



UNIVERSITATEA DIN BUCUREȘTI

**FACULTATEA DE
MATEMATICĂ ȘI
INFORMATICĂ**



SPECIALIZAREA INFORMATICĂ

Lucrare de licență

The Knight

Absolvent

Mihai Lazăr

Coordonator științific

Lect.dr. Ștefan George Popescu

București, iulie 2023

Rezumat

În această lucrare, voi explora dezvoltarea unui joc video folosind motorul Unity, lansat în anul 2005, la Apple Worldwide Developers, de către dezvoltatorul Unity Technologies, din punct de vedere al designului dar și din punct de vedere istoric. Totodată voi explora și relevanța jocurilor video în industrie precum și impactul lor asupra culturii și societății. Componentele principale pe care le voi dezvolta vor fi: jocul propriu-zis, cum ar fi caracterele și inamicii, sisteme de meniuri, audio și de salvare.

Abstract

In this paper, I will explore the development of a video game using the Unity engine, released in 2005, at the Apple Worldwide Developers Conference by Unity Technologies, from both a design and historical perspective. I will also delve into the relevance of video games in the industry, as well as their impact on culture and society. The main components I will develop include: the actual game, such as characters and enemies, menu systems, audio, and saving mechanisms.

Cuprins

Introducere.....	4
1.1 Descrierea lucrării	4
1.2 Motivația.....	5
1.3 Relevanța temei.....	5
1.4 Repere istorice.....	6
Prezentarea Aplicației	8
2.1 Experiența utilizatorului.....	8
2.2 Experiența utilizatorului în joc.....	9
Dezvoltarea aplicației	11
3.1 Dezvoltarea jocului	11
3.1.1 Tilemap	11
3.1.2 Player	12
3.1.3 Inamici Melee	16
3.1.4 Inamici zburători.....	18
3.1.5 Inamici Ranged	20
3.2 Sistemul de meniuri	22
3.3 Sistemul de sunet	23
3.4 Sistemul de salvare.....	24
3.5 Hărțile jocului	26
3.6 Sistemul de key rebind.....	27
Concluzii	28

Capitolul 1

Introducere

1.1 Descrierea lucrării

The Knight este un joc 2D, side view, action, adventure, platformer, level based, realizat în Unity, similar cu jocurile precum: *Hollow Knight* și *Ori and the Will of the Wisps*. Subgenul principal în care se află este acela de metroidvania care are ca reper de dezvoltare principal neliniaritatea gameplay-ului.

Premisa este una simplă. Jucătorul se trezește la marginea unui regat și pornește într-o aventură în care vrea să descopere mai multe despre el și despre lumea în care se află. Pe parcursul jocului el descoperă că nu este agreat de această lume, prin urmare va trebui să exploreze lumea și să răzbească provocările pe care le va întâmpina.

Unity este un motor de joc dezvoltat de Unity Technologies, anunțat și lansat pentru prima dată, în iunie 2005, la Apple Worldwide Developers Conference ca motor de joc Mac OS X.

Am ales Unity pentru dezvoltarea aplicației, deoarece Unity oferă suport pentru platforme multiple, oferind capacitatea de a dezvolta jocul o singură dată și de a-l lansa pe mai multe platforme. De asemenea Unity oferă un set bogat de instrumente și funcționalități cum ar fi: sistemele de animație, fizică, sunete, muzică, etc., care ajută la o dezvoltare, testare și modificare a unei aplicații mult mai rapidă.

În realizarea lucrării am dezvoltat un sistem de meniuri, sistem de mișcare pentru caracterul jucătorului, o inteligență artificială simplă pentru inamici, un sistem de salvare, sistem de animații pentru inamici și caracterul jucătorului, designul nivelurilor.

De asemenea, folosind Unity Store, am achiziționat asset-uri gratis cum ar fi sprite-ul și animațiile caracterului jucătorului și al inamicilor și sprite-uri pentru elemente din interfața

utilizatorului și teren.

1.2 Motivația

În opinia mea, jocurile video sunt o formă de artă ce combină avansul tehnologic, cu formele mai vechi de artă. Pentru a dezvolta un joc ai nevoie de: desen pentru sprite-uri, sculptură pentru modele 3D, compoziție muzicală, scris pentru poveste, etc.

Pe lângă partea artistică dezvoltatori, au nevoie de o viziune, pentru a putea crea o experiență de neuitat. Realizarea acesteia se poate face prin: mecanicile jocului, dificultatea jocului, flow-ul jocului etc. Flow-ul reprezintă capacitatea jocului de a captiva jucătorul, ce se poate realiza prin: poveste, efecte vizuale, mecanicile jocului, etc.

În cazul meu, jocurile dificile precum *Dark Souls*, *Elden Ring*, *Hollow Knight*, mi-au lăsat o impresie foarte bună, așa că am decis, să încerc, să dezvolt un joc similar cu *Hollow Knight*. Momentul de triumf pe care l-am simțit, după ce am reușit să răzlesc o provocare pe care un joc a pus-o în fața mea, a fost de neuitat, fapt ce m-a împins, în direcția de a dori, să ofer aceeași experiență pe mai departe.

De asemenea, curiozitatea de a afla mai multe, despre dezvoltarea jocurilor pe calculator, împreună cu pasiunea mea pentru acestea, mi-au influențat alegerea de a crea un joc pentru această lucrare.

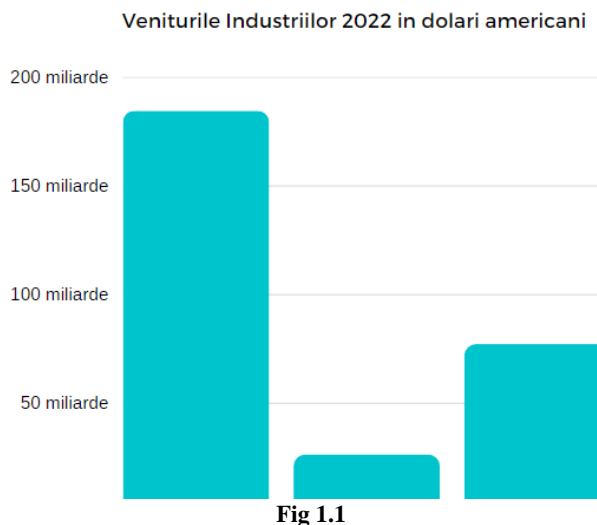
1.3 Relevanța temei

Jocurile video sunt o formă de divertisment interactivă, care reușesc să creeze experiențe virtuale, prin intermediul tehnologiei informației. Jocurile video pot fi consumate prin diferite modalități, cum ar fi: calculatoare, telefoanele mobile, console, căști de realitate virtuală, etc.

Jocurile video au o varietate largă de genuri și stiluri în care pot apărea, cum ar fi: genuri de acțiune, aventură, strategie, puzzle, platforming, se pot afla în formate: singleplayer, online multiplayer, 2D, 3D, side view, top-down view, etc. Atât timp cât există creativitate în ființa umană, jocurile pot apărea în orice combinație de stiluri și

genuri.

Veniturile industriei jocurilor video, în 2022, au fost de 184.4 miliarde de dolari [3] comparativ cu industria muzicii cu 26.2 miliarde de dolari [2] și industria filmelor cu 77 miliarde de dolari [1]. În același an, se observă că industria jocurilor video au venituri, cu 103.2 miliarde de dolari, mai mari decât industria jocurilor și filmelor combinată (Fig 1.1).



Jocurile video pot fi folosite și ca: simulări ale lumii înconjurătoare, simularea interacțiunilor corpurilor cerești (stelele, planetele, găurile negre, etc.) ce se regăsesc în jocul *Universe Sandbox*, simularea unui ecosistem uman ce se regăsesc jocurile open world ca și *Grand Theft Auto*, *Red Dead Redemption*, etc., simularea condusului (de mașini, bărci, avioane, rachete) ce se regăsesc în jocurile precum *Forza Horizon*, *Microsoft Flight Simulator*, *Spaceflight Simulator*, etc.

De asemenea, jocurile influențează și din punct de vedere filosofic. De exemplu: teoria simulării spune că realitatea înconjurătoare este de fapt o simulare, a fost inspirată din faptul că specia umană a ajuns în punctul de a putea simula o lume. Nu e greu să ne imaginăm că ar putea exista o specie, suficient de avansată tehnologic, încât să poată simula întreaga lume.

1.4 Repere istorice

“În octombrie 1958, fizicianul William Higinbotham a creat ceea ce se crede a fi primul joc video. Acesta era un joc de tenis foarte simplu, similar cu jocul clasic din anii 1970, Pong care a avut un mare succes, la o zi, a porților deschise la Laboratorul Național Brookhaven.” [5]

Donkey Kong, lansat în anul 1981 de către Nintendo, este primul joc platformer creat. Scopul jucătorului este să urce scări și să evite butoaiile aruncate, de către gorila Donkey Kong, în speranța de a salva femeia capturată de către gorilă.

Metroidvania este un subgen de jocuri de acțiune, aventură, ce se axează pe neliniaritate și de obicei prezintă o hartă mare interconectată. Numele metroidvania provine de la o combinație de nume ale jocurilor *Metroid* și *Castlevania*.

Metroid, lansat în anul 1986, creat de către Gunpei Yokoi, și deținut de către Nintendo, prezintă o femeie vânătoare de recompense, pe nume Samus Aran, care explorează lumi extraterestre, în scopul de a găsi ceea ce caută.

Castlevania, lansat în anul 1986, dezvoltat și publicat de către Konami, unde eroul are scopul de a explora castelul lui Dracula și de a răzbi legendarul vampir în scopul de a salva omenirea.

