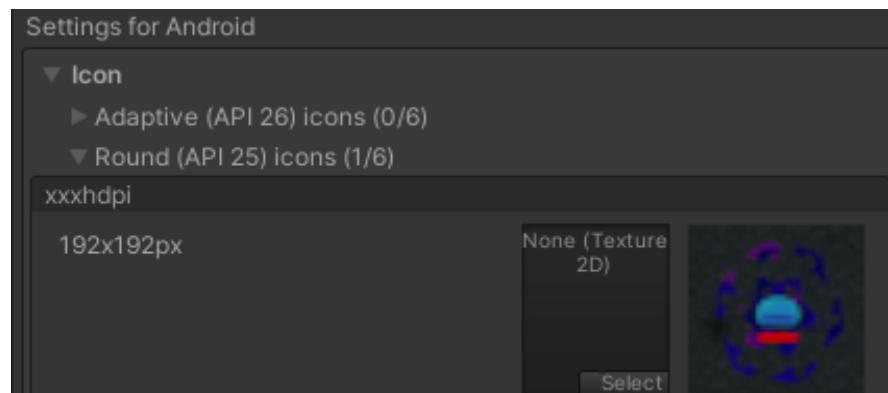


Fig. 110 Implementarea iconiței de joc

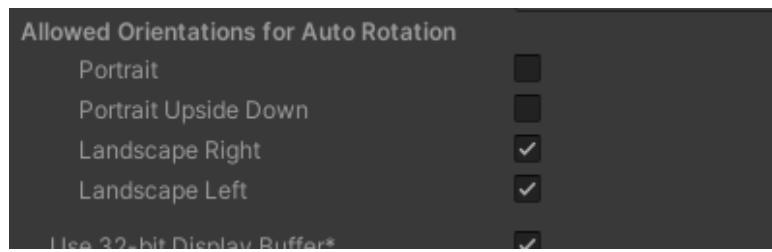


Fig. 111 Dezactivarea modului portret

După apăsarea butonului „Build” este construită aplicația în fișierul dorit fiind obținută și versiunea apk a aplicației (Figura 112). Această versiune este cea care face jocul să ruleze.

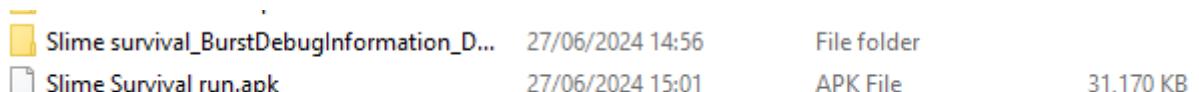


Fig. 112 Fișierele obținute după construirea aplicației

## 4.6 Contribuția personală

Majoritatea aplicației a fost construită după ce m-am instruit urmărind o listă (playlist) de tutoriale, regăsite mai jos, jocul fiind construit pentru o variantă de calculator, care se găsește pe următorul URL:

[https://youtube.com/playlist?list=PLgXA5L5ma2Bveih0btJV58REE2mzfQLOQ&si=Sk\\_gNC7yDD-SQgXpH](https://youtube.com/playlist?list=PLgXA5L5ma2Bveih0btJV58REE2mzfQLOQ&si=Sk_gNC7yDD-SQgXpH) .

Pentru a putea lansa aplicația pe mobil, au fost necesare însă câteva modificări. În ghidul urmat de mine, pentru a putea face pauză, utilizatorul apăsa tasta „Esc”, însă acest lucru nu este posibil pe mobil. Pentru a rezolva această problemă, am decis să implementez un buton (Figura 113).



Fig. 113 Butonul de pauză

Acesta este poziționat în partea dreaptă a ecranului, iar atunci când este apăsat, apelează funcția de pauză a script-ului „Game Manager”, sau continuă jocul în caz că este deja pauză (Figura 114).

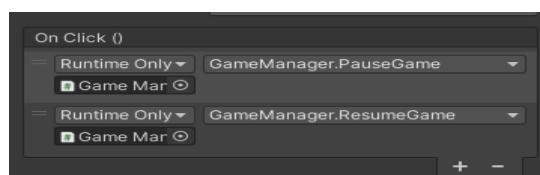


Fig. 114 On Click pentru butonul de pauză

Pe lângă aceste modificări, am făcut și implementările menționate în subcapitolul 4.5.

Deși am urmărit o listă de tutoriale, nu înseamnă că acestea erau făcute perfect. În subcapitolul 4.3.3, se regăsesc erori ce nu au fost încă rezolvate de autorul tutorialului, pe care eu le-am abordat și am reușit să le rezolv.

Chiar dacă imaginile și texturile folosite sunt resurse descărcate din magazinul Unity, tematica a fost aleasă de mine. Am vrut ca harta să fie întunecată, fundalul reprezentând interiorul unei peșteri. Inamicii care vin la început sunt slime-uri de altă culoare, personajul principal fiind unicul slime albastru.

Design-ul nivelului a fost realizat după o perioadă de testare, perioadă în care a fost verificată dificultatea jocului. Chiar dacă acesta a fost inspirat din *Vampire Survivors*, am vrut ca aplicația să fie de o dificultate diferită. Pentru a realiza acest lucru, fiecare armă a fost verificată împotriva valurilor de inamici pentru a nu fi prea puternică sau ineficace.

Am încercat să diferențiez aspectul armelor de cele originale, Figurile 115 și 116 reprezentând un astfel de exemplu.



Fig. 115 Exemplu design armă în starea 1



Fig. 116 Exemplu design armă în starea 2

Arma aceasta acoperă o zonă în jurul personajului și funcționează precum arma „Garlic” din *Vampire Survivors* (Figura 117).

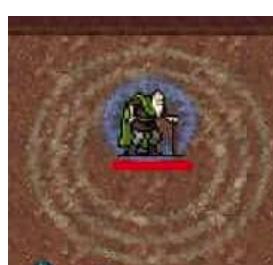


Fig. 117 Garlic din *Vampire Survivors*

Am ales să diferențiez aspectul, arma mea folosind o animație și un sistem de particule pentru a se mișca. Animația a fost descărcată din magazinul Unity, eu schimbându-i culoarea. Ea se folosește de un sistem de particule atribuit unui script ce rotește animația și o face să dispară în timp (Figura 118).

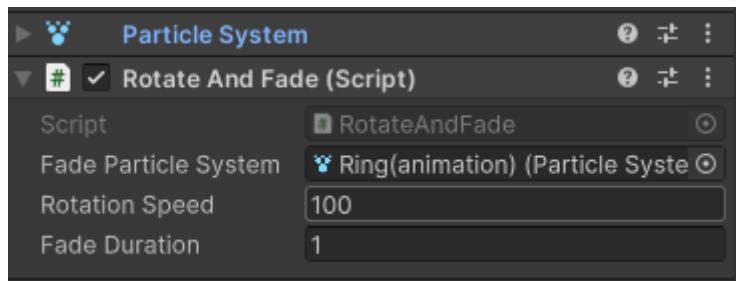


Fig. 118 Sistemul de particule din script-ul de rotație

În mod similar, fiecare echipament are un aspect diferit față de cel original, jocul ajungând să se diferențieze de cel original.

După adăugarea unui nou sistem de generare a inamicilor, am decis să implementez un tip nou de rundă. Până acum, jucătorul poate juca doar o rundă de 15 minute, aşa că am adăugat un alt mod de a încheia runda. De data aceasta, runda se termină când toate valurile au fost generate și toți inamicii au murit.

