UNIVERSITATEA BUCUREȘTI

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

LUCRARE LICENȚĂ

~ RPG 3D în Unity pentru PC ~

~ Cammo-Survival ~

ABSOLVENT

Vîlceanu Sonia-Nicoleta

COORDONATOR ȘTIINȚIFIC

Lect. Dr. Ștefan Popescu

REZUMAT

Aplicația noastră este un joc Action-Shooter RPG (Role Playing Game), dezvoltat în platforma Unity, într-un mediu 3D, pentru sistemul de operare Windows.

Jocurile de tip Action-Shooter RPG sunt jocuri ce încorporează acțiunea cu impersonarea unui rol într-o narațiune. Jucătorul controlează direct acțiunile personajului său, și are ca scop supraviețuirea și parcurgerea a mai multor nivele pentru a putea avansa în narațiune.

Tema aleasă este una de actualitate, industria jocurilor video fiind la momentul actual cea mai dezvoltată și căutată formă de divertisment. Printre multiplele avantaje aduse de acest fapt se numără și nivelul ridicat de surse de informare ce sunt puse la dispoziție pe internet, în cărți de specialitate sau în alte medii, ce ne pot îndruma în a construi o astfel de aplicație proprie.

Cum platforma Unity este cea mai des întrebuințată în majoritatea jocurilor actuale și de asemenea ușor de înțeles pentru începători, am decis că ar fi platforma ideală pentru a fi utilizată în conceperea de bază a jocului nostru. De asemenea, am ales mediul 3D în loc de cel 2D deoarece conferă posibilități mai diverse din punct de vedere al abordării implementării, oferă o deschidere mai largă spre realism, creativitate și versatilitate.

Crearea aplicației are ca ideal aducerea unor îmbunătățiri asupra aptitudinilor personale în cadrul programării jocurilor video, dar și explorarea cunoștințelor ce pot fi oferite de platforma Unity.

În acest demers vom concepe o analiză în detaliu asupra procesului de planificare și realizare a unui astfel de proiect, pașii parcurși și dificultățile întâmpinate. Vom dezvolta informații despre platforma Unity și beneficiile prezentate de aceasta față de alte platforme cu întrebuințare similară de pe piață, împreună cu modul în care aceasta se potrivește nevoilor în proiectarea aplicației noastre.

ABSTRACT

Our application is an Action-Shooter RPG (Role Playing Game), developed using the Unity game engine, in a 3D environment, for the Windows operating system.

Action-Shooter RPG games are games that incorporate action by impersonating a role in a narrative. The player directly controls the actions of his character, and aims to survive and go through several levels in order to advance in the narrative.

The elected theme is a topical one, the video game industry being currently the most developed and sought after form of entertainment. Among the many advantages brought by this, is the high level of information sources that are made available on the Internet, in specialized books or in other media, which can guide us in building such an application of our own.

As the Unity platform is most often used in most current games and also easy to understand for beginners, we decided that it would be the ideal platform to be used in the basic design of our game. We also chose the 3D environment instead of the 2D one because it gives more diverse possibilities in terms of implementation approach, offers a wider openness to realism, creativity and versatility.

The creation of the application aims to bring improvements to personal skills in video game programming, but also to explore the knowledge that can be offered by the Unity game engine.

In this approach we will conceive a detailed analysis on the process of planning and carrying out such a project, the steps taken and the difficulties encountered. We will develop information about the Unity engine and its benefits over other similarly used game engines on the market, along with how it fits the needs of our application design.

CUPRINS

INTRODUCERE

Industria jocurilor video a continuat să crească signifiant de-a lungul deceniilor, evoluând în pas cu tehnologia. Amploarea jocurilor video actuale a ajuns să aibă o producție cu o scară pe același nivel cu cea a filmelor și emisiunilor televizate. Costurile de piață și complexitatea proiectelor continuă să se destindă, atât pentru companiile care construiesc un joc cu propriile resurse de la zero, cât și pentru companiile de editoare de jocuri video ce oferă cumpărătorilor un "schelet" pe care aceștia să își dezvolte propriul joc.

Unity este un astfel de editor de jocuri video gratuit, ce necesită bani de la utilizator doar la eventuala lansare pe piață a jocului. Exemple de jocuri de success comercial create în Unity sunt *Rust, Subnautica, Temple Run, Ori and the Blind Forest, Angry Birds* și altele. Alt editor cu mare întrebuințare pe piața este Unreal Engine, ce are un model de gestionare a plății pentru utilizatori asemănător cu cel oferit de Unity. Unreal Engine este cunoscut pentru jocuri precum *Fortnite, Valorant* și *Borderlands*. Printre alte editoare mai rar utilizate se numără și GameMaker și GDevelop. La momentul actual, Unity este platforma cea mai folosită.

Diferențele majore dintre cele două se centralizează pe vastitatea și complexitatea proiectului. Unity este cel mai des întrebuințat pentru jocuri la o scară mai joasă, ce nu solicită mult din punct de vedere grafic și al puterii de procesare. Astfel, Unity este mai des întâlnit la jocurile video pentru platforma mobilă sau de către programatorii de jocuri indie. Jocurile indie sunt jocuri create fie de o companie cu resurse limitate, fie de programatori individuali, care nu au posibilitatea de a pune pe piața un produs pe o scară așa de ridicată ca studiourile ce conduc industria actuală.

Chiar și companiile ce sunt cap de piață mai apelează uneori la editoare de jocuri video pentru a mai tăia din costuri. Spre exemplu, compania Blizzard-Activision a utilizat Unity pentru jocul video *Hearthstone*, deși au capacitatea pentru producție proprie, cum ar fi jocurile din seria *Diablo, World of Warcraft* și *Starcraft*.

Unity este foarte ușor de înțeles pentru începători și cu o gamă largă de opțiuni puse la dispoziție pentru a putea programa o varietate de jocuri pe multiple platforme, atât pentru întreprinzătorii individuali cât și pentru companiile de nivel înalt.

Cum jocurile video acoperă un areal mare de interese și idei, acestea îndeplinesc diferite funcții pentru fiecare jucător în parte. Pentru unii jocurile reprezintă o formă de divertisment ce aduce relaxare și comfort, de asemenea fiind ușor accesibile tuturor, pentru alții înseamnă creativitate, oportunități și provocări ce pot fi doborâte, iar alții sunt atrași de aspectul social, interacționarea cu alți jucători și comunitatea ce se formează în jurul unui joc.

Unul din scopurile aplicației noastre este de a reda o astfel de experiență și de a furniza un mediu ce încurajează autoperfecționarea și competiția într-un mod sănătos.

Am ales ca aplicația noastră să se încadreze la categoriile Shooter, RPG (Role-Playing Game) și Action deoarece acestea sunt cele 3 clase cele mai căutate la jocurile moderne, și am dori ca jocul nostru să se încadreze de asemenea și el aici. Alte jocuri bine cunoscute ce fac parte din aceleași categorii sunt jocurile din seria *Fallout, Destiny 2 și Borderlands 3*.

Structura lucrării, împărțită pe capitole :

Capitolul 1 – Noțiuni preliminarii, viziunea generală asupra structurii proiectului, obiectivului jocului și rezumat al contribuției proprii

Capitolul 2 – Noțiuni teoretice și elemente specifice platformei Unity

Capitolul 3 – Analiză în amănunt asupra contribuției proprie în implementare

Capitolul 4 – Concluzii, greutăți întâmpinate și eventuale îmbunătățiri ce pot fi aduse proiectului

CAPITOLUL 1 : PRELIMINARII

Jocul nostru este o variantă demo ce conține caracteristicile de bază necesare și unei variante finale de joc.

Aplicația este proiectată într-un mediu 3D, iar perspectiva jucătorului este la persoana a 3-a, mai exact nu privim prin ochii personajului nostru, ci de undeva din spate, lucru ce ne permite să ne vedem și personajul efectiv și modul în care se mișcă și interacționează cu împrejurimile sale.

Acțiunea are loc în timp real, jucătorul având control complet asupra manevrelor de mișcare, și a interacțiunilor sale, ceea ce conferă un element de dinamicitate și receptivitate, spre deosebire de jocurile ce au un sistem de luptă bazat pe obligația pusă asupra jucătorilor de a-și aștepta rândul pentru a putea interacționa fie unul cu celălalt, fie cu mediul hărții.

Conform categoriei Role-Playing Game, personajul nostru are o poveste. A fost trimis într-o misiune de recunoaștere într-o zonă umblată de oameni afectați de o boală necunoscută. Acesta trebuie să supraviețuiască până ajunge la punctul final de plecare ce îi oferă siguranță. Pe parcursul nivelelor va descoperi pe bucăți câteva informații despre situația locului în care se află.

Jocul încurajează explorarea, această activitate fiind răsplătită prin informații adiționale asupra poveștii cât și bonusuri ascunse pe hartă ce ne vor ajuta să trecem mai ușor nivelele.

Constituit din 3 nivele, fiecare nivel cu o hartă individuală pe care jucătorul se poate deplasa într-un mod liniar pentru a continua să progreseze povestea. Fiecare nivel are o dificultate mai ridicată decât precedentul iar acestea trebuie parcurse în ordine atât pentru ca povestea să păstreze o logică narativă, cât și pentru ca jucătorul să simtă că menține un progres constant în joc.