Grand Theft Auto 3, lansat în anul 2001, și publicat de către Rockstar Games, este primul joc open world de succes, unde jucătorul poate să trăiască o viață periculoasă de mercenar. Pe parcursul jocului, personajul întâmpină diferite grupuri de oameni de la care primește misiuni pe care trebuie să le îndeplinească.

Dark Souls, lansat în 2011, dezvoltat de FromSoftware și publicat de către Namco Bandai Games, a fost primul joc de succes care a promovat dificultatea jocurilor în industrie. Acțiunea jocului se desfășoară într-o lume fantastică (plină de dragoni, spirite și alte bestii fantastice) dar care este în același timp necruțătoare. Scopul personajului este să exploreze lumea să răzbească provocările și să obțină tronul regatului în care se află.

Hollow Knight a fost dezvoltat în mare parte de 3 persoane din Australia, Ari Gibson, William Pellen și David Kazi, ce au format echipa Team Cherry și cu ajutorul unui kickstarter de 50 de mii de dolari australieni, au reușit să lanseze jocul pe data de 24.02.2017. Până în anul 2019 au reușit să vândă peste 2.8 milioane de copii [4].

Hollow Knight este un joc dificil și nu oferă opțiunea pentru a alege dificultatea, ceea ce dintr-un punct de vedere al designului îl consider benefic, dezvoltatorii având o mai mare putere, de a-și exprima viziunea, prin faptul că toți utilizatorii vor avea experiențe asemănătoare.

Capitolul 2

Prezentarea Aplicației

2.1 Experiența utilizatorului

Utilizatorul când pornește aplicația este întâmpinat de un meniu principal de unde poate să înceapă jocul, să-și schimbe setările sau să închidă complet aplicația.

În momentul când va apăsa butonul de start, un meniu, de unde își va putea alege salvarea, va apărea. În acest meniu sunt prezente 3 sloturi dintre care poate alege. Pe fiecare slot vor apărea după logică următoarele opțiuni:

- Dacă există pe calculator salvarea respectivă a slotului, atunci utilizatorul va avea opțiunea de a continua jocul, de unde a rămas, sau de a șterge salvarea.
- Dacă nu există pe calculator salvarea respectivă slotului, atunci utilizatorul va avea doar opțiunea de a porni un joc nou.

În momentul când va apăsa butonul de opțiuni, utilizatorul va fi întâmpinat de un meniu, în care va avea posibilitatea de a alege, dacă vrea, să-și schimbe setările la sunete (volumul muzicii sau volumul efectelor sonore) sau setările de configurație al butoanelor, adică își va

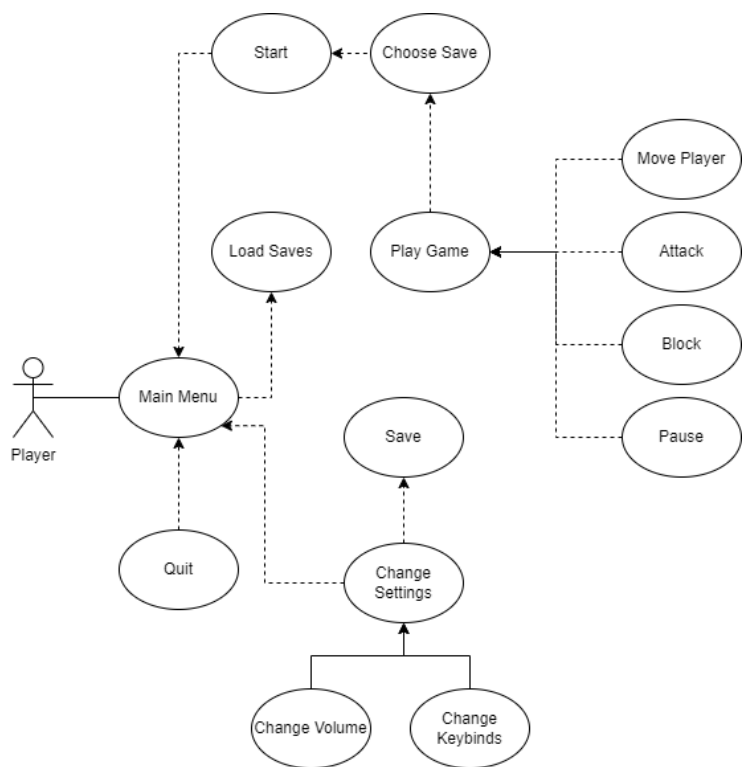


Fig 2.1

putea alege pe ce taste v-a apăsa, ca să execute acțiuni în joc (mișcare stânga, mișcare dreapta, sărit, atac, blocat sau interacțiune cu obiecte). În același timp, va avea opțiunea de a reseta configurația butoanelor, la cea inițială, care este: A – mișcare stânga, D – mișcare dreapta, SPACE – sărit, CLICK-STÂNGA – atac, CLICK-DREAPTA – block, E – interacțiune. Doresc să menționez că, utilizatorul poate folosi tasta ESCAPE, pentru a da toggle jocului pe pauză și de a naviga înapoi în meniu, tastă pe care o consider că nu ar trebui modificată, așadar utilizatorul nu va primi opțiunea de a o modifica.

2.2 Experiența utilizatorului în joc

În timpul jocului, utilizatorul se va întâlni cu 3 tipuri de inamici: inamici melee, pot interacționa cu jucătorul, doar din apropiere, sau vor patrula dacă acesta nu a fost detectat; inamici zburători, vor sta inactivi, cât timp nu vor detecta jucătorul, sau vor zbura spre acesta, încercând să-l rănească la contact, și inamici ranged, vor sta pe loc iar la detectarea jucătorului vor trage, cu un proiectil, în direcția lui.

În continuare, jucătorul se poate mișca stânga, dreapta, va sări, va ataca și bloca atacurile inamicilor. În timp ce blochează jucătorul, trebuie să înțeleagă faptul că blocarea atacurilor se face doar în partea cu scutul. Adică, dacă inamicul este în dreapta jucătorului și jucătorul blochează spre dreapta, atacul va fi blocat. Jucătorul în această situație nu va pierde din viață. Dacă blochează spre stânga și inamicul lovește din direcția dreapta a jucătorului, jucătorul va fi lovit și va pierde din viață. Menționez că, jucătorul poate sări doar dacă se află pe o platformă solidă de teren. Jucătorul are opțiunea să sară de pe pereți, doar în cazul în care nu este în contact cu podeaua și este în contact cu peretele.



Fig 2.2

Pe parcursul jocului, utilizatorul are posibilitatea de a interacționa cu puncte de salvare (Fig 2. 2), simbolizate sub formă de steag roșu. Interacțiunea se întâmplă, în cazul când jucătorul se află suficient de aproape de steag. Menționez că, interacțiunea cu aceste puncte de salvare regenerează viața curentă a jucătorului la maximum posibil.

În deplasare, prin lumea jocului, utilizatorul se va lovi de “uși”, care reprezintă o adâncitură în relieful terenului (Fig 2.3). În momentul în care, jucătorul intră prin aceste “uși”, va fi transportat în altă cameră de pe hartă.

Utilizatorul poate pune oricând jocul pe pauză, prin simpla apăsare a butonului ESCAPE. Din moment ce este apăsat acest buton, va apărea un meniu de pauză în fața utilizatorului, cu

următoarele opțiuni: un buton de “Continue”, pentru a ieși din meniul pauză și a reveni la joc (menționez că, apăsarea butonului ESCAPE are același efect), și a unui buton, de întoarcere la meniul principal.

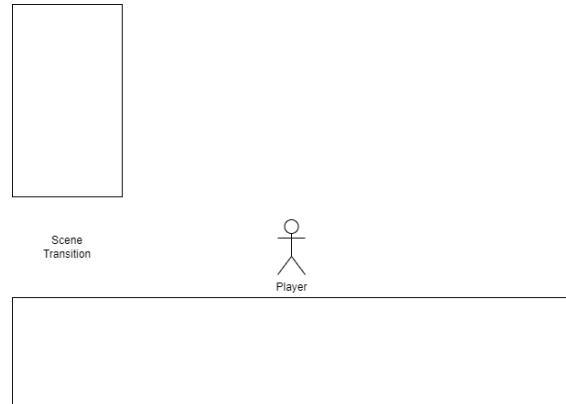


Fig 2.3

Capitolul 3

Dezvoltarea aplicație

3.1 Dezvoltarea jocului

3.1.1 Tilemap

În dezvoltarea aplicației, a trebuit să iau o decizie, în legătură cu modalitatea de construire a hărții. Pentru ușurarea procesului de construire a terenului, am decis să folosesc un tilemap de tip dreptunghiular. Un tilemap dreptunghiular împarte întreaga scenă într-un grid de 1x1 unități, unde se pot plasa tile-urile, la fel ca niște piese de puzzle.

Pentru a obține tile-uri, am importat un tileset de pe Unity Asset Store [6]. Acest asset are mai multe componente, iar eu tot ce folosesc din el sunt tile-urile din figura 3.1.

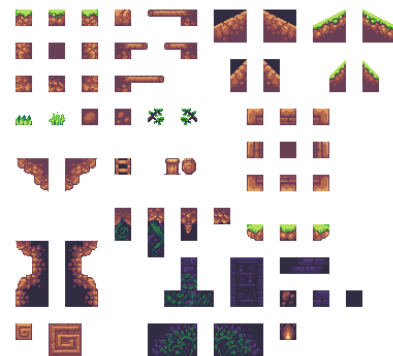


Fig 3.1

Pentru a folosi tile-urile imagine, acestea trebuie pregătite. Pentru pregătire trebuie urmați următorii pași: trebuie selectată imaginea în editorul unity, selectat sprite mode pe multiple, să adăugăm câți pixeli, pe unitate, ocupă un tile (în cazul meu 16 este numărul perfect). După ce am selectat aceste opțiuni, apăsăm pe apply și urmăm să facem tăierea propriu zisă a imaginii. Pentru a face această tăiere, trebuie să: deschidem imaginea în sprite editor, selectăm tipul de slice pe grid by cell size, alegem pixel size-ul de ($X = 16$, $Y = 16$), apăsăm butonul de slice după care, aplicăm modificările, apăsând pe butonul apply.