Pentru a implementa acest lucru, a fost nevoie să adaug următoarele funcții:

„AreAllWavesAndEnemiesCleared()”, ce este introdusă în clasa „SpawnManager”, având rolul de a verifica dacă au fost generate toate valurile și dacă au murit toți inamicii. Funcția „CheckWaveAndEnemyStatus” este apelată în funcția Update, a clasei „Game Manager”. Aceasta verifică condițiile de mai devreme. În acest caz, este apelat ecranul de final de rundă, jocul oprindu-se.

```
public bool AreAllWavesAndEnemiesCleared()
{
    if (currentWaveIndex < data.Length) return false; // not all of the waves ended.
    if (EnemyStats.count > 0) return false; // there are enemies left.
    return true; // all the waves ended and the enemies are dead.
}
```

```
void CheckWaveAndEnemyStatus()
{
    if (SpawnManager.instance.AreAllWavesAndEnemiesCleared())
    {
        GameOver();
    }
}
```

Pentru a-i oferi utilizatorului posibilitatea de a alege tipul de rundă dorita, am dublat scenele „Game” și „Menu” și le-am redenumit în „Timer Game” și „Timer Menu” pe cele originale, iar cele noi se numesc „Wave Count Game” și „Wave Count Menu”.

A fost introdusă și o scenă nouă numită „Round Selector”. Toate scenele prezente pot fi văzute în Figura 119.

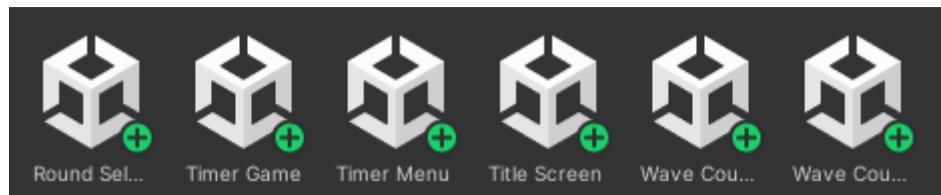


Fig. 119 Toate scenele actuale

Singura diferență între cele două scene de joc este faptul că se termină în mod diferit, dar cea recentă se termină mult mai repede, fiind generate doar 10 valuri de inamici, timp de 5 minute. Deși nu este o modificare majoră, acest lucru deschide porțile unor noi moduri de joc, precum un mod în care mobilitatea este limitată sau unul în care personajul apără o locație, nu pe el însuși.

După implementările făcute și erorile reparate, codul sursă schimbat și adăugat de mine este constituit din aprox. 350 de rânduri, acestea făcând parte din aprox. 3500 folosite pentru crearea aplicației. Iată câteva dintre implementările făcute de mine:

- Funcția SpriteDirectionChecker, din script-ul EnemiesMovement;
- Crearea script-ului EnemiesAnimation, care împreună cu funcția menționată anterior fixeaza animația inamicilor;
- Fixarea funcției ChunckChecker din script-ul MapController, reparând astfel generarea hărții;
- Crearea funcției AreAllWavesAndEnemiesCleared;
- Crearea funcției CheckWaveAndEnemyStatus, care împreună cu funcția menționată anterior ajută la crearea unui nou mod de joc.

## Capitolul 5. Ghidul utilizatorului

Pentru a putea oferi o experiență plăcută jucătorului, a fost creat și un ghid al utilizatorului. Scopul acestuia este de a face o introducere în lumea jocului, prezentând obiectivele sale. După deschiderea aplicației, utilizatorul este întâmpinat de ecranul de pornire și două butoane (Figura 120).



Fig. 120 Ecranul de pornire

La apăsarea butonului „Instructions” sunt deschise câteva instrucțiuni (Figura 121), acestea regăsindu-se pe parcursul ghidului. Pentru a închide fereastra, se apasă butonul „Close Instructions”.

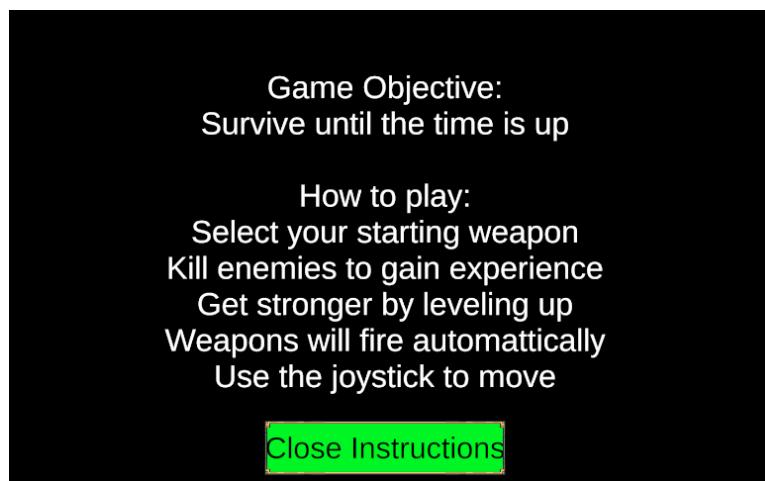


Fig. 121 Instrucțiunile jocului

Butonul „Start ” face tranziția către ecranul intermediu (Figura 122), loc în care jucătorul poate alege tipul de rundă. Timer este o rundă de 15 minute, ce se termină atunci când se scurge timpul, iar Wave Count se încheie când mor toți inamicii, durând aproximativ 5 minute.

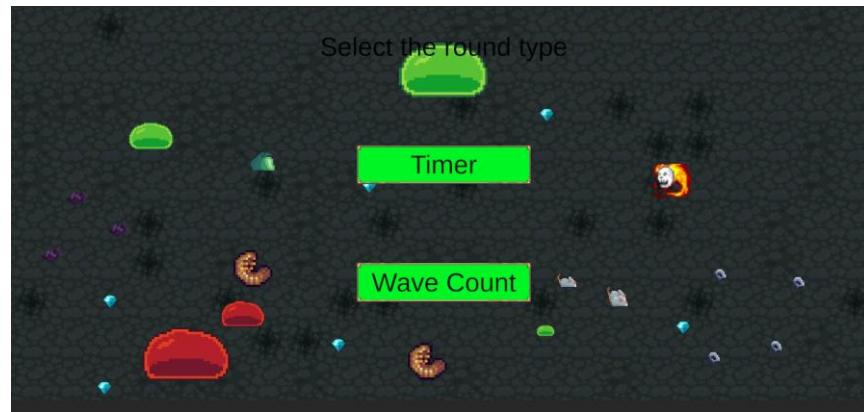


Fig. 122 Ecranul în care este selectat tipul de rundă

Apăsarea oricărui buton duce către un ecran similar, de unde jucătorul își poate alege arma de început dorită (Figura 123), rolul ecranului fiind evidențiat de textul aflat în fruntea acestuia.



Fig. 123 Ecranul în care este selectat tipul de armă

Butonul „Back” duce jucătorul la meniul anterior, iar restul butoanele nu au o descriere, pentru a încuraja utilizatorul să încerce fiecare armă, găsind astfel varianta lui preferată.

Este necesară o armă de început deoarece obiectivul jocului este de supraviețuire, iar pentru a face acest lucru, personajul trebuie să se apare în mod independent, acesta autoatacând inamicii care vin spre el.

După alegerea armei, jucătorul este întâmpinat de începerea rundei de joc (Figura 124).



Fig. 124 Începutul unei runde de joc

Sunt prezente câteva elemente, dar cel mai important este personajul principal, aflat în centrul ecranului. Acesta se poate mișca folosind joystick-ul aflat în partea stângă. Trebuie să se folosească de el pentru a evita inamicii (toate celelalte personaje prezente), coliziunea cu aceștia scăzând viața personajului principal.

În momentul în care mor, inamicii pot lăsa în urmă o recompensă (Figura 125), reprezentată de un diamant de diferite culori, în acest caz fiind albastru.

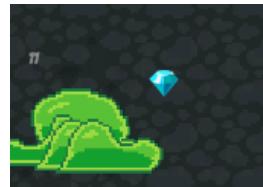


Fig. 125 Recompensă de experiență

Această recompensă, atunci când este ridicată de personaj, crește experiența acestuia, acest lucru fiind vizibil în bara de sus a ecranului, bară ce se umple cu verde în funcție de experiență acumulată. Iar cu suficientă experiență acumulată, acesta poate avansa în nivel (Figura 126), nivel afișat în colțul din dreapta sus al ecranului.