Pentru a se debloca nivelul următor este nevoie ca jucătorul să colecteze o cheie specială aflată la finalul hărții nivelului curent.

Limbajul C#

Unity este scris în limbajul C#, limbaj ce utilizatorii îl și folosesc pentru programarea aplicațiilor pe această platformă.

Unity obișnuia să ofere posibilitate de programare și în alte două limbaje : Boo, ce este foarte similară cu Python și Unityscript, ce este o variantă de Javascript. În prezent acestea au fost scoase din folosință, întrucât doar o mică parte din utilizatori le foloseau, iar C# a devenit un limbaj tot mai puternic, mai ales cu dezvoltarea ASP.NET. C# este de asemenea mult mai atractiv pentru utilizatori datorită faptului că ușurința sa de învățare îi conferă accesibilitate. Acesta este un alt avantaj prezentat asupra platformei Unreal Engine, deoarece aceasta întrebuințează C++, un limbaj mai greu de deprins decât C#.

Asset Store

Unity pune la dispoziție o colecție de librării online numită Asset Store, la care pot contribui și membrii comunității. Asset-urile sunt elemente ce pot fi create fie din Unity (Animator Controller, Avatar Mask, Audio Mixer etc.), fie importate din afara Unity (imagini, animații FBX, fișiere audio, modele FBX etc.), ce pot fi folosite într-un proiect.

Am importat librăria StandardAssets pentru a putea utiliza mai multe astfel de asset-uri în scopul de a putea popula mai ușor scenele din joc cu decorațiuni.

Obiectivul lucrării este de a utiliza platforma Unity de editare de jocuri video pentru a putea crea în limbajul C# o aplicație ce urmează exemplele jocurilor indie de pe piață și oferă utilizatorilor o formă căutată de divertisment, deschizând interesul mai ales spre genul Action – Shooter RPG.

Contribuția proprie se rezumă la construirea si decorarea hărții cu obiecte de scenă, crearea jucătorului și programarea intracțiunilor acestuia cu mediul înconjurător, mai exact a altor personaje și a unor obiecte colectabile, implementarea unui sistem de progresare pe nivele și scor și alcătuirea unei narațiuni pentru povestea personajului.

Etapele dezvoltării

1. Prima etapa a fost construirea modelului 3D al jucătorului și a armei în programul Blender, atât mesh cât și schelet. Apoi am creat texturi în Adobe Photoshop pentru acestea. Urmată de încărcarea modelului pe platforma Mixamo pentru a face rost de animații, descărcarea unui model si a unor animații și pentru inamici și importarea tuturor în Unity, în folderul Assets.

2. Construirea hărților nivelelor într-o măsură mai simplistă și plasarea unor obstacole (scări, ziduri) în scenă împreună cu modelul jucătorului pentru a pregăti terenul pentru testare.

3. Implementarea codului pentru deplasarea și animarea jucătorului (mișcare pe toate direcțiile, sărire, atac) și interacțiunea cu obiectele solide din scenă precum coliziunea cu pereții și podeaua, urcarea scărilor, împingerea obiectelor etc.

4. Implementarea codului pentru deplasarea și animarea inamicilor, împreună cu interacțiunea mediului înconjurător și cu jucătorul, conformându-ne cu opțiunile AI-ului pus la dispoziție de Unity, Nav Mesh.

5. Sistemul de viață pentru inamici și jucători, adăugarea animațiilor de năucire și moarte pentru ambii, crearea UI-ului și implementarea statisticilor precum scor, muniție și puterea de tragere a armei.

6. Crearea bonusurilor colectabile și întrebuințările fiecărora dintre ele și a cheilor ce fac tranzițiile între nivele.

7. UI-ul tuturor meniurilor, de început, mijloc și sfârșit de joc.

8. Implementarea Load și Save.

9. Finisarea detaliilor de pe hartă, decorarea în amănunt a scenelor

CAPITOLUL 2 : NOȚIUNI TEORETICE

2.1 COMPONENTE ȘI COLLIDERE

Unity pune la dispoziție numeroase funcționalități prin care se poate defini și controla comportamentul obiectelor într-un joc. Cele mai des folosite metode implică utilizarea componentelor.

Componentele sunt particularități ce pot fi adăugate unui obiect (*GameObject*), și care îi vor întrebuința o anume funcționalitate acestuia. Ele pot fi modificate atât din cod cât și direct din meniul lor din *Inspector*.

2.1.1 Transform. Aceasta determină poziția obiectului pe cele 3 axe X, Y, Z, împreună cu rotația în jurul celor 3 axe și scara de mărime a obiectului de-a lungul axelor. Spre deosebire de alte componente, *Transform* poate fi modificată și în *Scene View*, prin click-and-drag pe săgețile *gizmo*-ului afișat. Fiecare obiect are în mod implicit o componentă *Transform*.[1]

Când un obiect are un părinte, valorile din componenta *Transform* a copilului vor deveni coordonate locale, întrucât se vor raporta la poziția părintelui și nu la coordonatele globale.[1]

2.1.2 Componenta Collider poate fi de 3 tipuri în Unity 3D: *Capsule, Box* și *Sphere*. Acestea au rolul de a permite coliziuni între obiectele pe care sunt plasate și alte obiecte din jur.

Pentru a verifica prin evenimente coliziunea acestui component cu alte obiecte în cod este necesară marcarea opțiunii *is Trigger*. Iar pentru ca două obiecte cu componenta de tip *Collider* și opțiunea aceasta bifată să pornească un eveniment este nevoie ca măcar unul să conțină și un component *RigidBody*.

Un *Collider* este vizibil și în *Scene View* datorită faptului că marginile sale sunt reprezentate printr-un *Gizmo*, însă acesta nu poate fi modificat. [2]

Funcții pre-definite oferite de Unity prin care putem să controlăm coliziunile:

OnCollisionEnter() - se apelează pe un obiect cu *collider* care intra în *collider*-ul altui obiect.[3]

OnTriggerEnter() - se apelează pe un obiect cu *collider* și opțiunea *is Trigger* bifată când un alt obiect cu *collider* intră în *collider*-ul celui din urmă.[4]

2.1.3 Rigidbody este o componentă care odată plasată pe un obiect, va permite obiectului să fie afectat de forțe fizice precum gravitație, inerție etc., în scopul de a simula un mod realistic de mișcare. Chiar dacă obiectul ce are *RigidBody* are un părinte, forțele fizice aplicate copilului nu vor fi inhibate de părinte. Spre exemplu, gravitația va trage copilul în jos chiar dacă părintele rămâne pe loc, însă dacă părintele este mutat, copilul tot îl va urma.

Dacă un obiect are ambele componente *RigidBody* și *Collider* atunci la o coliziune cu un alt obiect cu componenta *Collider*, cele doua obiecte pot să ricoșeze.

Aceasta componentă are mai multe opțiuni pentru customizarea forțelor fizice ce i se aplică, printre care : *Mass* (controlează greutatea obiectului), *Drag* (rezistența la aer a obiectului), *Use Gravity* (permite oprirea și pornirea gravitației), *is Kinematic* (obiectul va trebui să aibă modificată prin cod componenta *Transform* pentru a se putea mișca, întrucât nu mai e afectat de forțele fizice oferite de Unity) [5].

2.1.4 Character Controller. Cel mai des folosit pentru controlul unui personaj, fie third-person, fie first-person. Are scopul de a înlocui componenta *RigidBody* pentru obiectele pentru care dorim un mod de mișcare mai dinamic, deci care să nu fie restricționat de forțe fizice.[6] Spre exemplu, jucătorul ar putea avea nevoie să ia viraje la stânga când aleargă cu o viteza mare. Dacă ar avea o componenta *Rigidbody* acesta ar face mai greu întoarcerea la stânga din cauza inerției creată de viteza ridicată.[7]

Aceasta componenta este în esențial un Capsule Collider dar care poate fi instrucționat din cod modul în care dorim să se deplaseze. Acest lucru include și ricoșarea altor obiecte la coliziunea cu aceasta.[7]

2.1.5 Componente de text. În mod implicit Unity oferă componenta *Text*, care pentru a o utiliza este necesară introducerea în cod a *namespace*-ului *" using UnityEngine.UI; "*.

În schimb, pentru toate instanțele de text am folosit pentru un maxim de claritate posibil pachetul *TextMesh Pro*. Pentru a-l importa se poate naviga la *Windows -TextMesh Pro - Import*, sau direct la crearea unui obiect de tip *UI - Text - Text Mesh Pro*, selectăm *Import TMP Essential* din fereastra ce apare. [8] Crearea unui obiect în acest fel va crea automat și un obiect *Canvas* în care obiectul text va fi plasat ca copil, doearece toate obiectele de tip UI se recomandă a fi plasate într-un Canvas. Pentru a lucra cu el în cod trebuie să fie introdus namespace-ul *" using TMPro; "*.

Pachetul TextMesh Pro oferă opțiuni mai diverse de stilizare a textului, incluzând shadere personalizate (spre deosebire de componenta *Text*) și metode mai înalte de Text Rendering, ce produc o imagine de mai mare calitate. [9]

Se pot accesa două tipuri de componente: *TextMeshPro* și *TextMeshProUGUI*. Primul este special făcut pentru a lucra cu un *Mesh Renderer*, pe când cel de-al doilea e recomandat pentru lucrul cu *Canvas Renderer*.