În momentul în care avem un tilemap pregătit, trebuie să creăm o paletă. Pentru a crea o paletă de tile-uri, trebuie să ne ducem pe window -> 2D -> tile palette. După ce am deschis fereastra pentru palete, creăm o paletă nouă, o salvăm în proiect. Suntem gata de a construi hărțile perfecte.

Pentru acest proiect, am creat două tilemap-uri: unul pentru teren și unul pentru background. Am realizat 2 layere (unul pentru background și unul pentru teren), în așa fel încât, să obținem layer-ul de teren. Pentru fiecare tilemap am selectat, în tilemap renderer, layer-ul respectiv fiecărui tilemap. Menționez faptul că, asset-ul de background a fost importat de pe unity store [16].

Pentru layer-ul de teren a trebuit să-l configurez, în așa fel, încât player-ul să poată interacționa cu el. Pentru a realiza acest lucru la tilemap, am fost nevoit să adaug două componente, un Tilemap Collider 2D și un Rigidbody 2D. Dacă pe această configurație, dăm start la scenă, observăm că tot terenul cade. Pentru a rezolva această problemă, în componenta Rigidbody 2D, selectăm Body Type-ul pe static. Dacă vom da play, se observă că terenul nu va mai cădea.

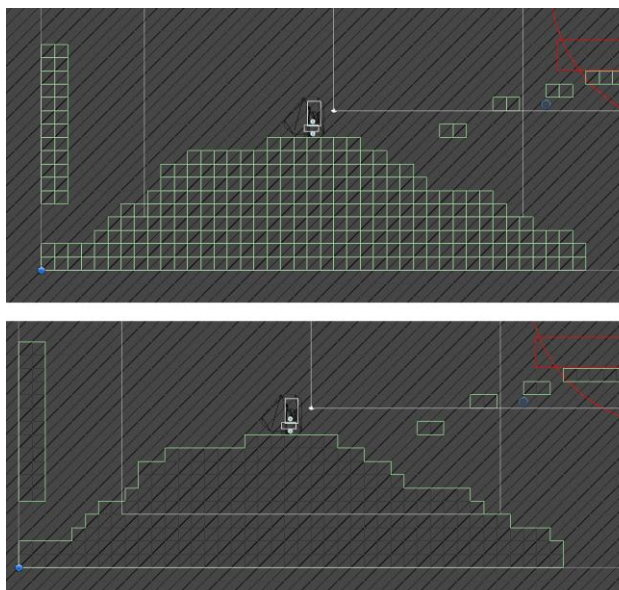


Fig. 3.2

Dacă schimbăm shading mode-ul în wireframe, observăm că fiecare tile de teren va avea colliderul ei, ceea ce nu este necesar. Pentru a optimiza acest lucru vom: adăuga componenta Composite Collider 2D pe tilemap-ul terenului și în Tilemap Collider 2D voi bifa căsuța Used By Composite și se va obține efectul din figura 3.2.

3.1.2 Player

Pentru player am importat de pe Unity Asset Store [7] modelul de la figura 3.3. Pentru început am creat în scenă un nou obiect 2D, de tip pătrat, am schimbat sprite-ul din componenta Sprite Renderer cu sprite-ul HeroKnight_0, am ales sorting layer-ul egal cu Player, am mărit scale-ul obiectului pe axele x și y la 2.



Fig 3.3

Mai departe, am adăugat componenta de rigidbody 2D, unde s-a pus un material de fizici 2D care are proprietatea de frecare și elasticitate egale cu 0 (pentru a ignora frecarea și reacțiunea la coliziune cu terenul) și un box collider 2D, pe care l-am configurat puțin mai mic, decât sprite-ul lui, deoarece așa pot evita aparența de levitare a sprite-ului deasupra pământului.

Pentru orientarea corectă a obiectelor în scenă, am creat o clasă care va roti pe axa y cu 180° la fiecare flip și pe care am adăugat-o ca și componentă la obiectul player.

În continuare, am adăugat componenta animator (fig 3.4) care are rolul de a anima mișcările caracterului jucătorului. Caracterul se poate afla în următoarele stări: Idle, Running, Falling, Jumping, Attack, Block, Hurt, Wallslide, Death. Animatorul va avea un parametru de tip int pentru starea de mișcare, 3 de tip trigger pentru attack, hurt și block și 2 de tip bool pentru isDead și walled. Logica este următoarea, caracterul se află în una din cele 4 stări de mișcare: idle, moving, falling, jumping. La setarea unui trigger se va executa animația trigger-ului respectiv și se va întoarce în starea de mișcare corectă, atât timp cât nu a murit sau nu mai este agățat de un perete.

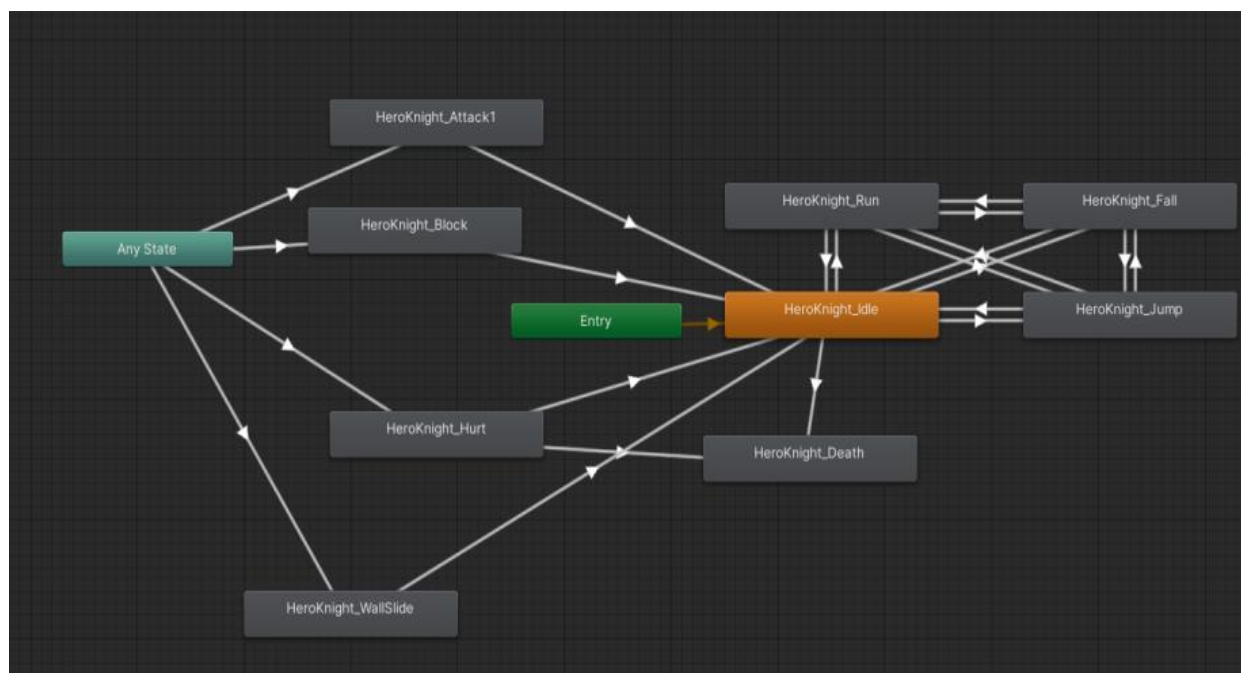


Fig 3.4

La pornirea scenei caracterul se află în starea Idle, la apăsarea butoanelor de movement, player-ul se poate mișca fie în stânga, fie în dreapta. În funcție de direcția în care merge, player-ul va fi orientat în direcția corespunzătoare iar starea i se va schimba în Moving. Dacă viteza pe axa y este pozitivă, indiferent de direcția de mișcare, în animator player-ul se va

afla în starea Jumping. Odată ce, va avea viteza negativă pe axa y, în animator se va afla în starea Falling. Menționez că, verificarea se face cu pragurile de -0.1, pentru verificare vitezei negative, și 0.1, pentru cea pozitivă.

În continuare, voi discuta despre logica săritului. Pentru a evita săritul nelimitat, la fiecare apăsare de buton, trebuie să facem o verificare, dacă player-ul se află în contact cu pământul. Pentru a realiza acest lucru, îmi trebuie un detector de pământ (voi adăuga un obiect Empty, în obiectul player-ului, care va avea rol de poziție pentru box cast-ul detectorului). „Un BoxCast este conceptual similar cu tragerea unei cutii prin scenă într-o anumită direcție. Orice obiect care intră în contact cu cutia poate fi detectat și raportat.” [8] Voi face acest cast la picioarele caracterului jucătorului și voi filtra după layerMask-ul terenului. Adică, am luat în calcul doar collider-ele care fac parte din acest layer. În situația în care detectez o coliziune cu terenul, la picioarele jucătorului, îi voi permite să sară.