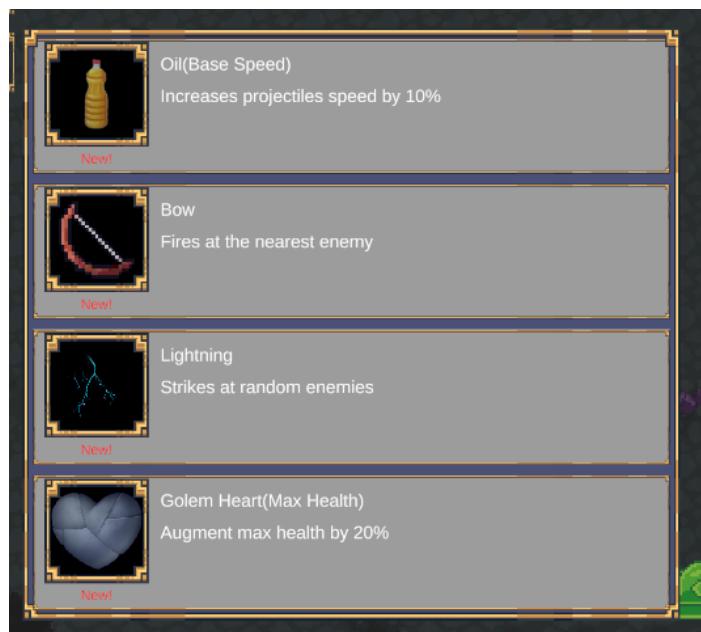


Fig. 126 Ecranul afișat când se avansează în nivel

Odată ce a avansat, utilizatorul poate alege să primească un echipament nou sau să imbunățească unul vechi, mărind nivelul acestuia. Fiecare opțiune are un nume și o descriere, acestea reprezentând ce face fiecare bucată de echipament. De asemenea poate fi vizualizată iconița reprezentativă și starea acestuia (dacă este nou sau dacă poate avansa în nivel).

Starea curentă a echipamentelor poate fi văzută în partea stângă a ecranului, în inventar (Figura 127). Aceasta reprezintă și capacitatea maximă, jucătorul putând alege maxim 6 arme și 6 echipamente pasive.



Fig. 127 Inventarul în timpul jocului

Deși nivelul maxim al armelor este 8, acestea pot evoluă dacă sunt ăndeplinite anumite condiții. Pentru a le afla, jucătorul trebuie să experimenteze, dar fiecare evoluție are câteva elemente comune :

- Armele trebuie să fie de nivel maxim;
- Este necesar un echipament pasiv specific;
- Este necesară obținerea unui cufăr.

Cufărul este o recompensă oferita atunci când un inamic special este ucis. (Figura 128)

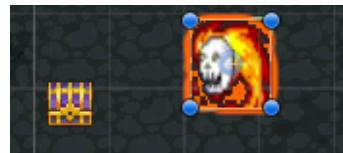


Fig. 128 Cufărul și inamicul special

În Figurile 129 și 130 poate fi observată starea armei înaintea și după obținerea cufărului.



Fig. 129 Starea armei înaintea deschiderii cufărului

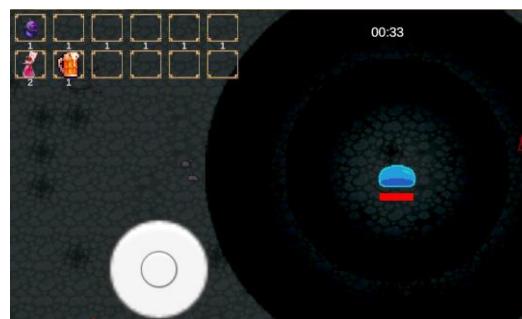


Fig. 130 Starea armei după obținerea cufărului

Pentru a pune pauză, jucătorul poate apăsa pe butonul „P”, aflat în dreapta ecranului. Acesta deschide o imagine transparentă (Figura 131) unde pot fi văzute statisticile actuale ale personajului. În acest ecran, utilizatorul poate apăsa pe butonul „Quit” pentru a ieși din runda de joc, în ecranul de început sau poate apăsa „Resume” pentru a reveni la runda actuală de joc.



Fig. 131 Ecranul de pauză

Runda se termină atunci când trec 15 minute, când mor toți inamicii, sau când personajul moare, cazuri în care utilizatorul este întâmpinat de ecranul final al rundei (Figura 132).



Fig. 132 Ecranul de final de rundă

În acest ecran, sunt prezente statistici precum: timpul supraviețuit, nivelul la care a ajuns, arma de început și inventarul final. Apăsând e butonul „Done”, jucătorul este trimis la primul ecran, de unde poate continua să rejoace în alt stil, încercând să supraviețuiască cât mai mult.

## Capitolul 6. Observații, greșeli făcute și lecții învățate

Dezvoltarea unui joc este un proces îndelungat, în special dacă dezvoltatorul nu are o experiență anterioară. Din fericire, sunt disponibile foarte multe tutoriale, acestea introducând dezvoltatorul în lumea creării de jocuri.

Motorul Unity, precum și alte unelte de dezvoltare sunt în continuă dezvoltare, însă videoclipurile vechi nu se schimbă. În cazul în care este urmărit un tutorial vechi, dezvoltatorul trebuie să decidă dacă folosește versiunea recentă a motorului sau cea din tutorial. Această decizie este influențată de experiențele acumulate și necesitățile creatorului.

Ca exemplu, versiunea folosită în tutorialul urmărit de mine este 2020.3.33f1, în timp ce versiunea mea este 2022.3.30f1. Am decis să folosesc această versiune deoarece jocul creat de mine este dezvoltat pentru mobil. O diferență semnificativă între aceste versiuni îl reprezintă uneltele folosite pentru testarea jocului. Dupa cum a fost prezentat în Figura 82, versiunile mai recente oferă varianta de a simula rularea jocului pe telefonul mobil. Testarea pe un dispozitiv scoate în evidență mici detalii, precum cel din Figura 133:



Fig. 133 Simularea unui telefon ce decupează din ecran

Se poate observa cum este „tăiată” o parte din marginea ecranului, iar acest lucru face ca nivelul personajului (dreapta sus) să nu poată fi văzut clar. Așadar, interfața trebuie adaptată pentru mai multe tipuri de ecrane, dar și testată pe cât mai multe tipuri.

Un alt lucru important îl reprezintă realizarea copiilor backup. Salvarea fișierelor este importantă deoarece, în procesul de dezvoltare, pot fi făcute greșeli, care se remediază mai rapid prin folosirea unor fișiere salvate anterior. Un backup poate fi realizat manual, dezvoltatorul salvând o copie a fișierelor proiectului sau pot fi folosite unelte ce se ocupă de acest lucru. În Unity, o astfel de unealtă este „Version Control”,

ce poate fi activată apăsând pe Window > Asset Management > Version Control (Figura 134).