Un obiect de tip text mereu are dat implicit o componentă *RectTransform*. Acesta este un *Transform* deseori folosit pentru obiectele de tip *GUI* (forma 2D, dreptunghiulară).

2.1.6 Componenta Canvas. Crearea oricărui element de tip UI : *Image, Text, Button* etc. va crea automat și un obiect *Canvas* pe care acestea vor fi plasate pentru a le putea afișa în joc. Obiectul Canvas conține și o componentă *Canvas*. Acesta are 3 moduri posibile de afișare (Render Mode): *Screen Space - Overlay* (canvas-ul va fi afișat fix pe spațiul ecranului, indiferent de locul de plasare al acestuia în scenă, *Screen Space - Camera* (canvas-ul va fi afișat la o distanță anume față de o cameră selectată) și *World Space* (va fi plasat și vizibil în scenă, la fel ca majoritatea obiectelor)[10]

2.1.7 NavMeshAgent este o componentă des folosită pentru AI, care controlează obiectele pentru a naviga pe un *NavMesh* desemnat. Acesta știe să evite și alte obiecte cu această componentă.

NavMesh este un mesh care decide ce areale sunt navigabile și ce areale sunt obstacole de ocolit pentru obiectele cu componenta *NavMesh Agent*. Pentru a-l modifica mergem la *Window - AI - Navigation*. După ajustarea setărilor dorite, se apasă *Bake* pentru a îl construi. [12]

Componenta permite setarea unei destinații pentru obiect spre care să caute constant să se îndrepte. Algoritmul folosit este *A**. Se poate seta atât din cod cât și din *Inspector* viteza de mișcare, distanța de oprire, accelerația și altele.[11]

2.1.8 Componenta Animator. Permite atribuirea de animații unui obiect, necesitând pentru aceasta legarea de un *Animator Controller*. Câteva opțiuni utilizate sunt *Avatar* și *Culling Update*.

Culling Update este o opțiune cu scopul de îmbunătățire a performanței care decide cum sunt animate obiectele din afara câmpului vizual al camerei. *Always Update* - animațiile sunt mereu în desfășurare, indiferent dacă obiectele sunt prezente pe ecran sau înafara ecranului. *Cull Update Transforms* - animația se oprește însă Unity continuă să calculeze *frame*-urile pentru aceasta pentru ca atunci când revine în câmpul vizual al camerei să continue cu poziția la care ar fi fost și dacă continua să ruleze normal. *Cull Always* - oprește complet animația când *Camera* nu îl are pe ecran, iar în cazul revenirii în dreptul ecranului se va continua de unde a rămas. [13]

Avatar. După importarea unui model FBX, dacă acesta nu vine cu un avatar predefinit, putem crea noi unul. Pentru asta în fereastra *Rig* a modelului, vom preciza tipul său de schelet : *Humanoid* sau *Transform/Generic*. Avatar-ul nu este obligatoriu dar este recomandat.[14]

Un **Animator Controller** este un Asset cu scopul de a gestiona animațiile, cum se îmbină și ordinea și modul în care sunt declanșate, eventual în funcție de condiții impuse de parametrii și *layer*-ele pe care sunt împărțite animațiile.

2.1.9 Componenta Script se poate creea și exista separat de obiecte, însă trebuie plasată pe un obiect pentru a putea fi rulat codul din aceasta. Script-ul este scris în C# și folosește colecții de clase, numite *namespace*, care sunt cheie pentru utilizarea platformei Unity. Cel mai des utilizat *namespace*, ce e mereu inclus în mod implicit, este UnityEngine.

Tipurile de variabile utilizate sunt clase/noțiuni predefinite: *GameObject, LayerMask, Vector3* etc. ori componente ce le-am adăugat noi în *Inspector: Transform, Text, Character Controller*, clase personalizate etc.

2.2 FUNCȚII EVENIMENT

Funcțiile eveniment sunt : *Awake, On Enable, Start, FixedUpdate, Update, Late Update, On Diqable, On Destroy*, apelate în ordinea aceasta. Cele folosite în proiectul nostru sunt *Start* si *Update*.

Funcția Start - este apelată doar dacă componenta *Script* ce e plasată pe obiect este activată. Aceasta rulează doar o dată la începutul jocului, înainte de orice *frame* din joc, deci înainte de toate funcțiile *Update*. Ceea ce înseamnă că nu va rula dacă în timpul jocului un *prefab* cu componenta *Script* se instanțiază.

Funcția Update - se apelează o dată pentru fiecare *frame* în parte. Funcția *Late Update* se apelează în același mod, însă mereu după ce se termină mai întâi de rulat *Update*.

Funcția Awake. Este o eventuală alternativă pentru funcția *Start*. Aceasta este apelată în două cazuri: după ce se instanțiază un obiect ce conține script-ul cu funcția (un *prefab*) și mereu înainte de apelarea funcțiilor *Start*, atunci când script-ul este încărcat. Aceasta se apelează și dacă componentul *Script* în care se află este inactiv.

Funcția FixedUpdate - se apelează la un interval fix, de obicei aceasta se întâmplă să fie mai des apelat decât se apelează *Update*, dar nu mereu. Aceasta funcție face ca indiferent dacă jocul rulează

cu un număr mic sau mare de *frame*-uri, codul să fie mereu asigurat să ruleze la intervalele egale de timp.[15][16]

2.3 ANIMAȚII

Animația este un proces ce are ca scop operarea diferitelor componente din scheletul unui model 3D pentru a încerca să imite o mișcare realistă. Pentru animarea unui obiect o componentă *Animator* este obligatorie.

Avatar Mask este un *Asset* folosit la izolarea anumitor părți ale scheletului obiectului pentru rularea a diferite animații pe fiecare parte izolată. Acesta se selectează pentru un *Animation Layer*, din fereastra specifică a unui *Animator Controller*, pentru ca layer-ul sa știe cum să separe stările de animație. [17]

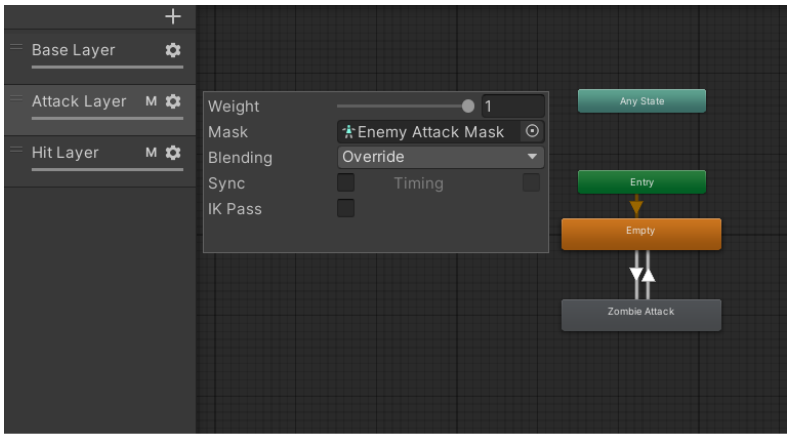

Figura 1

Un ***Animation Layer*** oferă două tipuri de combinare a animațiilor. *Override*, ce va bloca animațiile desfășurate pe alte layere, pentru a permite doar animațiile de pe layer-ul curent (doar pe bucățile de schelet ce se suprapun) și *Additive*, ce va adăuga animația curentă de pe acest layer deasupra animațiilor de pe celelalte layere. [18]

Un **Animator Controller** nou are în mod implicit un layer de bază (*Base Layer*) ce are 3 stări: *Any State*, *Entry* și *Exit*.

Orice animație plasată în *Controller* devine o stare, și în mod automat prima animație plasată se va lega de starea *Entry* printr-o tranziție. Aceasta devine implicit starea de bază a layer-ului (*Layer Default State*), distinsă printr-o culoare portocalie. Starea de bază este starea ce se declanșează automat la început de joc. Cel mai des, starea de bază este o animație "*Idle*".