Fig 3.5

Pentru detectarea coliziunii cu peretele voi face similar ca în cazul detectării podelei. De data aceasta verific dacă nu se află în coliziune cu pământul și este în contact doar cu peretele. Dacă acest lucru se întâmplă, voi da permisiunea să sară de pe perete, purtând denumirea, în industria jocurilor, de wall-jumping. Box cast-urile vizualizate le puteți vedea în figura 3.5.

La partea de atac, voi adăuga un detector similar ca la detectarea coliziunii cu peretele. De data aceasta voi folosi metoda `OverlapCircleAll` din `Physics2D`, filtrată pe layer-ul inamicilor, pentru a selecta toți inamicii care se află în acest cerc. De ce am ales un cerc și nu o cutie? Deoarece corespunde mai bine cu sprite-ul atacului (figura 3.6). De menționat că, pe animația de atac, am adăugat un eveniment care va face overlap-ul respectiv și va lua din viața inamicilor, găsiți în zona respectivă.

Pe partea de blocare a atacurilor, am adăugat în player, un `boxCollider` care apare și dispare, la apăsarea butonului mapat de utilizator. La atacul inamicului, se verifică dacă pe axa x acest collider se află între collider-ul inamicului și al player-ului. Dacă

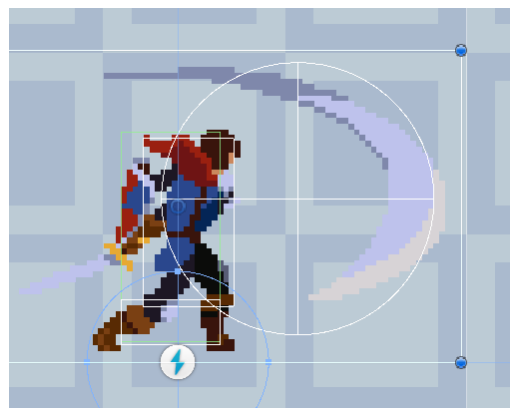


Fig 3.6

se află și este activ, atunci player-ul nu va pierde din viață, iar la contactul cu proiectilele acestea dispar la atingerea cu collider-ul scutului. Collider-ul va fi activat și dezactivat, de către comportamentul animației. La începutul animației acesta va fi activat, iar la sfârșit va fi dezactivat.

Pentru sistemul de viață, am creat o componentă care va memora viața maximă și viața curentă, componentă ce poate fi adăugată la orice inamic din scenă. Va avea metodele `takeDamage`, care vor scădea din viața curentă a entității și va verifica dacă aceasta a ajuns la 0. Dacă da, va activa animația de moarte și va dezactiva obiectul. În plus această componentă poate interacționa cu bara de viață ce se află în scenă.

Bara de viață este un slider care are o componentă creată de mine. În ea vom putea seta valoarea maximă și valoarea curentă al slider-ului, dar voi adăuga și un gradient care își va schimba culoarea în funcție de valoarea pe care o va primi. Deoarece gradientul are valori doar între 0 și 1 și slider-ul poate avea valori între 0 și orice număr decidem să punem ca și viață maximă, v-a trebui să extragem din slider valoarea normalizată. Acest lucru este simplu de realizat, deoarece unity ne oferă deja această metodă.

Pentru sistemul de input, voi importa noul sistem de input creat de Unity Technologies care este mult mai extensibil și customizabil decât cel clasic. În primul rând, pentru a folosi acest sistem, trebuie importat. Pentru a face acest lucru, deschidem package manager-ul de la unity și vom căuta "Input System". După ce l-am importat, trebuie să creăm un input action asset, unde vom mapa butoanele de care avem nevoie. În cazul meu am nevoie de trei butoane pentru mișcare și patru butoane pentru restul acțiunilor. Toate butoanele vor fi verificate, doar dacă sunt apăstate, așa că acțiunea lor va fi de tip buton. În cazul butoanelor de mișcare, vreau să primesc o valoare negativă dacă player-ul se mișcă la stânga și pozitivă dacă se mișcă la dreapta. Tipul acțiuni va fi de tip valoare, și tipul de control este axis, deoarece mă interesează o singură direcție.

După ce am creat input action asset-ul, vom adăuga o componenta Player Input, pe jucător. Vom atribui în câmpul actions asset-ul nostru. Vom schimba comportamentul, să fie de tip `Invoke Unity Events`, și pentru fiecare acțiune pe care am creat-o în asset îi atribui o metodă creată de mine, pe care să o execute.

În final, după finalizarea caracterului jucătorului, acesta a fost transformat într-un prefab. Un prefab este un obiect prefabricat care poate fi reutilizat în alte scene.

3.1.3 Inamici Melee

Pentru inamicii de tip melee, am importat de pe Unity Asset Store [9], modelul de la figura 3.7. La început am creat în scenă un nou obiect 2D, de tip pătrat, și am schimbat sprite-ul din componenta Sprite Renderer cu sprite-ul LightBandit_0. Am ales sorting layer-ul egal cu Enemies, am mărit scale-ul obiectului pe axele x și y la 2.



Fig 3.7

Mai departe, am adăugat componenta de rigidbody 2D, unde am adăugat același material de fizici 2D ca și la player, și un box collider 2D, pe care l-am configurat puțin mai mic decât sprite-ul lui, deoarece așa pot evita aparența de levitare a sprite-ului deasupra pământului. Pentru orientarea corectă a obiectului, am adăugat aceeași componentă ca și la player, care va roti obiectul pe axa Y cu 180^0 .

În continuare, am adăugat componenta animator (fig 3.8), care are rolul de a anima mișcările inamicului (caracterul se poate afla în următoarele stări: Idle, Running, Attack, Hurt, Death). Animatorul va avea parametrul de tip: int, pentru starea de mișcare; 3 de tip trigger, pentru attack, hurt; și unul de tip bool, pentru isDead. Logica este următoarea: caracterul se află în una din cele 2 stări de mișcare (idle, moving). La setarea unui trigger, va executa animația trigger-ului respectiv și se va întoarce în starea de mișcare corectă, atât timp cât nu a murit.

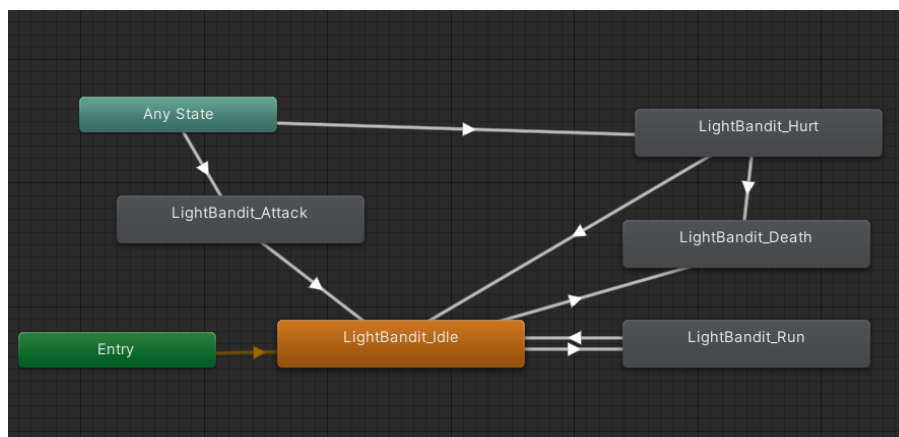


Fig 3.8

Inamicul va avea aceeași componentă de sistem de viață ca și player-ul, cu diferența că nu va avea atribuită o bară de viață.

La începutul scenei, inamicul se va afla în modul de patrulare. Modul de patrulare funcționează în felul următor: voi crea un obiect empty, la care voi atașa o componentă ce se va ocupa de comportamentul de patrulă. În acest obiect gol, voi adăuga inamicul ca și copil, și două obiecte empty, ce vor avea rol de capete ale zonei de patrulare, un capăt stâng și unul drept. Inamicul va începe să meargă de la un capăt la altul, oprindu-se câteva secunde în starea de idle, când ajunge într-unul din capete (fig 3.9).



Fig 3.9

Pe lângă comportamentul de patrulare a inamicului, acesta va verifica printr-un box cast (fig 3.10) filtrat pe layer-ul jucătorului, dacă jucătorul se află în aria lui vizuală. Odată detectat acesta părăsește comportamentul de patrulare și intră în comportamentul de urmărire. Menționez că, inamicul va urmări, câteva secunde, jucătorul după ce iese din raza lui vizuală, iar după terminarea timpului se va întoarce la comportamentul lui de patrulare.

Pentru a evita detectarea jucătorului prin pereți, în momentul când este detectat, se va face un ray cast în direcția jucătorului, pentru a verifica dacă se află pereți între inamic și jucător.

În timp ce inamicul urmărește jucătorul, acesta va face un circle cast pentru a verifica dacă se află în raza lui de atac. Când este detectat jucătorul în această rază, inamicul va iniția animația de atac, care va avea, similar cu jucătorul, un trigger la un anumit punct, unde va încerca să rănească jucătorul.

Odată inițiat, trigger-ul de rănire al jucătorului, se va face o verificare dacă jucătorul se află în continuare în raza de atac, și se va verifica poziția scutului dacă acesta este activat. Dacă scutul este activ și se află între jucător și inamic, atunci jucătorul nu va pierde din viață, altfel va

pierde.

Similar cu jucătorul, inamicul are un detector, sub formă de cerc, în fața lui la picioare, cu scopul de a detecta dacă poate merge în față. Dacă detectează lipsa pământului în fața lui, acesta se va opri din mers.



Fig 3.10

Într-un final, după ce am terminat de creat acest tip de inamic, îl voi transforma într-un prefab la fel cum am procedat și la caracterul jucătorului pentru al putea refolosi în scene viitoare.