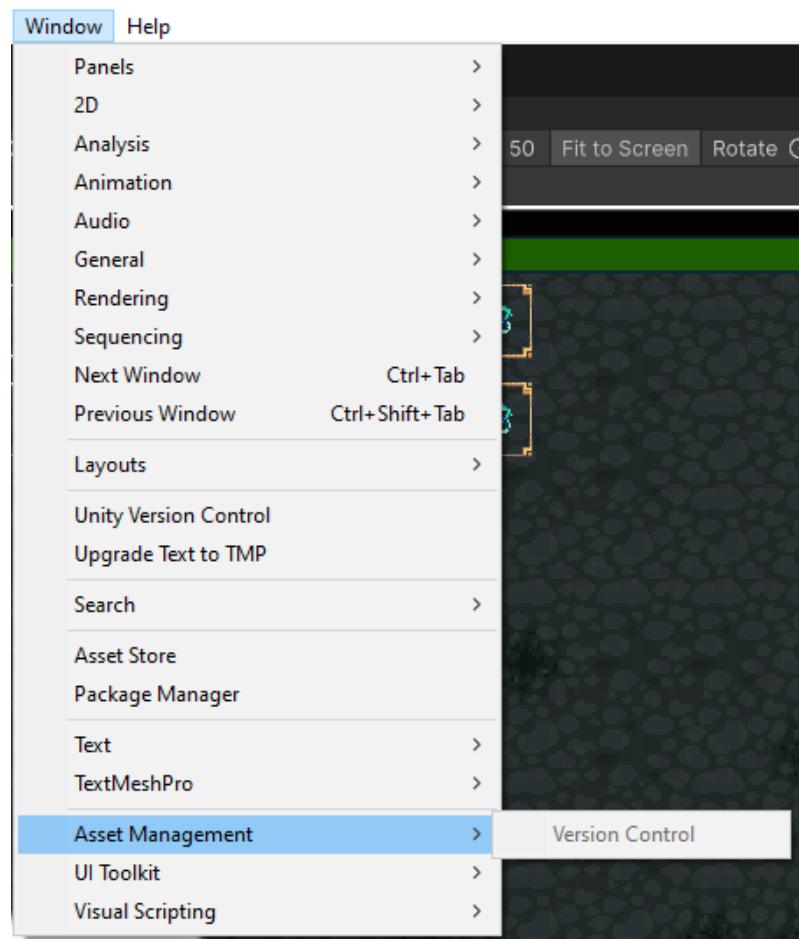


Fig. 134 Activare Version Control

În modul acesta, nu doar că fișierele sunt salvate în cloud, dar acestea pot fi și partajate cu alți utilizatori ai motorului Unity.

La începutul proiectului, nu știam de acest lucru și nici nu am făcut copii backup, lucru ce a dus la întârzierea dezvoltării jocului. În momentul în care ștergeam obiecte de joc, din neatenție, am șters scena jocului. Aceasta nu a putut fi recuperată, iar componentele ei au fost refăcute de la început.

O altă greșală făcută, încă de la început, este faptul că am nu m-am informat suficient de bine despre uneltele puse la dispoziție. Interfața a fost dezvoltată pe parcurs, iar Unity pune la dispoziție două tipuri de elemente pentru implementarea acesteia: cele TextMeshPro și cele vechi, normale.

TextMeshPro oferă o posibilitate mai mare de a diversifica aceste elemente și de a le modifica pe placul dezvoltatorului. Eu am folosit elementele vechi, conform tutorialului, însă am schimbat acest lucru pe parcursul dezvoltării. Pentru înlocuirea lor am folosit un Plugin ce schimbă toate componentele și referințele acestora în script-uri. Acesta a fost descărcat din magazinul Unity:

<https://assetstore.unity.com/packages/tools/utilities/text-to-textmesh-pro-upgrade-tool-176732>.

## **Capitolul 7. Concluzii și idei pentru dezvoltări ulterioare**

Am ales această temă datorită pasiunii mele pentru jocurile video, iar tot acest proces m-a ajutat să le înțeleg mai bine. Deși am folosit o listă de tutoriale, dezvoltarea nu a fost ușoară. A fost un proces în care am reușit să învăț cum decurge crearea unui joc, am înțeles importanța membrilor unei echipe, fiind pus în rolul multora dintre aceștia. M-am ocupat de dezvoltarea jocului, design-ul nivelelor și al interfețelor, testare și optimizare, adaptarea jocului pentru telefoanele mobile, iar acestea nu sunt toate etapele necesare lansării jocului. Nu m-am ocupat de muzica și sunetele jocului, crearea unor imagini 2D originale pentru a le folosi în joc, iar publicitatea este o altă etapă importantă pe care nu am acoperit-o.

Prin acest proces am reușit să îmi dezvolt abilitatea de organizare, învățând cum să aranjez resursele pe care le am la indemâna și ordinea în care să implementez elemente noi. Am realizat importanța testării fiecărui element, atât independent, cât și împreună cu restul jocului.

Plănuiesc să lucrez în continuare la această aplicație, revenind asupra ei când mai învăț căte ceva nou, pentru a o îmbunătăți. Unity oferă posibilitatea de a lansa jocul pe mai multe platforme fără a modifica prea multe elemente, astfel că plănuiesc să implementez abilitatea de a putea salva același progres în cadrul jocului, indiferent de platforma folosită. Lansarea jocului pe mai multe platforme este un plan de viitor, aplicație fiind făcută, momentan, pentru mobil.

Pe lângă acest lucru, vreau să creez un sistem de recompense precum cel din *Vampire Survivors*, jucătorul devenind mai puternic după ce termină o rundă. Desigur că nu pot uita de muzică, sunete și nivele noi, oferind o experiență diversificată.

Totuși înainte de a continua acest proiect sau de a crea unul nou, vreau să îmi formezi o bază mai stabilă. În urma experienței avute, aş recomanda tuturor dezvoltatorilor noi să învețe și să se informeze bine despre motorul folosit, pentru a putea beneficia în totalitate de acesta.

Pentru alte dezvoltări ulterioare, plănuiesc să cercetez mai bine sursele de inspirație, sau, dacă este o idee originală, să-mi definesc mai bine toate elementele acesteia. Am observat că întrebările „de ce” și „cum”, mă fac să realizez mai ușor rolul fiecărei componente implementate.

Jocul este în format 2D, lucru ce poate limita frumusețea grafică, deși aceasta are farmecul ei. Plănuiesc să repet proiectul în format 3D, punându-mi astfel la încercare cunoștințele dobîndite.

Forma jocului prezentată în această lucrare nu este finală, acesta urmând să evolueze odată ce eu cresc ca dezvoltator.

# Bibliografie

1. Wikipedia: Joc mobil: [https://ro.wikipedia.org/wiki/Joc\\_mobil](https://ro.wikipedia.org/wiki/Joc_mobil)
2. Wikipedia: Roguelike: <https://ro.wikipedia.org/wiki/Roguelike>
3. Wikipedia: Vampire Survivors:  
[%20Galante,for%20an%20Ultima%20Online%20server">https://en.wikipedia.org/wik/Vampire\\_Survivors#:~:text=Italian%20developer%20Luca%20"poncle">%20Galante,for%20an%20Ultima%20Online%20server](https://en.wikipedia.org/wik/Vampire_Survivors#:~:text=Italian%20developer%20Luca%20)
4. Nat Rowley, The Addictive Nature Of Vampire Survivors:  
<https://medium.com/@Nat.Rowley/the-addicting-nature-of-vampire-survivors-dc1ad4c8cf99>
5. Unity Manual: Game Objects:  
<https://docs.unity3d.com/ScriptReference/GameObject-scene.html>
6. Unity Manual: Inspector Window:  
<https://docs.unity3d.com/Manual/UsingTheInspector.html>
7. Unity Manual : Colliders:  
<https://docs.unity3d.com/560/Documentation/Manual/CollidersOverview.html>
8. Unity Manual: Rigidbody 2D:  
<https://docs.unity3d.com/560/Documentation/Manual/class-Rigidbody2D.html>
9. Unity Manual: Canvas:  
<https://docs.unity3d.com/560/Documentation/Manual/class-Canvas.html>
10. Unity Manual: Event System:  
<https://docs.unity3d.com/560/Documentation/Manual/EventSystem.html>
11. Unity Manual: Sprites:  
<https://docs.unity3d.com/560/Documentation/Manual/Sprites.html>
12. Unity Manual: Sprite Renderer:  
<https://docs.unity3d.com/560/Documentation/Manual/classSpriteRenderer.html>
13. Unity Manual: Transform:  
<https://docs.unity3d.com/560/Documentation/Manual/class-Transform.html>
14. Unity Manual: The Animator Window:  
<https://docs.unity3d.com/560/Documentation/Manual/AnimatorWindow.html>

15. Unity Manual: Prefabs:

<https://docs.unity3d.com/560/Documentation/Manual/Prefabs.html>

16. Unity Manual: ScriptableObjects:  
<https://docs.unity3d.com/560/Documentation/Manual/class-ScriptableObject.html>

17. Unity Asset Store: Slime Enemy – Pixel Art:

18. <https://assetstore.unity.com/packages/2d/characters/slime-enemy-pixel-art-228568>