Animațiile pot fi făcute să se redea doar o dată sau la infinit într-o buclă, prin setarea *loop Time* din *Inspectorul* animației. În cazul din urmă, animația va continua să fie activă până când este întreruptă de altă animație. [19]

Tranzițiile sunt legături între stări care reglementează ordinea, condițiile și modul în care animațiile pot fi activate, precum și durata acestora. Activitatea unei stări poate fi controlată prin parametrii *Int, Float, Bool, Trigger*, și prin implementarea în cod a modificării valorilor acestora. În *Inspectorul* tranziției se poate selecta parametrul care să îi controleze activitatea.[20]

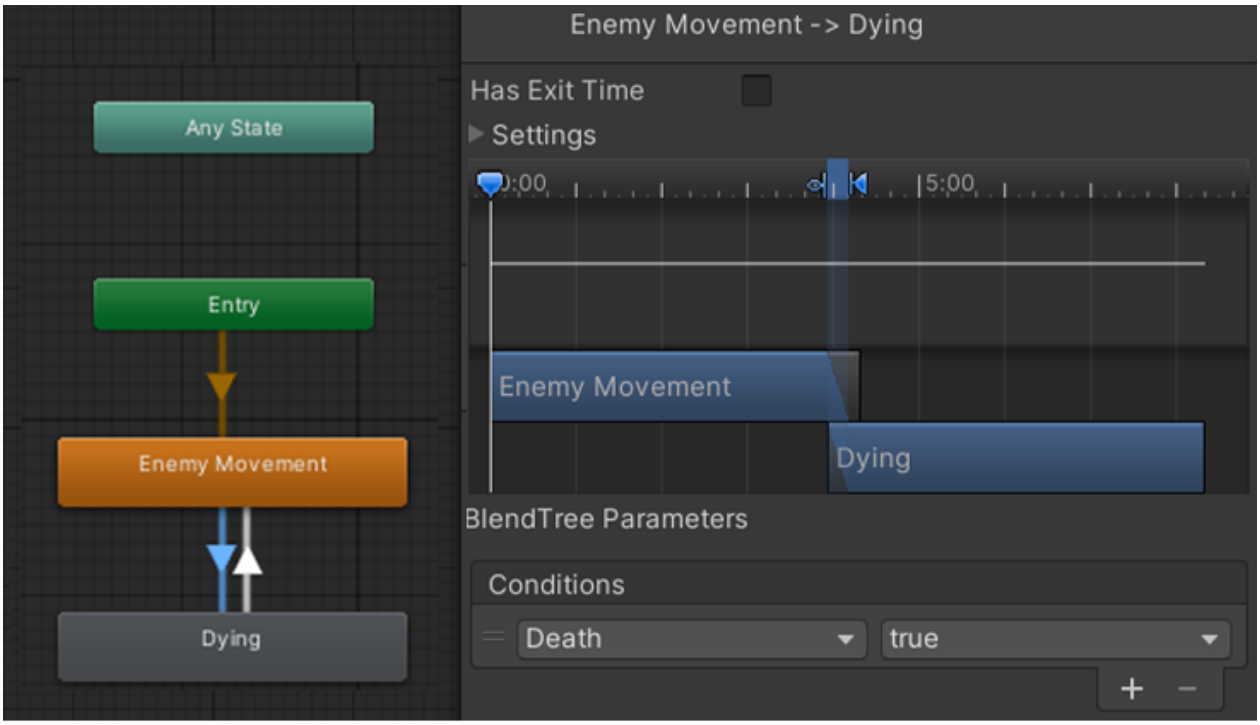

Figura 2

Fără parametri animațiile se vor declanșa mereu fiecare la terminarea precedentei, în ordinea tranzițiilor. Terminarea unei animații are loc la *Exit Time*, care în mod implicit are valoarea egală

cu durata de timp a animației, însă poate fi modificată ca animația să se termine mai devreme sau mai târziu, caz în care este nevoie ca animația sa aibă buclă. Spre exemplu, *Exit Time* cu o valoare de 0.45 înseamnă că animația se va opri când ajunge la un procent de 45% parcursă, trecând apoi la animația următoare. Dacă *Exit Time* are o valoare de 2.7, animația va trece de 2 ori prin buclă și apoi la a treia trecere se va opri când e parcursă în procent de 70%.

Dacă opțiunea *Has Exit Time* este bifată, adăugarea de parametri nu va înlocui condiția *Exit Time*, ci se vor evalua în plus fața de aceasta, deci tranziția spre următoarea animație are loc doar când toate condițiile sunt îndeplinite. Dacă dorim însă ca singurele condiții asupra animației să fie cele definite prin parametri, și nu și de *Exit Time*, atunci putem debifa opțiunea *Has Exit Time*. [20]

Blend Tree. Pe lângă stările simple formate dintr-o singură animație, există și stări formate dintr-o agregare de animații, denumite *Blend Tree*. Acestea au rolul de a combina într-un mod uniform mai multe bucăți din animații cât-de-cât asemănătoare în mișcări (exemplu: stat pe loc -> mers -> alergat), spre a crea o tranziționare dinamică și naturală între ele. Un *Blend Tree* poate fi de două tipuri, 1 și 2-dimensional [21].

- **Blend Tree 1-dimensional**. Odată cu crearea stării avem creat și un parametru *Blend* de tip *Float*. *Blend Tree*-ul ia valori pe intervalul 0 și 1, care va fi împărțit în atâtea "segmente" câte animații am pus în *Blend Tree*. Exemplu: dacă avem 3 animații : stat pe loc, mers și alergat. Când parametrul *Blend* are valoare 0, animația activă este cea de stat pe loc, când valoarea este 0.5, animația va fi cea de mers, iar pentru valoarea 1, animația este cea de alergat. Dacă avem însă valoarea 0.66 animația de o vedem este o alergare ușoară, mai rapidă decât mersul, dar mai înceată decât alergarea. [21][22]

- **Blend Tree 2-dimensional**. Când un *Blend Tree* e creat, el este mereu 1-dimensional. Pentru a-l schimba în 2D, vom selecta Blend Type -> 2D. Unity oferă 3 tipuri de blend tree 2D : *Simple Directional, Freeform Directional* și *Freeform Cartesian*. Diferența între acestea este modul în care tranziționarea dintre animații este calculată. [21][23]

Simple Directional este făcut pentru o singură animație pe o direcție în linie dreaptă a grafului. Nu e indicat pentru animații de tipul mers - alergare.

Freeform Directional este ideal pentru deplasarea unui personaj, precum jucătorul nostru, pe un plan 2D orizontal, permițând mișcări față-spate, stânga-dreapta și pe cele patru diagonale.

Freeform Cartesian este proiectat în special pentru animații pe un plan 2D vertical. Un exemplu ar fi animația unei mâini ce indică o locație în fața sa (jos-sus, stânga-dreapta și diagonalele).

Direct. Ultima opțiune prezentă în *Blend Type*, cel mai des utilizat pentru animație facială.[23]

Acesta are nevoie de 2 parametrii de tip *Float*, câte unul pentru fiecare axă, X și Y. În *Inspector* va apărea graful ce reprezintă câmpul de mișcare al animațiilor. Culoarea albastră deschis de pe graf indică zonele "acoperite" de animații. Cu cât nuanța e mai deschisă, cu atât animația va fi mai "slabă". Punctul roșu din mijloc reprezintă valoarea curentă a celor doi parametrii float, iar celelalte puncte albastre reprezintă animațiile adăugate de noi. Dacă plimbăm punctul roșu aproape de cele albastre, arii circulare se formează în jurul punctelor albastre. Acestea indică nivelul de influență al animațiilor specifice punctelor albastre asupra animației curente a punctului roșu.[23]

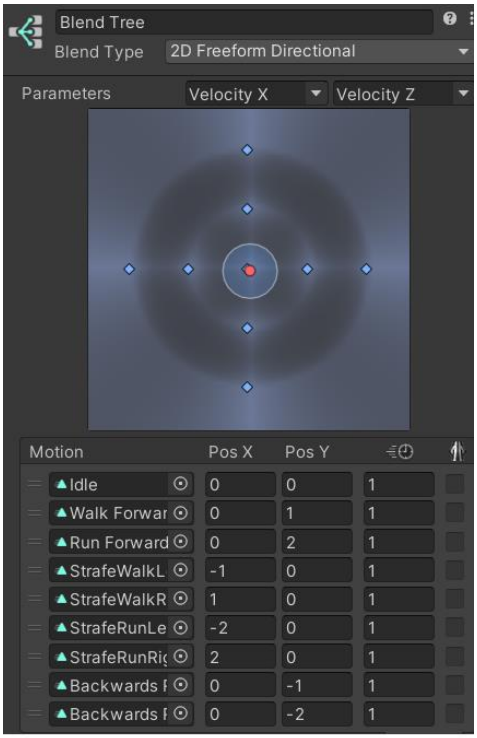

Figura 3

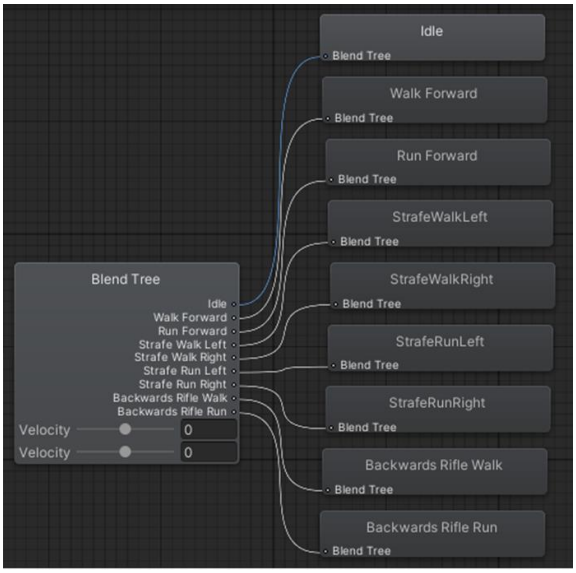

Figura 4

2.4 SCENE

Scenele sunt elemente folosite pentru a secționa jocul în părți bine definite, fiecare cu un rol individual. O scenă poate fi folosită pentru meniu, alta pentru înregistrare în joc, altele pentru fiecare nivel etc.