3.1.4 Inamici zburători

Pentru inamicii de tip zburători, am importat de pe Unity Asset Store [10], modelul de la figura 3.11. Pentru început, am creat în scenă un nou obiect 2D, de tip pătrat, am schimbat sprite-ul din componenta Sprite Renderer cu sprite-ul Idle (46x30)_0, am ales sorting layer-ul egal cu ContactEnemies și am mărit scale-ul obiectului pe axele x și y la 3.

Mai departe, am adăugat componenta de rigidbody 2D împreună cu același material de fizici 2D, ca și la player, și un box collider 2D, pe care l-am configurat, puțin mai mic decât sprite-ul lui. Pentru orientarea corectă a obiectului, am pus aceeași

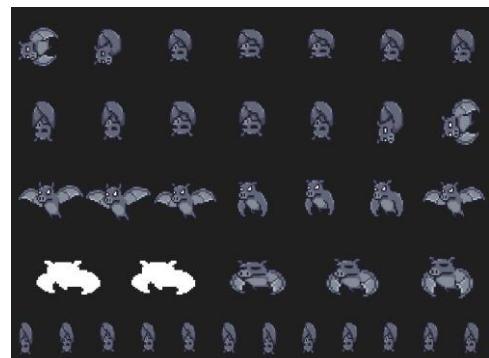


Fig 3.11

componentă ca și la player, care va roti obiectul pe axa Y cu 180^0 , și componenta de logică a vieții. pe care o au și restul caracterelor.

În continuare, am adăugat componenta animator (fig 3.12), care are rolul de a anima mișcările inamicului, caracterul se poate afla în următoarele stări: Idle, Flying, Ceiling_out, Ceiling_in, Hurt, Death. Animatorul va avea parametrul de tip int, pentru starea de mișcare, unul de tip trigger pentru hurt și doi de tip bool pentru isDead și IsMoving. Logica este următoarea, caracterul se află în una din cele 2 stări de mișcare: idle, moving. La schimbarea stării, inamicul va trece prin starea de Ceiling_out, înainte de a intra în starea Flying, și va trece prin starea Ceiling_in, pentru a intra în starea de idle. La setarea unui trigger va executa animația, trigger-ului respectiv, și se va întoarce în starea de mișcare corectă, atât timp cât nu a murit. Starea de Flying are un comportament în plus, la intrare în această stare parametrul de tip bool isMoving va fi setat la true, și la ieșirea din această stare este setat înapoi la false, acest comportament este adăugat pe acest inamic deoarece există stările de tranziție între Idle și Flying, și doresc să aplic forțe de mișcare pe inamic doar dacă acesta se află în starea de Flying.

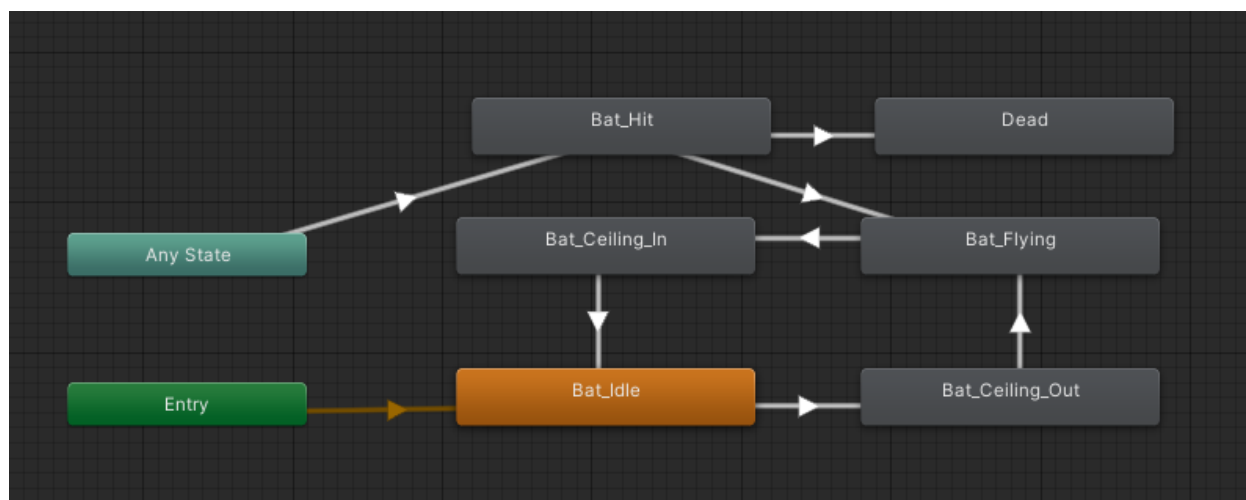


Fig 1.12

Inamicul va avea aceeași componentă de sistem de viață ca și player-ul, cu diferența că nu va avea atribuită o bară de viață. La începutul scenei, inamicul se va afla în modul idle.

Inamicul va verifica printr-un circle cast (fig 3.13), filtrat pe layer-ul jucătorului, dacă se află jucătorul în aria lui vizuală, iar pentru a evita detectarea jucătorului prin pereți, odată detectat, se va face un ray cast în direcția jucătorului. Pentru a verifica dacă pereții se află între acesta și jucător, odată detectat, acesta părăsește starea de idle și intră în starea de urmărire a

jucătorului, similară cu cea a inamicilor de tip melee. Inițial inamicul mergea într-o linie dreaptă, spre poziția jucătorului, fără a evita obstacolele care se află pe hartă. Pentru a rezolva această problemă, am decis, să dau import la pachetul A* pathfinding, dezvoltat de către Aron Granberg [11].

Pentru a folosi algoritmul de pathfinding, trebuie adăugat în scenă un obiect empty, ce va avea componenta Pathfinder. În această componentă, am decis, să folosesc un graf de tip grid, unde voi bifa căsuța 2D în secțiunea grid graph, și căsuța “Use 2D Physics”, iar nodurile vor avea dimensiunea de 0.45, pentru o precizie mai mare la detectarea terenului.

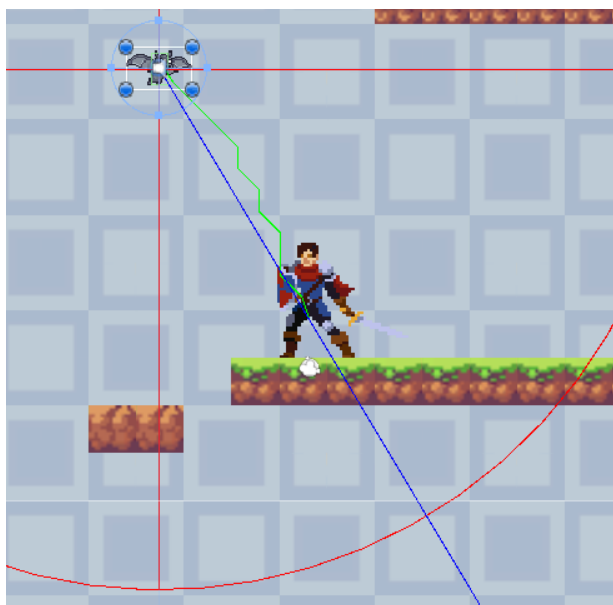


Fig 3.13

Pentru generarea căii, trebuie să adăugăm componenta seeker pe inamicul nostru iar acum pot folosi algoritmul de A* pathfinding în scripturile mele.

Datorită faptului că, inamicul nu are o animație de atac, am decis, să creez componenta de rănire la contact. Astfel inamicul va încerca să zboare în caracterul jucătorului. De asemenea, acesta verifică dacă există un scut între el și hitboxul jucătorului. Dacă acest scut este prezent, jucătorul nu va pierde viață. Pentru a îmbogăți feedback-ul pe care îl primește jucătorul, la contact cu acest tip de inamic, vor fi aplicate forțe de knockback. Adică el va fi aruncat în direcția opusă și în sus față de inamic.

Într-un final, după ce am terminat de creat acest tip de inamic, îl voi transforma într-un prefab, la fel cum am procedat și la caracterul jucătorului, pentru al refolosi în scene viitoare.

3.1.5 Inamici Ranged

Pentru inamicii de tip ranged, am importat de pe Unity Asset Store [10] modelul de la figura 3.14. La început, am creat în scenă un nou obiect 2D, de tip pătrat, și am schimbat sprite-ul din componenta Sprite Renderer cu sprite-ul Idle (44x42)_0. Am ales sorting layer-ul egal cu Enemies, am mărit scale-ul obiectului pe axele x și y la 5.



Fig 3.14

Mai departe, am adăugat componenta de box collider 2D, pe care l-am configurat puțin mai mic decât sprite-ul lui. Pentru orientarea corectă a obiectului, am adăugat aceeași componentă, ca și la player, care va roti obiectul pe axa Y cu 180^0 precum și componenta de logică a vieții pe care o au și restul caracterelor.

În continuare, am adăugat componenta animator (fig 3.15), care are rolul de a anima mișcările inamicului, caracterul se poate afla în următoarele stări: Idle, Attack, Hurt și Dead. Animatorul va avea doi parametri de tip trigger (unul pentru atac și unul pentru hurt) și un parametru de tip bool isDead, care reprezintă starea inamicului de mort sau viu.