19. Creating a Rogue-like (like Vampire Survivors) in Unity — Part 1: Movement and Camera, 2022:

<https://youtu.be/EIJk5KYzSJM?si=domNNZrhI0JNVoMo> accesat: 06/06/2024:

20. Creating a Rogue-like (like Vampire Survivors) in Unity — Part 2: Map Generation, 2022: <https://youtu.be/QN8dm0RD3mY?si=cFcL6xtCrTaz5L7> accesat: 06/06/2024

21. Creating a Rogue-like (like Vampire Survivors) in Unity — Part 7: Enemy Spawning, 2023 <https://youtu.be/h2cg4ucDuWw?si=PvJfuemFnrYc0kd> accesat: 07/06/2024

22. Creating a Rogue-like (like Vampire Survivors) in Unity — Part 5: Characters and Pick-ups, 2023: <https://youtu.be/qREiQ5vSAng?si=Yk0QTTNQCjU9Tb3P> accesat: 08/06/2024

23. Fixing Pickup Collection — Creating a Rogue-like (like Vampire Survivors) in Unity: Part 17, 2024: [https://youtu.be/iiviXsCjHfc?si=D4MftMVg6\\_oPB70w](https://youtu.be/iiviXsCjHfc?si=D4MftMVg6_oPB70w) accesat: 09/06/2024

24. Modularising the Stat System — Creating a Rogue-like (like Vampire Survivors) in Unity: Part 18, 2024:

<https://youtu.be/0J0Jmc2IZYA?si=Mg2U8zoKY9U-mbVS> accesat: 10/06/2024

25. Creating a Rogue-like (like Vampire Survivors) in Unity — Part 3: Weapons and Enemy AI, 2022: <https://youtu.be/RCOxhTsbAWo?si=aNMiWGEXN0kwlbSi> accesat: 11/06/2024

26. Revamped Weapon System — Creating a Rogue-like (like Vampire Survivors) in Unity: Part 15, 2024 :

<https://youtu.be/bbk7OPyFU?si=gFDRDoTF33shfnpZ> accesat: 12/06/2024

27. Unity Asset Store: Virtual Joystick Pack:

<https://assetstore.unity.com/packages/tools/utilities/virtual-joystick-pack-261317>

28. Make a Rogue-like (like Vampire Survivors) in Unity, 2022

<https://youtube.com/playlist?list=PLgXA5L5ma2Bveih0btJV58REE2mzfQLOQ&si=PYnu2Mq8ipqc2icP> accesat: 15/06/2024

29. Creating a Rogue-like (like Vampire Survivors) in Unity — Part 11: Code and Aesthetic Touch-ups, 2023:

[https://youtu.be/4U\\_f\\_qjOpZE?si=-CKi7gdiEqHjhlnL](https://youtu.be/4U_f_qjOpZE?si=-CKi7gdiEqHjhlnL) accesat: 16/06/2024

30. Text to TextMesh Pro Upgrade Tool:

<https://assetstore.unity.com/packages/tools/utilities/text-to-textmesh-pro-upgrade-tool-176732>

# Anexă

## A) Cod sursă

### Player Movement

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Terresquall;
public class PlayerMovement : MonoBehaviour
{
    public const float DEFAULT_MOVESPEED = 5f;
    //Movement
    [HideInInspector]
    public Vector2 moveDir;
    [HideInInspector]
    public float lastHorizontalVector;
    [HideInInspector]
    public float lastVerticalVector;
    [HideInInspector]
    public Vector2 lastMovedVector;
    Rigidbody2D rb;
    PlayerStats player;
    // Start is called before the first frame update
    void Start()
    {
        player = GetComponent<PlayerStats>();
        rb = GetComponent<Rigidbody2D>();
        lastMovedVector = new Vector2(1, 0f); // for when the game starts and the
        player doesn't move, so the weapon will have momentum
    }
    // Update is called once per frame
    void Update()
    {
        InputManagement();
    }
    void FixedUpdate()
    {
        Move();
    }
    void InputManagement()
    {
        if (GameManager.instance.isGameOver)
        {
            return;
        }
        float moveX, moveY;
        if (VirtualJoystick.CountActiveInstances() > 0 )
        {
            moveX = VirtualJoystick.GetAxisRaw("Horizontal");
            moveY = VirtualJoystick.GetAxisRaw("Vertical");
        }
        else
        {
            moveX = Input.GetAxisRaw("Horizontal");
            moveY = Input.GetAxisRaw("Vertical");
        }

        moveDir = new Vector2(moveX, moveY).normalized;

        if (moveDir.x !=0 )
        {
            lastHorizontalVector = moveDir.x;
            lastMovedVector = new Vector2(lastHorizontalVector, 0f); //Last moved x
        }
        if (moveDir.y != 0)
```

```

        {
            lastVerticalVector = moveDir.y;
            lastMovedVector = new Vector2(0f, lastVerticalVector); //last moved y
        }
        if(moveDir.x != 0 && moveDir.y != 0)
        {
            lastMovedVector = new Vector2(lastHorizontalVector,
lastVerticalVector); // while moving
        }
    }
    void Move()
    {
        if (GameManager.instance.isGameOver)
        {
            return;
        }
        rb.velocity = moveDir * DEFAULT_MOVESPEED * player.Stats.moveSpeed;
    }
}

```

## Camera Movement

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class CameraMovement : MonoBehaviour
{
    public Transform target;
    public Vector3 offset;
    // Update is called once per frame
    void Update()
    {
        transform.position = target.position + offset ;
    }
}

```

## Map Controller

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class MapController : MonoBehaviour
{
    public List<GameObject> terrainChuncks;
    public GameObject player;
    public float checkerRadius;
    public LayerMask terrainMask;
    public GameObject currentChunk;
    Vector3 playerLastPosition;
    [Header("Optimization")]
    public List<GameObject> spawnedChunks;
    GameObject latestChunk;
    public float maxOpDist; //Must be greater than the length and width of the tilemap
    float opDist;
    float optimizerCooldown;
    public float optimizerCooldownDur;
    // Start is called before the first frame update
    void Start()
    {
        playerLastPosition = player.transform.position;
    }
    // Update is called once per frame
    void Update()
    {
        ChunkChecker();
        ChunkOptimizer();
    }
    void ChunkChecker()
    {

```

```

        if (!currentChunk)
        {
            return;
        }
        Vector3 moveDir = player.transform.position - playerLastPosition;
        playerLastPosition = player.transform.position;
        string directionName = GetDirectionName(moveDir);
        CheckAndSpawnChunk(directionName);
        // check additional adjacent directions for diagonal chunks
        if (directionName.Contains("Up"))
        {
            CheckAndSpawnChunk("Up");
            CheckAndSpawnChunk("Right");
            CheckAndSpawnChunk("Left");
        }
        if (directionName.Contains("Down"))
        {
            CheckAndSpawnChunk("Down");
            CheckAndSpawnChunk("Right");
            CheckAndSpawnChunk("Left");
        }
        if (directionName.Contains("Right"))
        {
            CheckAndSpawnChunk("Right");
            CheckAndSpawnChunk("Up");
            CheckAndSpawnChunk("Down");
        }
        if (directionName.Contains("Left"))
        {
            CheckAndSpawnChunk("Left");
            CheckAndSpawnChunk("Up");
            CheckAndSpawnChunk("Down");
        }
    }
    void CheckAndSpawnChunk(string direction)
    {
        if (!Physics2D.OverlapCircle(currentChunk.transform.Find(direction).position,
        checkerRadius, terrainMask))
        {
            SpawnChunk(currentChunk.transform.Find(direction).position);
        }
    }
    string GetDirectionName(Vector3 direction)
    {
        direction = direction.normalized;
        if (Mathf.Abs(direction.x) > Mathf.Abs(direction.y))
        {
            // moving horizontally more than vertically
            if (direction.y > 0.5f)
            {
                //moving upwards
                return direction.x > 0 ? "Right Up" : "Left Up";
            }
            else if (direction.y < -0.5f)
            {
                //moving downwards
                return direction.x > 0 ? "Right Down" : "Left Down";
            }
            else
            {
                //moving horizontally
                return direction.x > 0 ? "Right" : "Left";
            }
        }
        else
        {
            // moving vertically more than horizontally
            if (direction.x > 0.5f)
            {
                //moving upwards
                return direction.y > 0 ? "Right Up" : "Right Down";
            }

```