În mod implicit Unity ne pune la dispoziție o scena nouă denumită *SampleScene* la fiecare creare nouă de proiect, ce este lipsită de corpuri fizice, însă conține obiecte esențiale unui joc precum lumina (*Directional Light*) și camera principală (*Main Camera*). [24]

Scenele sunt tratate ca *Asset-uri* și au propriul folder definit, numit *Scenes*. Pentru operarea scenelor tebuie importată biblioteca *Unity.SceneManagement*. Tranziția unei scene se face prin instrucțiunea *SceneManager.LoadScene("numele scenei");* Încărcarea unei scene cu această comandă nu va face însă ca scena să fie și imediat activă, ci va aștepta până se ajunge la următorul *frame*. Ceea ce înseamnă că nu putem accesa în aceeași apelare de funcție obiectele din scena de

am încărcat-o fără să folosim un *Coroutine*. Pentru ca un *Coroutine* să aștepte un *frame* folosim *yield return null*. [25]

Scenele nu sunt înregistrate în joc de la sine, ceea ce va face ca o scenă să fie tratată ca inexistentă de către program. Pentru a înregistra o scenă trebuie să o introducem manual în *File -> Build Settings -> Add Open Scenes* când avem scena respectivă activă. Jocul nostru are patru scene: trei de meniu și una pentru nivele.

CAPITOLUL 3 : CONTRIBUȚIA PROPRIU-ZISĂ

3.1 SCENE SI MENIURI

3.1.1 Scena Menu - are 3 ferestre subcomponente : meniul principal, meniul de opțiuni și meniul de nivele.

Meniul principal - 3 butoane, butonul Play care va deschide meniul pentru alegerea nivelului, butonul Options care va deschide meniul de opțiuni și butonul Quit care va închide jocul.

Meniul de opțiuni - avem butoane ce ne oferă opțiune de intrare și ieșire din modul Fullscreen, posibilitatea de alegere a calității grafice dintr-o listă, ajustare a volumului și buton de întoarcere la meniul principal.

Figura 5

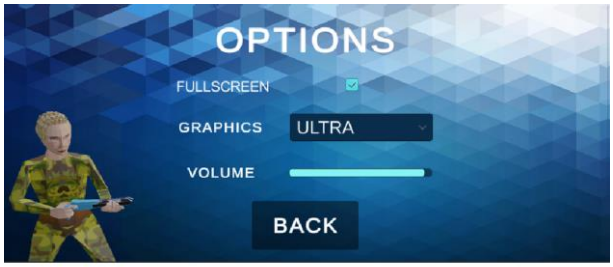

Figura 6

- **Volum** - ajustabil printr-un component *Slider*. Pentru a controla volumul în sine, trebuie să legăm input-ul din *Slider* de un *Audio Mixer*. Acesta este un *Asset* de bază pus la dispoziție de Unity. Un *mixer* permite împărțirea sistemului audio pe canale, spre exemplu să putem să modificăm nivelul muzicii dar nu și a sunetelor ambientale. În proiectul nostru avem un singur canal pentru toate sunetele, cel principal numit *Master*. Pentru a putea modifica valorile sale în cod trebuie să importăm librăria *UnityEngine.Audio*. [26]

- **Calitatea grafică** - are patru opțiuni: *Ultra, High, Medium și Low*, afișate intr-o componentă *Dropdown*. La setarea lor a trebuit să ne asigurăm ca acestea corespund cu setările grafice furnizate de Unity (aflate în *Edit -> Project Settings -> Quality*). Schimbarea calității durează câteva secunde până să fie vizibilă, atât în joc cât și în fereastra *Quality*.
- **FullScreen** - opțiunea e selectabilă folosind o variabilă *bool*, dintr-o componentă *Toggle*. Schimbarea aceasta nu e vizibilă decât după ce am construit jocul (*Build*), nu și în editor-ul Unity.

Meniul de nivele - are butoane pentru toate cele 3 nivele și unul de întoarcere la meniul principal. Dacă jucătorul dorește să parcurgă jocul în varianta completă, trebuie să pornească de la nivelul 1, întrucât dacă alege să înceapă de la nivele mai înalte nu îi am oferit posibilitatea de a se întoarce la nivele mai mici. Acest fapt a fost implementat cu scopul de a preveni jucătorul de la a avea un avantaj incorect și a obține mai ușor un scor ridicat, deoarece dificultatea crește odată cu nivelul, deci progresul obținut din bonusuri îl va ajuta să parcurgă mult mai ușor nivelul precedent, ceea ce nu intenționăm să permitem.

3.1.2 Scena DeathScreen

- este afișat pe partea stângă scorul curent, cel mai mare scor (*Highscore*) și timpul în minute și secunde cât a durat jocul curent.

Figura 7

Pe partea dreaptă se află butoane pentru ieșire din aplicație, reîntoarcere la meniul principal și opțiune directă de reîncepere a unei noi sesiuni de joc.

Highscore - cum Unity resetează variabilele la fiecare sesiune nouă de joc, variabila ce păstrează scorul record ar fi mereu reinițializată cu valoarea 0.

Pentru a putea implementa un scor record, folosim clasa *PlayerPrefs* ce oferă posibilitatea de stocare a variabilelor în sistem (utilizează o cheie de tip *string* unică aleasă de noi pentru a le indentifica), pentru a putea fi utilizate de-a lungul mai multor sesiuni de joc. *PlayerPrefs* nu poate însă să rețină decât variabile de tip *Int, Float* și *String*. [27]

Timpul îl măsurăm prin adunarea succesivă a *Time.deltaTime* pentru fiecare frame, într-o funcție *Update*. Acesta este numărul total de secunde cât a rulat jocul. Nu luam în calcul și timpul în care jocul se află în stare de pauză.

3.1.3 Scena FinalScreen - asemenea scenei precedente, însă în partea stângă mai este prezentă și o opțiune de resetare a scorului record.

3.1.4 Scena SampleScene - Scena principală în care rulează jocul efectiv. Aceasta a fost populate cu obiecte decorative din *Asset Store*: copaci, pietre, clădiri, jucătorul efectiv, alte personaje, bonusuri colectabile etc.

Pe ecranul acesteia apar informații precum bara de viață a jucătorului, scor, muniție (*ammo*) și puterea armei (*damage*).

Figura 8

Scorul curent crește cu doborârea fiecărui inamic și colectarea de bonusuri.

"**Ammo**" este numărul curent de gloanțe pe care jucătorul îl are. La începutul jocului se vor primi mereu 30 de gloanțe. Restul vor trebui colectate de pe hartă sau de la inamicii doborâți. Când muniția ajunge la zero, va apărea o imagine de atenționare iar textul devine roșu pentru a intra mai ușor în atenția jucătorului faptul că a rămas fără muniție. La recâștigarea de muniție, avertismentul dispare.

"**Damage**" este nivelul de putere curent al armei, acesta are o valoare de start de 10 damage și poate crește prin colectarea de upgrade-uri pentru armă, fie de pe hartă, fie de la inamicii doborâți.

Bara de viață - conține un component *Slider* a cărui valoare e ajustabilă printr-un script ce se leagă printr-o variabilă la viața jucătorului.

Aceasta scenă mai conține și alte trei ferestre ce fac parte din meniu, dar nu au fost plasate în scena *Menu* pentru o operare mai ușoară a codului.

Alte meniuri de joc :

Meniul de încărcare de joc. Cu opțiune *Load*, pentru încărcare a ultimului joc salvat în sistem, fie din sesiunea curentă, fie dintr-o sesiune precedentă și o opțiune *New Game* ce pornește un joc nou cu progresul începând de la zero.

Meniul de pauză. Poate fi accesat prin apăsarea tastei *ESC*. Avem opțiuni pentru ieșire totală din joc, salvare a progresului, ieșire la meniul principal și revenire în joc.

Meniul intermediar al nivelelor. Face tranziția de la un nivel la altul, oferind și opțiuni de salvare, ieșire și revenire la meniul principal.

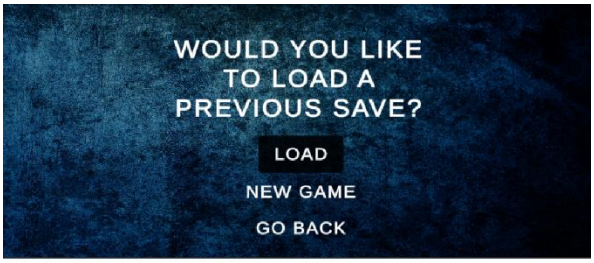

Figura 9

Figura 10

Când intrăm în aceste meniuri cursorul este deblocat *(Cursor.lockState = CursorLockMode.Locked*) pentru a putea interacționa cu butoanele. Jocul este pus în pauză prin instrucțiunea *Time.timeScale = 0f,* ce printr-un *Float* controlează viteza de desfășurare a evenimentelor fizice vizibile de pe ecran: animații, mișcare, alte operații fizice (ex.: gravitație) etc. Valoarea 0 este oprire totală, valoarea 1 este rulare la viteză normală, valoare 2 este rulare la o viteză de două ori mai rapidă. De observat este că *Time.timeScale* nu oprește execuția programului, *frame*-urile continuă să altereze, deci funcțiile *Update* încă sunt apelate.

Pentru a pune în funcțiune butoanele ce vor face tranziționarea între scene și între ferestrele din aceeași scenă, folosim *event*-ul *On Click ()* din *Inspectorul* butonului. La acest event se pot adăuga acțiuni ce dictează ce se întâmplă la apăsarea butonului.

- Pentru tranziția între scene este nevoie de un script cu o metodă ce conține funcția *LoadScene*. La *On Click ()* vom alege obiectul ce conține script-ul și apoi funcția respectivă.