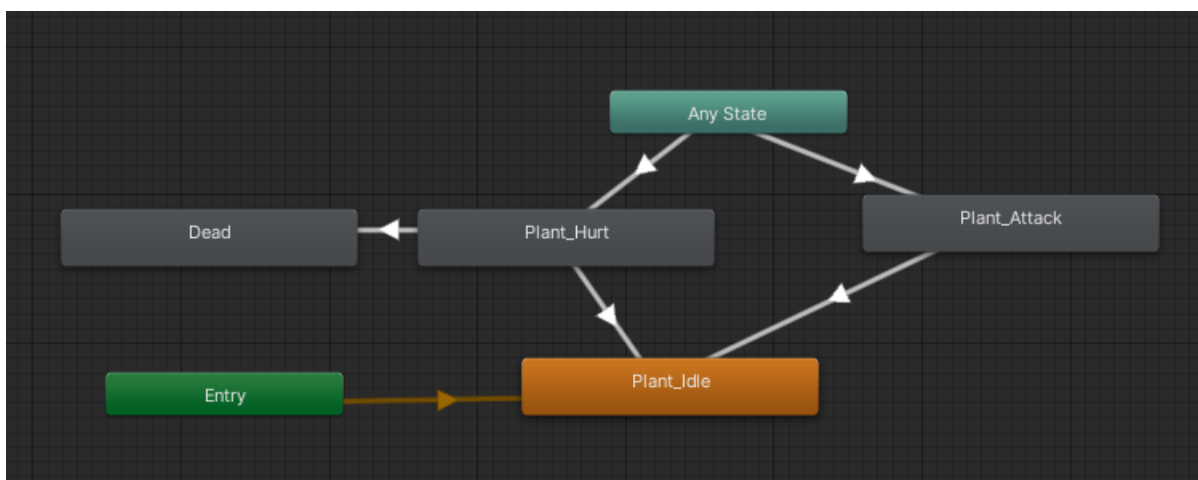


Fig 3.15

Detectarea jucătorului, o fac cu un box cast (fig 3,16), destul de mare comparativ cu restul inamicilor, deoarece acesta este un inamic care atacă de la depărtare. Consider faptul că acesta trebuie să aibă și o zonă de detecție mult mai mare. Bineînțeles că, și la el voi face un raycast pentru detectarea pereților, în scopul de a evita detectarea jucătorului, când acesta se află

în spatele unui zid.

Inamicului, i-am adăugat un obiect empty firePoint, care reprezintă punctul de unde el va trage cu proiectile, atunci când va detecta jucătorul. În momentul când inamicul detectează jucătorul, un obiect de tip proiectil va fi instanțiat.

Pentru ca inamicul să poată trage, are nevoie de proiectile. Voi crea un prefab pentru proiectile, care va fi instanțiat de fiecare dată când inamicul va trage (instanțiere este executată cu ajutorul unui trigger pe animația atacului). Acest proiectil va ignora coliziunile cu inamici, iar la contactul cu orice altceva va fi distrus. Excepție este coliziunea cu jucătorul, când acesta este rănit și se vor pierde puncte din viață.

AI-ul inamicului este unul simplu, dacă detectează jucătorul, se va uita spre el.

Într-un final, după ce am terminat de creat acest tip de inamic, îl voi transforma într-un prefab, la fel cum am procedat și la restul caracterelor, pentru al putea refolosi în viitoarele scene.

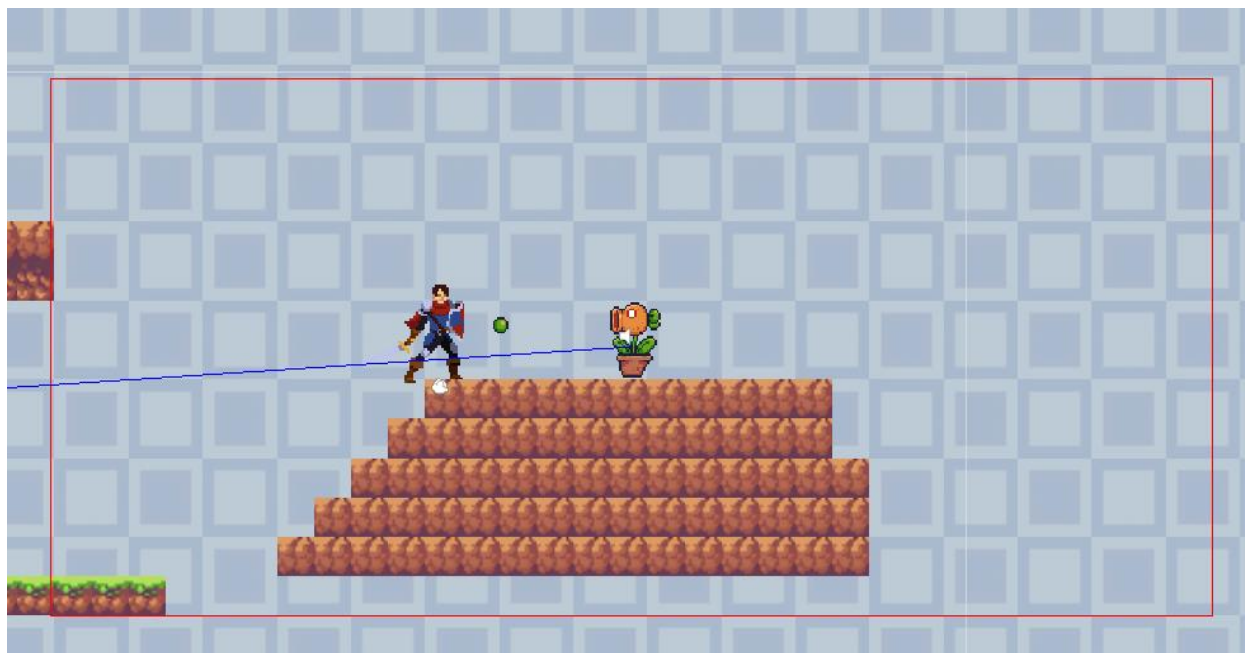


Fig 3.16

3.2 Sistemul de meniuri

Jocul este format din următoarele componente: meniul principal, meniul de selectare al salvărilor, meniul opțiunilor și meniul de pauză. Meniul principal este format dintr-un:

background, un text de titlu, și 3 butoane (butonul de start, butonul de opțiuni și butonul de ieșire din aplicație). Menționez că, sprite-ul butoanelor a fost importat de pe unity asset store [12].

La apăsarea butonului de start, va apărea un meniu cu slot-urile de salvare a datelor jocului, un slot poate apărea în una din 2 configurații: dacă slotul respectiv, nu are o salvare regăsită pe calculator, atunci va apărea doar un buton de new game pentru a porni un joc nou; dacă salvarea este regăsită pe calculator, utilizatorul va putea continua jocul de unde a rămas pe salvarea respectivă, sau poate șterge salvarea definitiv.

La apăsarea butonului de opțiuni, utilizatorul va fi întâmpinat cu meniul de opțiuni care este alcătuit din 2 butoane: un buton care va deschide setările sunetelor din aplicație; un buton care va deschide meniul de mapare a butoanelor.

La apăsarea butonului quit, aplicația se va închide. Menționez că, ambele meniuri au un buton back, care apăsând va trimite utilizatorul înapoi la meniul principal.

În timpul jocului, utilizatorul poate deschide meniul de pauză, apăsând pe tasta ESCAPE, unde va primi opțiunile: de a continua, de a deschide meniul de opțiuni, de a se reîntoarce la meniul principal.

3.3 Sistemul de sunet

Sistemul de sunet este alcătuit din surse audio, și un output mixer (fig 3.17). Sunetele [13] și muzica [14] au fost importate pe gratis de pe Unity Asset Store.

Pentru început am creat mixer-ul audio și un grup master, unde au fost adăugate 2 grupuri copii, unul pentru muzică și al doilea pentru efecte sonore.

Pentru muzică am creat un obiect empty care va avea ca sursă melodia respectivă, la care am adăugat un script, prin intermediul căruia voi utiliza metoda DontDestroyOnLoad, ce are ca efect păstrarea elementului la încărcarea diverselor scene. După care, la acest obiect am adăugat sursa audio Ambient 2, cu outputul setat la grupul muzicii din mixer.

Muzica este setată să înceapă la pornirea aplicației și este pusă pe modul de loop. Spațial Blend-ul este setat pe 2D pentru a putea auzi muzica de oriunde.

În cazul efectelor de sunet, acestea sunt puse pe obiectele din scenă cum ar fi: avatarul jucătorului, inamicii melee, inamicii zburători și inamicii ranged.

Sunetele, ce pot fi auzite, în timpul jocului sunt: sunete de sărit, sunete de lovit cu sabia,

sunete de mers, sunete de tras cu proiectile, sunete pentru pierderea punctelor de viață.

Pentru a putea modifica valoarea outputului dintr-un grup audio, în primul rând trebuie să expunem parametrul de volum. Relizarea acestui lucru, se face prin apăsarea pe grupul căruia dorim să-i expunem volumul, click dreapta pe volum în inspector, unde vom putea vedea opțiunea de expunere a parametrului. Voi obține această expunere pentru ambele grupuri de muzică și efecte sonore.

După ce am expus variabilele, pentru a seta volumul, mă voi folosi de sliderele din canvas. În setarea volumului, am observat că, slider-ul își schimbă valoarea liniar, iar volumul mixerului se modifică pe o scară logaritmică. Pentru a rezolva această problemă, voi transforma valoarea slider-ului în modul următor: $\text{volum mixer} = \log_{10}(\text{valoare slider}) * 20$. Menționez că, trebuie setat minimul la slidere din 0 în 0.0001, pentru a evita efectul, de a ajunge la volumul maxim, atunci când slider-ul se află pe valoarea 0.