```

        else if (direction.x < -0.5f)
    {
        //moving downwards
        return direction.y > 0 ? "Left Up" : "Left Down";
    }
    else
    {
        //moving straight vertically
        return direction.y > 0 ? "Up" : "Down";
    }
}
void SpawnChunk(Vector3 spawnPosition)
{
    int rand = Random.Range(0, terrainChuncks.Count);
    latestChunk      = Instantiate(terrainChuncks[rand],           spawnPosition,
Quaternion.identity);
    spawnedChunks.Add(latestChunk);
}
void ChunkOptimizer()
{
    optimizerCooldown -= Time.deltaTime;
    if(optimizerCooldown <= 0f)
    {
        optimizerCooldown = optimizerCooldownDur;
    }
    else
    {
        return;
    }

    foreach(GameObject chunk in spawnedChunks)
    {
        opDist          = Vector3.Distance(player.transform.position,
chunk.transform.position);
        if(opDist > maxOpDist)
        {
            chunk.SetActive(false);
        }
        else
        {
            chunk.SetActive(true);
        }
    }
}
}

Player Stats
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using TMPro;
using System;
public class PlayerStats : MonoBehaviour
{
    CharacterData characterData;
    public CharacterData.Stats baseStats;
    [SerializeField] CharacterData.Stats actualStats;
    public CharacterData.Stats Stats
    {
        get { return actualStats; }
        set
        {
            actualStats = value;
        }
    }
    // current stats
    float health;

    public float CurrentHealth

```

```

    {
        get { return health; }
        //if we try and set the current health, the UI interface on the pause screen
        will also be updated
        set
        {
            // check if the value has changed
            if( health != value)
            {
                health = value;
                UpdateHealthBar();
            }
        }
    }
    [Header("Visuals")]
    public ParticleSystem damageEffect;//if damage is dealt
    public ParticleSystem blockedEffect;//if damage is blocked completely
    //Experience and level of the player
    [Header("Experience/Level")]
    public int experience = 0;
    public int level = 1;
    public int experienceCap;
    //Class for defining a level range and the corresponding experience cap increase
    for that range
    [System.Serializable]
    public class LevelRange
    {
        public int startLevel;
        public int endLevel;
        public int experienceCapIncrease;
    }
    //I-Frames
    [Header("I-Frames")]
    public float invincibilityDuration;
    float invincibilityTimer;
    bool isInvincible;
    public List<LevelRange> levelRanges;
    PlayerCollector collector;
    PlayerInventory inventory;
    public int weaponIndex;
    public int passiveItemIndex;
    [Header("UI")]
    public Image healthBar;
    public Image expBar;
    public TMP_Text levelText;
    void Awake()
    {
        characterData = CharacterSelector.GetData();
        if(CharacterSelector.instance)
            CharacterSelector.instance.DestroySingleton();
        inventory = GetComponent<PlayerInventory>();
        collector = GetComponentInChildren<PlayerCollector>();
        //Assign the variables
        baseStats = actualStats = characterData.stats;
        collector.SetRadius(actualStats.magnet);
        health = actualStats.maxHealth;
    }
    void Start()
    {
        //spawn the starting weapon
        inventory.Add(characterData.StartingWeapon);
        //Initialize the experience cap as the first experience cap increase
        experienceCap = levelRanges[0].experienceCapIncrease;
        GameManager.instance.AssignChosenCharacterUI(characterData);
        UpdateHealthBar();
        UpdateExpBar();
        UpdateLevelText();
    }
    void Update()
    {
        if(invincibilityTimer > 0)

```

```

        {
            invincibilityTimer -= Time.deltaTime;
        }
        else if (isInvincible)
        {
            isInvincible = false;
        }
        Recover();
    }
    public void RecalculateStats()
    {
        actualStats = baseStats;
        foreach(PlayerInventory.Slot s in inventory.passiveSlots)
        {
            Passive p = s.item as Passive;
            if(p)
            {
                actualStats += p.GetBoosts();
            }
        }
        collector.SetRadius(actualStats.magnet);
    }
    public void IncreaseExperience(int amount)
    {
        experience += amount;
        LevelUpChecker();
        UpdateExpBar();
    }
    void LevelUpChecker()
    {
        if (experience >= experienceCap)
        {
            level++;
            experience -= experienceCap;

            int experienceCapIncrease = 0;
            foreach(LevelRange range in levelRanges)
            {
                if(level >= range.startLevel && level <= range.endLevel)
                {
                    experienceCapIncrease = range.experienceCapIncrease;
                    break;
                }
            }
            experienceCap += experienceCapIncrease;
            UpdateLevelText();
            GameManager.instance.StartLevelUp();
            if (experience >= experienceCap)
                LevelUpChecker();
        }
    }
    void UpdateExpBar()
    {
        expBar.fillAmount = (float)experience / experienceCap;
    }
    void UpdateLevelText()
    {
        levelText.text = "Level: " + level.ToString();
    }
    public void TakeDamage(float dmg)
    {
        // If the player is not invincible reduce health and start invincibility timer
        if (!isInvincible)
        {
            // take armor into account before taking damage
            dmg -= Stats.armor;
            if(dmg > 0)
            {
                //deal damage
                CurrentHealth -= dmg;

                // if there is a damage effect, apply it
            }
        }
    }
}

```

```

        if (damageEffect)           Destroy(Instantiate(damageEffect,
transform.position, Quaternion.identity), 5f);
        if (CurrentHealth <= 0)
        {
            Kill();
        }
    else
    {
        // play blocked effect
        if (blockedEffect)         Destroy(Instantiate(blockedEffect,
transform.position, Quaternion.identity), 5f);
    }
    invincibilityTimer = invincibilityDuration;
    isInvincible = true;
    UpdateHealthBar();
}
void UpdateHealthBar()
{
    healthBar.fillAmount = CurrentHealth / actualStats.maxHealth;
}
public void Kill()
{
    if (!GameManager.instance.isGameOver)
    {
        GameManager.instance.AssignLevelReachedUI(level);
        GameManager.instance.GameOver();
    }
}
public void RestoreHealth(float amount)
{
    //Heal only if the health is lower than maximum amount
    if (CurrentHealth < actualStats.maxHealth)
    {
        CurrentHealth += amount;
        if (CurrentHealth > actualStats.maxHealth)
        {
            CurrentHealth = actualStats.maxHealth;
        }
        UpdateHealthBar();
    }
}
void Recover()
{
    if (CurrentHealth < actualStats.maxHealth)
    {
        CurrentHealth += Stats.recovery * Time.deltaTime;
        //To make sure that the player doesn't overheat
        if (CurrentHealth > actualStats.maxHealth)
        {
            CurrentHealth = actualStats.maxHealth;
        }
        UpdateHealthBar();
    }
}
}
}

```

## Item Data

```

using UnityEngine;
public abstract class ItemData : ScriptableObject
{
    public Sprite icon;
    public int maxLevel;
    [System.Serializable]
    public struct Evolution
    {
        public string name;
        public enum Condition { auto, treasureChest }
    }
}

```

```

        public Condition condition;
        [System.Flags]public enum Consumption { passives = 1, weapons = 2}
        public Consumption consumes;
        public int evolutionLevel;
        public Config[] catalysts;
        public Config outcome;
        [System.Serializable]
        public struct Config
        {
            public ItemData itemType;
            public int level;
        }
    }
    public Evolution[] evolutionData;
    public abstract Item.LevelData GetLevelData(int level);
}

```

## Scene Controller

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
public class SceneController : MonoBehaviour
{
    public void SceneChange(string name)
    {
        SceneManager.LoadScene(name);
        Time.timeScale = 1;
    }
}