- Pentru tranziția între ferestre la *On Click()* vom alege două obiecte ce reprezintă ferestrele între care vom schimba. Una va fi activată, iar cealaltă dezactivată prin funcția *SetActive().*

3.2 ÎNCĂRCARE ȘI SALVARE A PROGRESULUI

O metodă de salvare a progresului este prin folosirea fișierelor de format simplu precum *JSON* sau *XML*, care, fiind ușor de modificat, oferă un avantaj programatorului, încă acest lucru este și dezavantajul lor, deoarece abilitatea de modificare ușoară a conținutului lor nu oferă securitate, deci nu ideale pentru salvarea progresului întrucât fișierele ar putea fi modificate și de jucător.

Metoda utilizată în proiect implică fișiere binare. Fișierele binare sunt greu de descifrat și prin urmare și de modificat, ceea ce oferă securitate.

Am implentat două clase *PlayerData* și *SaveSystem* ce au ca scop transmiterea, stocarea și serializarea/deserializarea datelor din fișierul binar. Cele două nu moștenesc *MonoBehaviour* întrucât nu o să fie plasate pe niciun obiect din joc.

3.2.1 Clasa *PlayerData* va avea ca câmpuri variabilele ce dorim să le salvăm: scorul, poziția jucătorului, câtă muniție și viață are și puterea armei. Pentru ca datele din aceste câmpuri să fie transformate în fișier binar, trebuie să definim clasa ca *Serializable*.

Clasa va permite doar date de tip de bază*: Int, Float, Bool, String* și nu și date specifice Unity precum *Vector3* sau *Color*, din cauza faptului că aceste tipuri de date nu sunt serializabile. [28]

Aceste tipuri de date pot fi însă reprezentate prin tipuri de date de bază. Spre exemplu, tipul *Vector3* are ca corespondent un vector cu 3 elemente ce stochează date de tip *Float*. Tipul *Color* de asemenea poate fi înlocuit de un vector de lungime 3 ce stochează valori *Int* pentru fiecare canal R, G, B de culoare și eventual un vector de lungime 4 ce ar stoca și canalul *Alpha* ce controlează transparența imaginii.

3.2.2 Clasa *SaveSystem* folosește un *Binary Formatter* (din librăria *System.Runtime.Serialization.Formatters.Binary*). Scopul acestuia este să citească datele din orice

clasă îi transmitem și să le transforme în date binare. Vom crea în cod un nou fișier în care acesta va pune codul.

Path-ul unde dorim să plasăm fișierul poate fi definit ca "C:/System/..." însă acest lucru este viabil dacă programul va rula doar pe mașină curentă, deoarece alte mașini au altfel denumite directoarele, ceea ce va face să nu fie găsit fișierul. Pe lângă aceasta, path-urile depind complet de sistemul de operare, *Windows* are altfel de structurare a directoarelor față de *Machintosh* sau *Android*. Unity oferă o funcție numită *Application.persistentDataPath* ce oferă o locație care nu va fi influențată de sistemul de operare.

Pentru a repune aceste date în joc folosim tot *Binary Formatter* pentru a deschide fișierul și a converti datele din binar înapoi în tipurile de date din clasa *PlayerData*.

Urmează apoi să atribuim datele din *PlayerData* înapoi la obiectele specifice din joc.

3.3 INAMICI

Inamicii sunt împărțiți în grupuri de diferite mărimi, în diverse locații pe hărțile celor tuturor 3 nivele. Cu cât se avansează mai mult pe fiecare nivel și de la un nivel la altul, cu atât mai mult dificultatea continuă să crească. Prin dificultate crescută se înțelege bara de viață a inamicilor crescând tot mai mult, împreună cu puterea de lovire a acestora (damage) și cu viteza lor de deplasare și atacare. Numărul de inamici poate de asemenea să crească .

Un inamic are 5 stări: repaus (*Zombie Idle*), alergare (*Running*), atac (*Zombie Attack*), năucire (*Reaction*) și moarte (*Dying*), ce au și fiecare câte o animație atribuită. Acestea sunt puse în acțiune în următoarele situații:

Starea de repaus - când inamicul nu are un jucător la o distanță de 50 unități în jurul său, și prin urmare nu se deplasează, acesta se va afla în starea de repaus.

De la începutul jocului și până un jucător interacționează cu acesta, inamicul va sta în același loc. Inamicii au o rază de angajare în atac de 50 de metri, mai exact dacă jucătorul se află într-o rază de maxim 50 de metri distanță de inamic (inamicul fiind punctul de centru), atunci inamicul se va deplasa spre jucător pentru a-l ataca. Dacă jucătorul iese apoi din raza de angajare a inamicului, atunci acesta va renunța la a mai urmări jucatorul și se va reîntoarce la starea de repaus, însă nu se va reîntoarce la locul unde a fost inițial poziționat. Inamicul se va deplasa însă spre un jucător, în cazul în care a fost țintit de acesta, chiar dacă jucătorul se află în afara ariei de angajare.

Alergare - când un jucător se afla în raza de angajare în atac (50 unități) , inamicul iese din starea de repaus și se deplasează rapid spre jucator. Starea de animație de se declanșează din *Enemy Animator* este notată cu "*Movement Animation*" și este un *Blend Tree 1-dimensional* ce conține ambele animații pentru starea de repaus și alergare.

Însă *Nav Mesh Agent* nu cunoaște care parte din model este fața sa, ceea ce ar putea cauza inamicul să se îndrepte spre noi alergând cu spatele sau orice altă poziție nenaturală. Pentru a preveni o astfel de situație, am implementat funcția *FaceTarget()* ce va ajusta rotația unui obiect pentru a fi corect orientat, prin utilizarea de *Quaternioni*.

Atac - când jucătorul se afla în raza de atac de 18 unități. Lovitura însă nu va face în toate cazurile contact cu jucătorul, deoarece acesta poate să se ferească. Animația corespunzătoare, notată "*Zombie Attack*", poate fi declanșată și în timp ce inamicul e în alergare și când stă pe loc, atâta timp cât se află la distanța necesară specificată față de jucător. Acest lucru se datorează faptului că animațiile sunt fiecare plasate pe propriul layer, deci se pot suprapune: "*Movement Animation*" se află pe *Base Layer*, iar "*Zombie Attack*" este pe *Attack Layer*, cu un *Avatar Mask* numit "*Enemy Attack Mask"*.

Pentru ca atacul unui inamic să pară realist, jucătorul trebuie să piardă din viață atunci când mâna inamicului îl atinge. Însă codul nu se oprește din rulare până când o animație se termină de rulat, ceea ce înseamnă că trebuie noi să ținem rularea codului în loc. În acest scop folosim funcția *Start Coroutine*.

Normal o funcție se rulează pe parcursul unui singur *frame*, dar *Start Coroutine(function())* apelează funcția, a cărei rulare se poate întinde pe mai multe *frame*-uri, prin folosirea instrucțiunii cheie, *"yield return x*" în function (), unde x reprezintă cât timp vrem să aștepte funcția până să execute codul aflat după instrucțiunea "*yield return"*.

Yield return null face ca funcția să aștepte doar până la *frame*-ul următor. *Yield return new WaitForSeconds (1.5f)* face să se aștepte 1.5 secunde. [30]

Animația *Zombie Attack* durează 2.633 secunde, însă mâna inamicului ajunge în poziția de contact pe la aproximativ 1.1 secunde, moment în care vom scădea din viața jucătorului. Dacă pâna s-a ajuns la momentul respectiv se poate ca jucătorul să se fi dat la o parte în scopul de a evita lovirea, eventual a și ieșit din raza de atac și inamicul nu îl mai poate ajunge, atunci vom declanșa animația de alergare a inamicului pentru a se deplasa spre jucător și vom verifica din nou dacă jucătorul se afla în razele de angajare și atac. Dacă un inamic este năucit, acesta nu e capabil să atace.

Năucire - Când un inamic este lovit aceste pierde din viața exact atâtea puncte cât are ca putere de atac arma jucătorului, iar șocul împușcării îl încetinește pentru puțin timp, de la viteza 50 a componentei *Nav Mesh Agent* la 20. De asemenea, animația de alergare va fi și ea încetinită pentru a nu da senzația că inamicul aleargă "în gol". Pentru a controla viteza unei animații folosim un parametru de tip *Float*, animSpeed pe care îl vom selecta ca *Multiplier* pentru starea "*Enemy Movement"*. Îi dăm valoarea 0.5 în cod pentru ca animația să fie derulată cu viteza redusă la jumătate din viteza normală. La sfârșitul animației, viteza inamicului și viteza de derulare a animației sunt readuse la parametri normali.

Animația "Reaction" se află pe "*Hit Layer*" și are un *Avatar Mask* numit *"Enemy Hit Mask*". Acest lucru permite inamicului să continuie să alerge în timp ce animația "*Reaction*" se derulează pe partea superioară a scheletului. Deoarece animația aceasta este concepută pentru un model 3D staționar, este neapărată nevoie sa fie plasată pe un layer separat dacă există posibilitatea să se declanșeze când inamicul se află în mișcare.

Moartea - când viața inamicului ajunge la zero, îi reducem viteza mai întâi la 20, apoi la 0, iar animația, plasată pe *Base Layer*, se declanșează. Când ajunge la final, viteza sa de derulare este redusă la zero pentru a "îngheța" inamicul în poziția de moarte. Vom dezactiva componentele Script "*Enemy"* și "*EnemyController*", copilul "healthbar" ce e copil al "Canvas" și *Capsule Collider*-ul, cel din urmă ca să putem colecta obiectul ce apare la moarte.