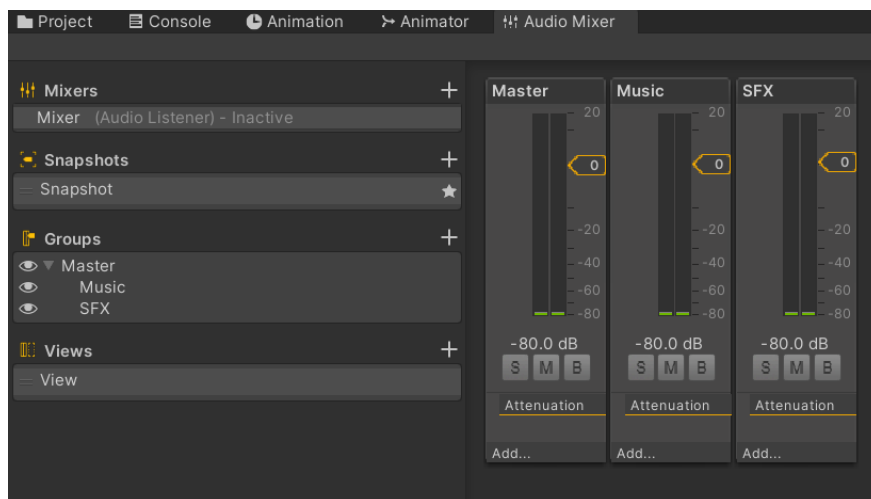


Fig 3.17

3.4 Sistemul de salvare

Sistemul de salvare este realizat prin folosirea obiectelor de tip `scriptableObjects` și formatare binară, pentru salvarea locală a datelor, pe care doresc să le salvez.

“Un `ScriptableObject` este un container de date pe care îl poți folosi pentru a salva cantități mari de date, independent de instanțele clasei. Unul dintre principalele cazuri de utilizare pentru `ScriptableObjects` este de a reduce consumul de memorie al proiectului prin evitarea copilor de valori.” [8]

“Clasa BinaryFormatter în C# realizează acțiunile de "serializare" și "deserializare" a datelor binare. Aceasta preia structuri de date simple, precum numere întregi (int), numere zecimale (float), colecții de litere și numere (string) și le poate converti într-un format binar. Acest lucru înseamnă că structuri mai complexe, precum o clasă, își pot codifica câmpurile într-un format binar pentru a fi salvate într-un fișier și citite ulterior de un program. În Unity, prin utilizarea acestei clase C#, este posibil să se salveze date, la fel cum s-ar putea face cu PlayerPrefs, dar la un nivel mai complex.” [15]

În aplicația mea, voi folosi formatarea binară pentru salvarea volumului și a salvărilor respective fiecărui slot, iar în cazul mapării butoanelor, salvarea o voi face în PlayerPrefs. În cazul meu, voi avea 3 slot-uri de salvări cum am menționat în subcapitolul 3.2. Călea la care vor fi salvate aceste date, este calculată folosind clasa Application și accesarea valorii de la atributul persistentDataPath, această valoare fiind diferită în funcție de sistemul pe care se află aplicația.

În continuare, voi folosi ScriptableObjects, pentru a stoca informații locale între scene în felul următor: pentru a tranzitiona de la o scenă la alta am creat un trigger fig. 3.18, de tipul BoxCollider2D, care va memora viața curentă și va schimba poziția pe care trebuie să o aibă în scena următoare. Adică, în momentul când jucătorul intră în contact cu acest trigger, o scenă nouă va fi încărcată, iar la încărcarea acesteia, se vor pierde datele curente ale jucătorului, cum ar fi viața curentă, așa că datele trebuie salvate undeva local. Mai departe, trebuie să menționăm și poziția unde ar trebui, să se afle jucătorul în scena următoare, la încărcare. Menționez că, aceste date nu sunt salvate în calculator.

În continuare, voi discuta despre punctele unde jucătorul poate salva jocul. Acestea sunt simbolizate sub forma unui steag roșu (fig 2.2). Acest steag roșu este, de asemenea, un trigger, la coliziunea între jucător și acest trigger. Jucătorul va avea opțiunea, prin apăsarea tastei de INTERACT, care este mapată în mod implicit pe tasta E, să salveze jocul. Salvarea presupunând viața, poziția și scena curentă în care se află, în același timp acest trigger va avea efectul de vindecare completă al jucătorului.

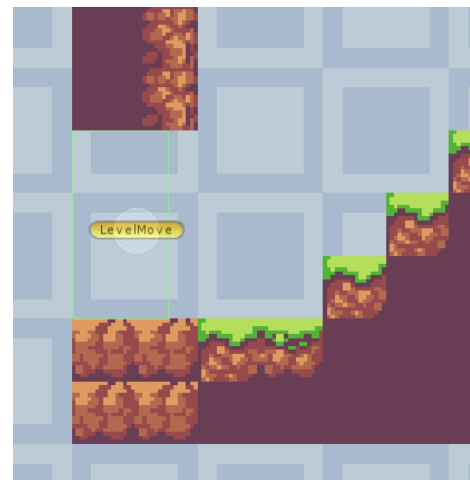


Fig 3.18

În momentul, când jucătorul își pierde toată viața, acesta va reapărea la ultimul punct de

salvare cu care a interacționat. Același efect, îl are și atunci când utilizatorul va apăsa pe butonul de încărcare (“Load”), respectiv unuia dintre cele 3 slot-uri.

3.5 Hărțile jocului

Jocul este format din 5 nivele, care au următoarele forme (fig. 3.19 – fig 3.23):