```

## Game Manager

```

using System.Collections;
using System.Collections.Generic;
using System.Xml.Serialization;
using UnityEngine;
using UnityEngine.UI;
using TMPro;
using System;

public class GameManager : MonoBehaviour
{
    public static GameManager instance;
    // defining the different states of the game
    public enum GameState
    {
        Gameplay,
        Paused,
        GameOver,
        LevelUp
    }
    // store the current state of the game
    public GameState currentState;
    //store the previous state of the game
    public GameState previousState;
    public event Action<GameState> OnGameStateChanged;

    [Header("Damage Text Settings")]
    public Canvas damageTextCanvas;
    public float textFontSize = 20;
    public TMP_FontAsset textFont;
    public Camera referenceCamera;

    [Header("Screens")]
    public GameObject pauseScreen;
    public GameObject resultsScreen;
    public GameObject levelUpScreen;
    int stackedLevelUps = 0;

    [Header("Current Stat Display")]
    public TMP_Text currentHealthDisplay;
    public TMP_Text currentRecoveryDisplay;
    public TMP_Text currentMoveSpeedDisplay;

```

```

public TMP_Text currentMightDisplay;
public TMP_Text currentProjectileSpeedDisplay;
public TMP_Text currentMagnetDisplay;

[Header("Results Screen Displays")]
public Image chosenCharacterImage;
public TMP_Text chosenCharacterName;
public TMP_Text levelReachedDisplay;
public TMP_Text timeSurvivedDisplay;

[Header("StopWatch")]
public float timelimit;// the time limit in seconds
float stopwatchTime; // the current time elapsed since the stopwatch started
public TMP_Text stopwatchDisplay;

// reference to the player's Gameobject
public GameObject playerObject;
public bool isGameOver { get { return currentState == GameState.GameOver; } }
public bool choosingUpgrade { get { return currentState == GameState.LevelUp; } }
// Gives us the time since the level has started.
public float GetElapsed Time() { return stopwatchTime; }
void Awake()
{
    //warning check to see if there is another singleton of this kind in the game
    if(instance == null)
    {
        instance = this;
    }
    else
    {
        Debug.LogWarning("EXTRA" + this + "DELETED");
        Destroy(gameObject);
    }
    DisableScreens();
}
void Update()
{
    // define the behavior for each state
    switch(currentState)
    {
        case GameState.Gameplay:
            // code for the gameplay state
            CheckForPauseAndResume();
            UpdateStopwatch();
            CheckWaveAndEnemyStatus();
            break;
        case GameState.Paused:
            //code for the paused state
            CheckForPauseAndResume();
            break;
        case GameState.GameOver:
        case GameState.LevelUp:
            // code for levelUp state
            break;
        default:
            Debug.LogWarning("STATE DOES NOT EXIST");
            break;
    }
}
void CheckWaveAndEnemyStatus()
{
    if (SpawnManager.instance.AreAllWavesAndEnemiesCleared())
    {
        GameOver();
    }
}
IEnumerator GenerateFloatingTextCoroutine(string text, Transform target, float duration =
1f, float speed = 50f)
{
    // start generating the floating text
    GameObject textObj = new GameObject("Damage Floating Text");
    RectTransform rect = textObj.AddComponent<RectTransform>();
    TextMeshProUGUI tmPro = textObj.AddComponent<TextMeshProUGUI>();
    tmPro.text = text;
    tmPro.horizontalAlignment = HorizontalAlignmentOptions.Center;
    tmPro.verticalAlignment = VerticalAlignmentOptions.Middle;
    tmPro.fontSize = textFontSize;
    if (textFont) tmPro.font = textFont;
    rect.position = referenceCamera.WorldToScreenPoint(target.position);
}

```

```

        Destroy(textObj, duration);

        // parent the generated text object to the canvas
        textObj.transform.SetParent(instance.damageTextCanvas.transform);
        textObj.transform.SetSiblingIndex(0);

        //pan the text upwards and fade it over time
        WaitForEndOfFrame w = new WaitForEndOfFrame();
        float t = 0;
        float yOffset = 0;
        Vector3 lastKnownPosition = target.position;
        while (t < duration)
        {
            // if the rect object is missing
            if (!rect) break;
            // fade the text to the right alpha value
            tmPro.color = new Color(tmPro.color.r, tmPro.color.g, tmPro.color.b, 1 - t / duration);
            // if target exists save it's position
            if (target)
                lastKnownPosition = target.position;
            //pan the text upwards
            yOffset += speed * Time.deltaTime;
            rect.position = referenceCamera.WorldToScreenPoint(lastKnownPosition + new Vector3(0, yOffset));
            // wait for a frame and update time
            yield return w;
            t += Time.deltaTime;
        }
    }

    public static void GenerateFloatingText(string text, Transform target, float duration = 1f, float speed = 1f)
    {
        // if the canvas is not set end the function
        if (!instance.damageTextCanvas)
        {
            return;
        }
        // find a relevant camera to convert the world position to a screen position
        if (!instance.referenceCamera)
        {
            instance.referenceCamera = Camera.main;
        }
        instance.StartCoroutine(instance.GenerateFloatingTextCoroutine(text, target, duration, speed));
    }

    // define the method to change the state of the game
    public void ChangeState(GameState newState)
    {
        previousState = currentState;
        currentState = newState;
        OnGameStateChanged?.Invoke(newState);
    }

    public void PauseGame()
    {
        if (currentState != GameState.Paused)
        {
            ChangeState(GameState.Paused);
            Time.timeScale = 0f; // stop the game
            pauseScreen.SetActive(true);
        }
    }

    public void ResumeGame()
    {
        if (currentState == GameState.Paused)
        {
            ChangeState(previousState);
            Time.timeScale = 1f; // resume the game
            pauseScreen.SetActive(false);
        }
    }

    // method to check for pause and resume input
    void CheckForPauseAndResume()
    {
        if (Input.GetKeyDown(KeyCode.Escape))
        {
            if (currentState == GameState.Paused)
            {
                ResumeGame();
            }
        }
    }
}

```

```

        else
        {
            PauseGame();
        }
    }
void DisableScreens()
{
    pauseScreen.SetActive(false);
    resultsScreen.SetActive(false);
    levelUpScreen.SetActive(false);
}

public void GameOver()
{
    Time.timeScale = 0f;
    timeSurvivedDisplay.text = stopwatchDisplay.text;
    ChangeState(GameState.GameOver);
    DisplayResults();
}
void DisplayResults()
{
    resultsScreen.SetActive(true);
}

public void AssignChosenCharacterUI(CharacterData chosenCharacterData)
{
    chosenCharacterImage.sprite = chosenCharacterData.Icon;
    chosenCharacterName.text = chosenCharacterData.name;
}
public void AssignLevelReachedUI(int levelReachedData)
{
    levelReachedDisplay.text = levelReachedData.ToString();
}

void UpdateStopwatch()
{
    stopwatchTime += Time.deltaTime;

    UpdateStopwatchDisplay();

    if(stopwatchTime >= timeLimit)
    {
        playerObject.SendMessage("Kill");
    }
}
void UpdateStopwatchDisplay()
{
    // calculate the number of minutes and seconds that elapsed
    int minutes = Mathf.FloorToInt(stopwatchTime / 60);
    int seconds = Mathf.FloorToInt(stopwatchTime % 60);
    //update the stopwatch text to display the elapsed time
    stopwatchDisplay.text = string.Format("{0:00}:{1:00}", minutes, seconds);
}
public void StartLevelUp()
{
    ChangeState(GameState.LevelUp);
    //if the level up screen is active , make a record of it
    if (levelUpScreen.activeSelf) stackedLevelUps++;
    else
    {
        Time.timeScale = 0f;
        levelUpScreen.SetActive(true);
        playerObject.SendMessage("RemoveAndApplyUpgrades");
    }
}
public void EndLevelUp()
{
    Time.timeScale = 1f; // resume the game
    levelUpScreen.SetActive(false);
    ChangeState(GameState.Gameplay);

    if(stackedLevelUps > 0)
    {
        stackedLevelUps--;
        StartLevelUp();
    }
}
}