Scorul obținut la doborârea inamicului se adaugă la scorul total și se actualizează pe ecran.

În urma doborârii unui inamic se instanțiază deasupra sa obiecte ce pot fi adunate de către jucător ca o răsplată pentru succesul său. Acestea pot fi găsite și în locuri mai ascunse de pe hartă, dacă jucătorul va căuta să exploreze.

Exista 3 tipuri de obiecte colectabile. Muniție, trusă medicală și upgrade pentru armă. La colectarea de bonusuri se va modifica corespunzător informația de pe ecran pentru fiecare din cele trei. Toate bonusurile au o valoare standard: trusa medicală va crește mereu 10 puncte de viață, muniția va fi mereu în pachete de câte 10, iar upgrade-urile vor avea mereu o creștere de 5 damage.

Obiectele bonus au un efect de "animație" creat prin cod prin rotirea constantă a obiectului concomitent cu ridicarea și coborârea acestuia.

3.4 JUCĂTORUL

Jucătorul este controlat prin tastele WASD pentru deplasare în toate cele 4 direcții, inclusiv pe diagonale, Shift pentru a alerga, Spațiu pentru a sări și Click Stânga pe mouse pentru a trage.

Mișcarea în cele 4 direcții se face printr-un *Blend Tree 2-Dimensional*, în care combinăm 9 animații: 4 pentru direcțiile stânga - dreapta, față - spate, 4 pentru diagonale și una pentru *Idle*. Parametrii *float* folosiți se numesc *VelocityX* și *VelocityZ*. Pentru valorile pozitive din *Blend Tree* ne deplasăm în direcția înainte, iar pentru cele negative, ne deplasăm înapoi.

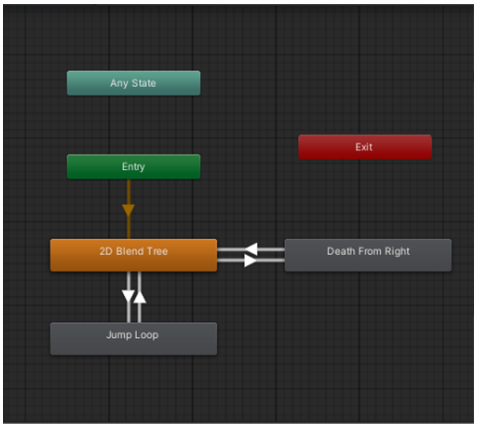

Figura 11

Se folosește *Input.GetAxis ("Vertical") și Input.GetAxis ("Horizontal")* pentru reținerea direcției de deplasare a jucătorului. Funcțiile returnează o valoare între -1 și 1, ce reprezintă mers înainte/dreapta pentru valoare 1 și mers înapoi/stânga pentru valoare -1. În funcție de acești doi parametrii vom determina animațiile corespunzătoare ce trebuie rulate pentru deplasare.

Săritul. Funcția *Input.GeyKeyDown(KeyCode.Space)* returnează *true* când tasta Space este apăsată. Când sărim trebuie să luăm în considerare și gravitația, iar pentru a verifica dacă jucătorul nostru atinge sau nu pământul vom folosi funcția *Physics.CheckSphere(),* ce creează o sferă la poziția jucătorului la nivelul picioarelor. Aceasta returnează *true* dacă există un *collider* ce intra în raza sferei și *false* dacă nu există. Parametru *groundMask* va limita *collider*-ele considerate pentru *CheckSphere()* ca fiind doar cele aflate pe *layer*-ul specificat, "*Ground*" (diferit de *layer*-ele din *Animation Controller*). Acesta este un *layer* creat de noi de îl vom atribui la toate obiectele peste

care dorim ca jucătorul să poată să sară. Această alegere previne și săritul la infinit și săritul în aer, iar animația nici ea nu se va declanșa pentru a nu da impresia că jucătorul sare în gol.

Tragerea cu arma. Animația "*Firing Rifle"* se află pe *Layer-ul "Shoot Layer*" și folosește *Avatar Mask-ul "Player Shoot Mask*" deoarece dorim să putem trage și în timp ce jucătorul este în deplasare, deci dorim două animații în derulare, în același timp, pe același model. Dacă animația nu ar fi fost pusă pe un *layer* separat atunci jucătorul s-ar fi deplasat fără să își miște picioarele, deoarece animația "*Firing Rifle"* este făcută pentru tragerea cu arma de pe loc și ar fi fost derulată pe tot scheletul.

Pentru a verifica dacă jucătorul încearcă să tragă, apelăm funcția *Input.GeyKeyDown(KeyCode.Mouse0)* ce returnează un bool, *Mouse0* reprezentând butonul stâng al mouse-ului.

Funcția *Shoot*() o vom defini ca funcție de tip *Coroutine* deoarece nu dorim ca jucătorul să abuzeze puterea de tragere, așa că vom limita arma să poată să tragă decât o dată la 0.5 secunde.

Pentru a simula o tragere cu arma, vom folosi funcția *Physics.Raycast* ce creează o rază dreaptă de la un punct de start (jucătorul), pe direcția camerei și se oprește în primul *collider* ce a fost lovit. Raza am limitat-o să se extindă până la maximum valoarea variabilei "range", iar informații despre ținta lovită le salvăm în variabila "hit". Dacă aceasta este nulă înseamnă că raza *Raycast* nu a lovit nimic, dar dacă da, atunci obiectele cu o componentă *RigidBody* vor fi împinse de forța impactului dată de glonțul armei. [29]

Cum raza *Raycast* este formată de la poziția camerei, iar camera este poziționată în spatele jucătorului, raza ar putea să lovească jucătorul în loc să lovească obiecte din fața lui cum ar trebui. Pentru a preveni acest lucru vom pune jucătorul pe *layer*-ul special dedicat, "*Ignore Raycast*".

La locul impactului vom instanția un *prefab* de particule din pachetul *Standard Assets* pentru a se observa dacă glonțul a făcut contact sau nu și unde. De asemenea, la fiecare lovitură vom instanția un *prefab TextMeshPro* pentru a afișa exact punctele de viață ce au fost pierdute de inamic.

Năucirea. Când jucătorul se află în această stare nu îi este posibil să tragă cu arma. Acest lucru este implementat pentru a ridica dificultatea. Animația "*Hit Reaction*", este de asemenea plasată pe un *layer* separat, *"Hit Layer",* pentru a permite jucătorului să se îndepărteze de inamici (să poată să ruleze și animația de mișcare în același timp), pentru a avea un mod de a se proteja chiar dacă acesta nu poate să tragă. Pentru a împiedica tragerea cu arma până când animația curentă se termină vom folosi un *StartCoroutine.*

Pentru multe instanțe pentru care folosim funcții *Coroutine* continuăm să ne legăm de variabile și instrucțiuni utilizate în acea funcție *Coroutine* și în afara ei. Dacă funcția *Coroutine* e apelată în funcția *Update* pot apărea probleme cu alte bucăți de cod ce se bazează pe rularea funcției *Coroutine* deoarece acest tip de funcții țin în loc doar rularea codului din interiorul lor, nu și rularea codului din restului programului. Cum codul din funcția *Update* se rulează la fiecare frame, funcția *Coroutine* se va apela și ea la fiecare *frame*, deci vom avea numeroase funcții rulând în paralel, deși noi am dori să se apeleze doar o dată. Pentru a rezolva problema am folosit variabile *bool* numite coroutineCheck. Vom apela o funcție nouă *Coroutine* doar când una identică nu se află deja în rulare.

Moartea - în momentul în care bara de viață a jucătorului ajunge la zero, declanșăm animația. Când aceasta se termină vom debloca cursorul și se va trece la un nou *Scene*, meniul de final de joc.

MAIN CAMERA - plasată ca copil al obiectului Player, pentru ca aceasta să urmărească continuu jucătorul în deplasare. Camera este fixată, întotdeauna privește înainte în relație cu obiectul Player, iar direcția înspre care este îndreptată din mouse este direcția spre care jucătorul se va roti pentru

a se deplasa înainte. Camera am restricționat-o să se ridice in sus până la maxim 40 grade și în jos până la maxim 0 grade.

La moartea unui inamic, unul din 3 bonusuri va apărea, bonus pe care jucătorul poate să îl colecteze. Pentru a verifica dacă obiectul jucător a atins un bonus instanțiat ne folosim de funcția *OnTriggerEnter()* pentru *collider*-ele obiectelor.

3.5 MODEL 3D JUCĂTOR

Modelul 3D al jucătorului a fost creat în Blender, atât mesh-ul, cât și scheletul. Materialul modelului 3D a fost făcut în Adobe Photoshop, iar pentru animații a fost folosită platforma gratuită Mixamo. Pentru a utiliza platforma, se exportă modelul final din Blender sub formatul *.fbx*, se încarcă Modelul 3D pe Mixamo, se alege setul de animații dorit, și se descarcă pachetul, alegând opțiunea “FBX for Unity”. Pe lângă Modelul 3D, în Unity va trebui importat și fișierul de texturi creat de Blender pentru a ține materialele folosite la model, întrucât materialul nu vine împachetat cu modelul. Altfel modelul 3D va fi de culoare uniform gri, fără texturi.