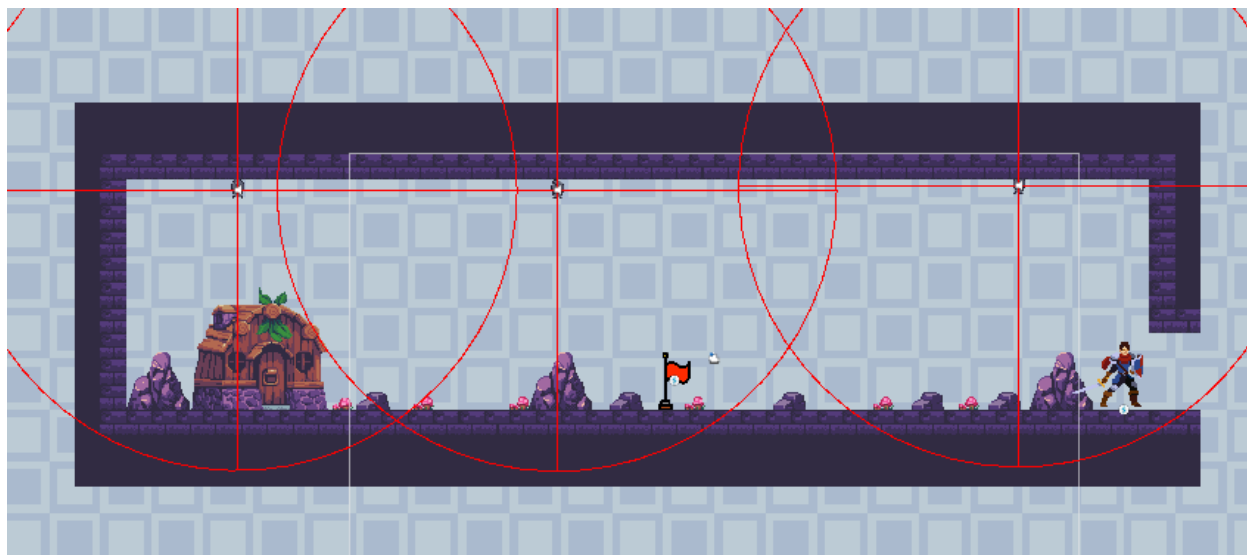


Fig 3.19

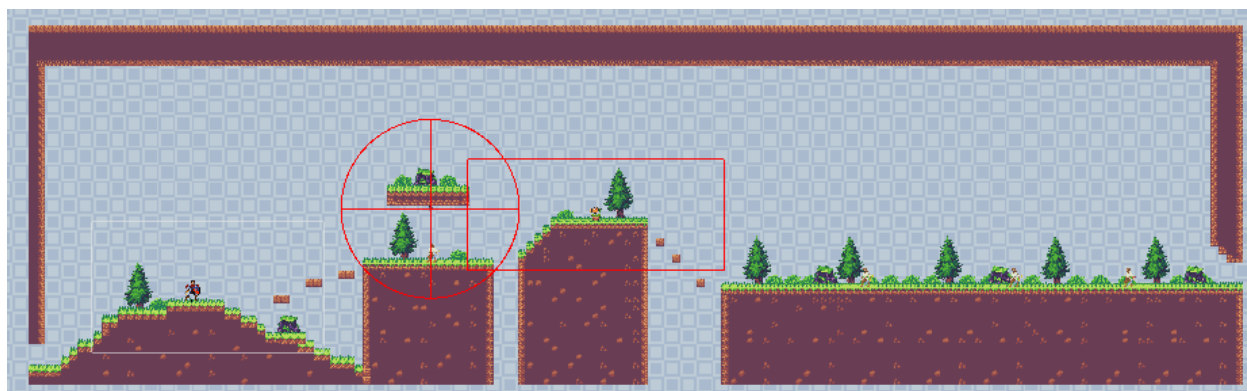


Fig 3.20



Fig 3.21

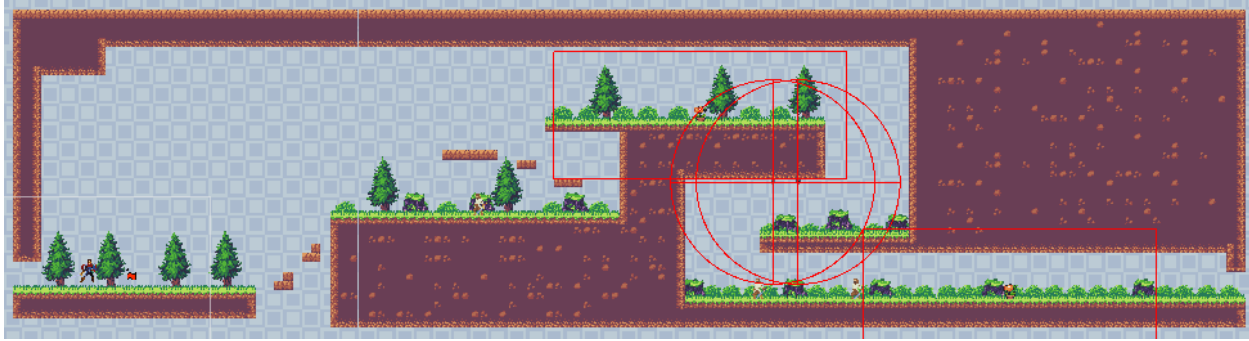


Fig 3.22

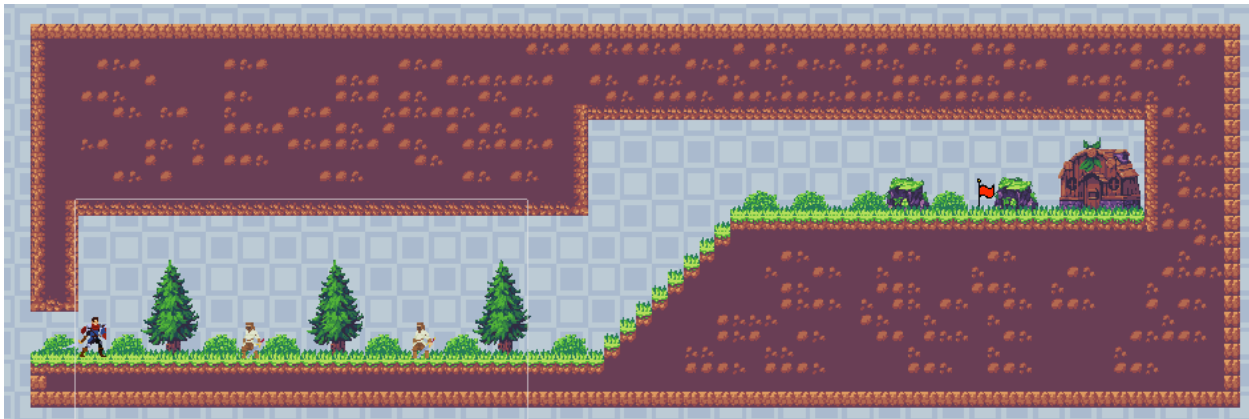


Fig 3.23

Menționez că propurile fac parte din asset-ul [6].

3.6 Sistemul de key rebind

În dezvoltarea sistemului de key rebind, am importat din pachetul Input System, dezvoltat de către Unity Technologies demo-ul Rebinding UI. Din acest demo, am folosit prefab-ul RebindUIPrefab, pentru construirea interfeței de rebinding, ce constă în: numele acțiuni pe care o modificăm; un câmp în care se află keybinding-ul curent, pe care putem apăsa pentru a începe procesul de rebinding; două butoane de reset.

Scriptul RebindingActionUI a rămas asemănător cu cel dezvoltat de Unity Technologies, cu următoarele modificări: am adăugat o verificare de dubluri între InputAction-uri și o verificare la anulare, în cazul în care InputAction-ul este de tip composite (unde verific dacă se află dubluri pe configurația curentă, dacă se observă dubluri voi reconfigura binding-urile acțiuni la ultima configurație validă).

Capitolul 3

Concluzii

În concluzie, proiectul meu, The Knight, de dezvoltarea a unui joc în Unity a oferit o experiență valoroasă, în ceea ce privește designul și implementarea unui joc captivant, obiectivul meu principal fiind crearea unui joc, care să poată oferi o experiență distractivă și în același timp să ofere o provocare utilizatorului.

Consider că, jocul și-a atins scopul de a ajunge într-o formă similară cu jocurile 2D, side view, action, adventure, platformer, level based, similar cu jocurile precum *Hollow Knight* și *Ori and the Will of the Wisp*. Apropiindu-se cât mai mult de subgenul metroidvenia, datorită faptului că jocul prezintă o hartă non-liniară, jucătorul are posibilitatea de explorare a lumii, în modul lui specific, prin urmare oferind o experiență unică.

Prin utilizarea motorului Unity, am beneficiat de instrumente și funcționalități puternice, permițându-mi să dezvolt într-un mod eficient ajungând la un produs vizual atrăgător, cu mecanici de joc interesante și o interfață intuitivă pentru utilizator.

Proiectul a implicat dezvoltarea unui sistem de meniuri, un sistem de mișcare pentru personajul jucătorului, o inteligență artificială simplă pentru inamici, și un sistem de salvare al progresului jucătorului.

De-a lungul proiectului, am întâlnit provocări și am depășit obstacole care mi-au permis dezvoltarea abilităților în programare și design pentru un joc. De asemenea am lucrat într-un mod optim, în așa fel încât, să obțin o performanță cât mai bună pentru utilizator.

Relevanța temei este evidențiată de impactul industriei jocurilor, în lumea divertismentului, având venituri semnificative precum și o varietate largă de genuri și stiluri de jocuri, jocurile video având capacitatea de a oferi experiențe virtuale variate.

În același timp, jocurile pot ajuta știința, prin oferirea posibilității simulării diferitelor lucruri (cum ar fi comportamentele corpurilor cerești) dar și din punct de vedere filosofic (așa cum a reușit să evidențieze prin influențarea teoriei simulării, care afirmă că lumea

înconjurătoare este defapt simulată).

Reperle istorice menționate, cum ar fi primul joc video creat de William Higinbotham în 1958, apariția subgenului metroidvania bazat pe jocurile *Metroid* și *Castlevania*, sau apariția jocurilor de succes precum *Grand Theft Auto*, *Dark Souls* și *Hollow Knight*, demonstrează evoluția și influența jocurilor video în industrie.

Într-un final, proiectul meu, *The Knight*, mi-a oferit oportunitatea de a explora tehnologiile oferite de unity, în dezvoltarea jocurilor video, dar și oportunitatea de a păstra o gândire creativă pe parcursul dezvoltării proiectului. Sunt mândru de rezultatele obținute și sper ca jocul meu să ofere o experiență cât mai captivantă și memorabilă pentru jucători.

Bibliografie

- [1] Bary Elad, *Film Industry Statistics – By The Distributor, Running Time, Demographic, Box Office Revenue*, <https://www.enterpriseappstoday.com/stats/film-industry-statistics.html> (ultima accesare: 07.06.2023)
- [2] Ifpi, *IFPI Global Music Report: Global Recorded Music Revenues Grew 9% In 2022*, <https://www.ifpi.org/ifpi-global-music-report-global-recorded-music-revenues-grew-9-in-2022/> (ultima accesare: 07.06.2023)
- [3] Tom Wijman, *The Games Market in 2022: The Year in Numbers*, <https://newzoo.com/resources/blog/the-games-market-in-2022-the-year-in-numbers> (ultima accesare: 07.06.2023)
- [4] Brianna Reeves, *Team Cherry Reveals Hollow Knight Sales Have Topped 2.8 Million*, <https://www.playstationlifestyle.net/2019/02/14/hollow-knight-sales-top-two-million/> (ultima accesare: 07.06.2023)
- [5] Ernie Tretkoff, *October 1958: Physicist Invents First Video Game*, <https://www.aps.org/publications/apsnews/200810/physicshistory.cfm>, (ultima accesare: 08.06.2023)
- [6] Ansimuz, *Sunny Land*, <https://assetstore.unity.com/packages/2d/characters/sunny-land-103349>, (ultima accesare: 08.06.2023)
- [7] Sven Thole, *Hero Knight – Pixel Art*, <https://assetstore.unity.com/packages/2d/characters/hero-knight-pixel-art-165188>, (ultima accesare: 08.06.2023)
- [8] Unity Documentation, <https://docs.unity3d.com/ScriptReference/>, (ultima accesare: 08.06.2023)
- [9] Sven Thole, *Bandit – Pixel Art*, <https://assetstore.unity.com/packages/2d/characters/bandits-pixel-art-104130>, (ultima accesare: 08.06.2023)
- [10] Pixel Frog, *Pixel Adventure 2*, <https://assetstore.unity.com/packages/2d/characters/pixel-adventure-2-155418>, (ultima accesare: 09.06.2023)

- [11] Aron Granberg, *A* Pathfinding Project*, <https://arongranberg.com/astar/download>, (ultima accesare: 09.06.2023)
- [12] KartInnka, *Buttons Set*, <https://assetstore.unity.com/packages/2d/gui/buttons-set-211824>, (ultima accesare: 09.06.2023)
- [13] leohpaz, *RPG Essentials Sound Effects - FREE!*, <https://assetstore.unity.com/packages/audio/sound-fx/rpg-essentials-sound-effects-free-227708>, (ultima accesare: 09.06.2023)
- [14] alkakrab, *25 Fantasy RPG Game Tracks Music Pack*, <https://assetstore.unity.com/packages/audio/music/25-fantasy-rpg-game-tracks-music-pack-240154>, (ultima accesare: 09.06.2023)
- [15] Dan Cox, *Using BinaryFormatter in Unity to Save and Load Game Data*, <https://videlais.com/2021/02/24/using-binaryformatter-in-unity-to-save-and-load-game-data/>, (ultima accesare: 09.06.2023)
- [16] Pixel Frog, *Pixel Adventure 1*, <https://assetstore.unity.com/packages/2d/characters/pixel-adventure-1-155360>, (ultima accesare: 14.06.2023)