```

## B) Listă de figuri

Fig. 1 Jocul Snake pe telefonul Nokia .....	9
Fig. 2 Jocul Angry Birds.....	9
Fig. 3 Google Play Pass .....	10
Fig. 4 Jocul Rogue 1980.....	11
Fig. 5 Jocul The Binding of Isaac (2011) .....	12
Fig. 6 Jocul Hades (2020) .....	12
Fig. 7 Scena Game și obiectele sale de joc.....	14
Fig. 8 Circle Collider 2D.....	15
Fig. 9 Rigidbody 2D .....	15
Fig. 10 Obiectul Canvas și obiectele derivate din el (copiii săi) .....	16
Fig. 11 Aspectul obiectului Canvas în scena de joc .....	16
Fig. 12 Slime Sprite .....	16
Fig. 13 Slime Spritesheet .....	17
Fig. 14 Tile Pallet-ul folosit pentru crearea fundalului hărții .....	17
Fig. 15 Componența Transform.....	18
Fig. 16 Jocul Vampire Survivors.....	19
Fig. 17 Unity Hub.....	20
Fig. 18 Alegerea şablonului .....	21
Fig. 19 Fișierele principale folosite pentru organizare .....	21
Fig. 20 Inspector listă de imagini .....	22
Fig. 21 Sprite Editor Slice .....	23
Fig. 22 Sprite Editor Apply .....	23
Fig. 23 Fereastra Inspector a obiectului Player .....	23
Fig. 24 Resetare componență Transform .....	24
Fig. 25 Controller de clip și animație .....	24
Fig. 26 Sprite renderer pentru Player .....	24
Fig. 27 Fereastra Animator pentru Player.....	25
Fig. 28 Fereastra Animation .....	25
Fig. 29 Fereastra Animator .....	25
Fig. 30 Acțiunea PlayerWalk .....	26
Fig. 31 Setări gravitație.....	26
Fig. 32 Partea legată de mișcare din Inspector a obiectului Player .....	27
Fig. 33 Script-ul Camera Movement .....	28
Fig. 34 Fundalul unei bucăți de hartă .....	28
Fig. 35 Obstacol cu un Polygon Collider.....	29
Fig. 36 Script-ul Prop Randomizer .....	29
Fig. 37 Obiectul Map Controller.....	30
Fig. 38 Stratul pentru obiecte .....	30
Fig. 39 Harta inițială .....	31
Fig. 40 Harta la o distanță mare față de punctul inițial.....	31
Fig. 41 Inamici .....	32

Fig. 42 Enemy Spawner .....	32
Fig. 43 Fereastra Inspector pentru un şablon de inamic .....	34
Fig. 44 Fereastra Inspector pentru o recompensă de experiență .....	35
Fig. 45 Diagrama ierarhiei claselor de echipament .....	36
Fig. 46 Crearea unei arme .....	38
Fig. 47 Alegerea Comportamentului unei arme .....	38
Fig. 48 Fereastra Inspector a unui obiect de tip armă .....	39
Fig. 49 Şablon pentru un obiect de tip armă .....	39
Fig. 50 Fereastra Inspector a unui echipament pasiv .....	40
Fig. 51 Obiectul scriptabil Sword .....	41
Fig. 52 Script-ul Enemy Stats .....	42
Fig. 53 Statisticile armei Ring .....	43
Fig. 54 Statisticile pasive .....	44
Fig. 55 Condițiile nivelerelor .....	46
Fig. 56 Alegerea echipamentelor în fereastra Level up .....	47
Fig. 57 Evolution Data pentru armă .....	48
Fig. 58 Evolution Data pentru echipament pasiv .....	48
Fig. 59 Scena Title Screen .....	50
Fig. 60 Acțiunea On Click pentru Start .....	50
Fig. 61 Aspectul butoanelor Start și Instructions .....	50
Fig. 62 Acțiunea On Click pentru instructions .....	51
Fig. 63 Imaginea Instructions .....	51
Fig. 64 Aspectul scenei Title Screen .....	51
Fig. 65 Obiectele scenei Menu .....	52
Fig. 66 Acțiunile butonului Sword .....	52
Fig. 67 Script-ul Game Manager .....	53
Fig. 68 Ecranul Level Up și componentele acestuia .....	54
Fig. 69 Interfața de Level Up .....	54
Fig. 70 Upgrade Option .....	55
Fig. 71 Ecranul de pauză .....	55
Fig. 72 Componentele Pause Screen .....	55
Fig. 73 Pânza scenei Game .....	56
Fig. 74 Interfața normală .....	56
Fig. 75 Timpul în Game Manager .....	57
Fig. 76 Interfața cu rezultatul final .....	57
Fig. 77 Componentele ecranului final .....	58
Fig. 78 Consola Unity .....	58
Fig. 79 Consola pentru arme atribuite .....	59
Fig. 80 Butonul Play și fereastra Game .....	59
Fig. 81 Selectarea rezoluției .....	60
Fig. 82 Selectarea Dispozitivului .....	60
Fig. 83 Buton de Level up .....	61
Fig. 84 Cufăr în scena jocului .....	61
Fig. 85 Debug Inspector .....	62
Fig. 86 Statistici ascunse .....	62

Fig. 87 Eroare animație mișcare.....	63
Fig. 88 Eroare la generarea hărții.....	63
Fig. 89 Enemy Spawner marcat ca Obsolete .....	64
Fig. 90 Val de inamici în sistemul vechi.....	65
Fig. 91 Ierarhie de clase pentru a genera inamici, obținută din tutorial .....	66
Fig. 92 Spawn Manager și Event Manager.....	66
Fig. 93 Primul val de inamici.....	66
Fig. 94 Val de inamici de tip grup .....	67
Fig. 95 Obiect scriptabil vechi pentru armă .....	68
Fig. 96 Controller vechi de arme .....	68
Fig. 97 Şablon armă veche.....	69
Fig. 98 Inventarul vechi .....	69
Fig. 99 Inventarul nou.....	70
Fig. 100 Pânza scenei cu joystick .....	71
Fig. 101 Script-ul joystick-ului.....	71
Fig. 102 Joystick-ul pe ecran.....	71
Fig. 103 Rect Transform.....	72
Fig. 104 Colțurile din Rect Transform.....	73
Fig. 105 Ecran cu rezoluție mare.....	73
Fig. 106 Ecran cu rezoluție mai mică .....	74
Fig. 107 Butoanele File și Build Settings .....	74
Fig. 108 Fereastra Build Settings .....	75
Fig. 109 Butonul Switch Platform .....	75
Fig. 110 Implementarea iconiței de joc .....	75
Fig. 111 Dezactivarea modului portret.....	76
Fig. 112 Fișierele obținute după construirea aplicației.....	76
Fig. 113 Butonul de pauză.....	76
Fig. 114 On Click pentru butonul de pauză .....	76
Fig. 115 Exemplu design armă în starea 1 .....	77
Fig. 116 Exemplu design armă în starea 2 .....	77
Fig. 117 Garlic din Vampire Survivors .....	77
Fig. 118 Sistemul de particule din script-ul de rotație .....	78
Fig. 119 Toate scenele actuale .....	79
Fig. 120 Ecranul de pornire .....	80
Fig. 121 Instrucțiunile jocului .....	80
Fig. 122 Ecranul în care este selectat tipul de rundă.....	81
Fig. 123 Ecranul în care este selectat tipul de armă .....	81
Fig. 124 Începutul unei runde de joc .....	81
Fig. 125 Recompensă de experiență.....	82
Fig. 126 Ecranul afișat când se avansează în nivel.....	82
Fig. 127 Inventarul în timpul jocului .....	83
Fig. 128 Cufărul și inamicul special .....	83
Fig. 129 Starea armei înaintea deschiderii cufărului.....	83
Fig. 130 Starea armei după obținerea cufărului.....	83
Fig. 131 Ecranul de pauză.....	84

Fig. 132 Ecranul de final de rundă.....	84
Fig. 133 Simularea unui telefon ce decupează din ecran .....	85
Fig. 134 Activare Version Control.....	86