Modelele inamicilor și animațiile acestora au fost descărcate întru cu totul de pe platforma Mixamo, modelele apoi importate în Blender pentru a avea componentele unite într-un singur model și aplicarea de "weight paint" pe mesh, pentru a evita desprinderea bucăților separate de mesh-ul principal în timpul unei animații.

Weight painting în Blender este marcarea pentru fiecare os din schelet a porțiunilor din mesh-ul unui model pe care le dorim ca acel os sa le poate afecta când se află în mișcare. Mesh-ul este o colecție de fețe, muchii și vectori conectate în așa fel încât să desemneze forma modelului 3D.

CAPITOLUL 4 : GREUTĂȚI ÎNTÂMPINATE ȘI ÎMBUNĂTĂȚIRI

4.1 Greutăți întâmpinate

1. La reglarea poziției camerei principale am dorit inițial să folosim o cameră mai dinamică cu poziționare ajustabilă în timpul rulării aplicației, în funcție de poziția jucătorului relativă la obstacolele din jur. Spre exemplu dacă avem un perete în spatele jucătorului, mai aproape decât camera, riscăm ca jucătorul sa poată să vadă prin perete, ceea ce ar exploata jocul. O metodă ușoară de rezolvare a unei astfel de probleme este pachetul *Cinemachine*. Conține seturi de unelte special dedicate pentru controlul unei camere.
   Am optat însă să renunț la aceasta idee deoarece nu am reușit să o reglez corespunzător iar camera de bază din Unity deja oferea funcționalitățile necesar proiectului.
2. La adăugarea de animații pentru jucător am încercat să introduc un *Blend Tree 1- Dimensional* într-un alt *Blend Tree 1- Dimensional*, apoi am apelat în schimb la un *Blend Tree 2D*, mai întâi *Simple Directional* iar la eșuarea și a acestuia, am ales un *Blend Tree 2D Freeform Directional.*
3. Pentru a atașa arma de modelul jucătorului am încercat să folosesc pachetul *Animation Rigging*, însă mesh-ul unuia din brațe nu se lăsa plasat într-o poziție naturală. Am rămas la a poziționa arma prin definirea acesteia ca obiect copil al jucătorului.
4. La construirea meniului am folosit inițial componenta *Text* de bază furnizată de Unity, însă claritatea acesteia este foarte scăzută, alegând în schimb să importez pachetul *TextMeshPro* ce atribuie o calitate mai ridicată textului.
5. Implementarea Load și Save a creat dificultăți la a face inamicii doborâți să rămână așa și după încărcarea ultimei salvări făcute.
6. Când am implementat scorul record și timpul petrecut în sesiunea curentă de joc, implementasem și un timp record, însă nu puteam să-l făcem să afișeze corect.

4.2 Îmbunătățiri

Există multe posibilități de îmbunătățire a unei astfel de aplicații. În primul rând, asemenea multor altor programe, performanța poate mereu fi crescută prin eficientizarea și curățarea codului.

1. Din trăsături potrivite jocului nostru în particular, implementarea opțiunii de deschidere în rețea către alți jucători pentru a putea permite interacțiuni sociale este un aspect dorit datorită faptului că este o caracteristică des întâlnită a foarte multor jocuri moderne, iar permiterea de socializare atrage mai mulți utilizatori ce doresc să încerce aplicația noastră alături de prieteni sau familie.
2. Un sistem de logare ar fi de asemenea foarte benefic aplicației deoarece la momentul actual dacă utilizatorul curent nu are o metodă de protejare a datelor asupra calculatorului, alte persoane ar putea să intervină și să reseteze scorul record al acestuia sau să îi piardă intenționat progresul prin alegerea opțiunii de joc nou.
3. Jocul ar beneficia de inamici mai diverși și de nivele mai multe de pus la dispoziția jucătorului, creând astfel o experiență mai diversă și provocatoare.
4. Multe jocuri la momentul actual au o calitate grafică ridicată și mulți cumpărători caută jocuri cu o fidelitate tot mai mare față de realitate, deci în acest fel și propria aplicație ar beneficia de o creștere a calității în cadrul grafic.

CONCLUZIE

Deși aflat încă într-o variantă demo, din punct de vedere propriu consider că aplicația noastră atinge caractersticile propuse de noi și câteva din cele întâlnite și la alte produse asemănătoare de pe piață, furnizate atât de companiile cu resurse limitate, cât și de programatorii liber-profesioniști ce aduc în industrie jocuri de tip indie. Platforma Unity oferă o simplitate în utilizarea sa, fapt ce o face extrem de atractivă, găsindu-se la îndemâna tutror programatorilor intersați de acest domeniu Limbajul C# este puternic și o alegere ideală comparativ cu alte limbaje oferite de Unity în trecut.

În urma acestei lucrări și al studiului ce a trebuit să se întocmească în paralel cu aceasta, pot spune că am rămas cu cunoștiințe de un nivel crescut, mai multe decât au fost propuse la început, ceea ce înseamnă un success.

Cât despre jocul în sine, o variantă demo are mereu loc de îmbunătățiti, idei ce se vor întreprinde pe viitor, pentru a continua să ne extindem cunoștiințe în acest domeniu.

BIBLIOGRAFIE

Toate accesate cel mai recent în data 02.09.2021

[1] - Unity - Manual: Transforms https://docs.unity3d.com/Manual/class-Transform.html

[2] - Unity - Manual: Colliders https://docs.unity3d.com/Manual/CollidersOverview.html

[3] - Unity - Scripting API: Collider.OnCollisionEnter(Collision)
http://docs.unity3d.com/ScriptReference/Collider.OnCollisionEnter.html

[4] - Unity - Scripting API: Collider.OnTriggerEnter(Collider)
http://docs.unity3d.com/ScriptReference/Collider.OnTriggerEnter.html

[5] - Unity - Manual: Rigidbody https://docs.unity3d.com/Manual/class-Rigidbody.html

[6] - Unity - Scripting API: Character Controller
https://docs.unity3d.com/ScriptReference/CharacterController.html

[7] - Unity - Manual: Character Controller https://docs.unity3d.com/Manual/class-CharacterController.html

[8] - TextMesh Pro User Guide | TextMesh Pro | 1.3.0

https://docs.unity3d.com/Packages/com.unity.textmeshpro@1.3/manual/index.html

[9] - Unity - Manual: Text Mesh Pro
https://docs.unity3d.com/Manual/com.unity.textmeshpro.html

[10] – Unity – Packages: Canvas
https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/UICanvas.html

[11] - Unity - Manual: Inner Workings of the Navigation System

https://docs.unity3d.com/Manual/nav-InnerWorkings.html

[12] - Unity - Manual: Nav Mesh https://docs.unity3d.com/Manual/nav-BuildingNavMesh.html

[13] - Unity - Manual: Animator component

https://docs.unity3d.com/Manual/class-Animator.html

[14] - Unity - Manual: Avatar https://docs.unity3d.com/2017.4/Documentation/Manual/class-Avatar.html

[15] - Unity - Manual: Order of execution for event functions

https://docs.unity3d.com/Manual/ExecutionOrder.html

[16] - Unity - Manual: Event Functions https://docs.unity3d.com/Manual/EventFunctions.html

[17] - Unity - Manual: Avatar Mask window https://docs.unity3d.com/Manual/class-AvatarMask.html

[18] – Unity - Manual: Animation Layers
https://docs.unity3d.com/Manual/AnimationLayers.html

[19] - Unity - Manual: Animation States https://docs.unity3d.com/Manual/class-State.html

[20] - Unity - Manual: Animation transitions https://docs.unity3d.com/Manual/class-Transition.html

[21] - Unity - Manual: Blend Tree https://docs.unity3d.com/Manual/class-BlendTree.html

[22] – Unity - Manual: 1D Blend Tree https://docs.unity3d.com/Manual/BlendTree-1DBlending.html

[23] - Unity - Manual: 2D Blend Tree https://docs.unity3d.com/Manual/BlendTree-2DBlending.html

[24] - Unity - Manual: Scene creation https://docs.unity3d.com/Manual/CreatingScenes.html

[25] - Unity - Scripting API: SceneManager.LoadScene
https://docs.unity3d.com/ScriptReference/SceneManagement.SceneManager.LoadScene.html

[26] - Unity - Manual: Audio Mixer https://docs.unity3d.com/Manual/AudioMixer.html

[27] - Unity - Scripting API: Player Prefs
https://docs.unity3d.com/ScriptReference/PlayerPrefs.html

[28] - Unity - Manual: Script Serialization https://docs.unity3d.com/Manual/script-Serialization.html

[29] - Unity - Scripting API: Physics.Raycast https://docs.unity3d.com/ScriptReference/Physics.Raycast.html

[30] - Unity - Manual: Coroutines https://docs.unity3d.com/Manual/Coroutines.html

Asset Store - https://assetstore.unity.com/top-assets/top-paid?aid=1011lfRDN&gclid=Cj0KCQjw7MGJBhD-ARIsAMZ0eeuVwV07pWp2a2X5-vzR4kv5KJC1m-Y3_ITpTi9gDAC2W6Y_b8KGJtYaAgpmEALw_wcB&utm_source=aff

Mixamo - https://www.mixamo.com/#/