



**Universitatea  
Transilvania  
din Brașov**  
**FACULTATEA DE MATEMATICĂ  
ȘI INFORMATICĂ**

# LUCRARE DE LICENȚĂ

---

## Serenity Garden TD

Conducător Științific:  
Conferențiar Universitar  
Deaconu Adrian

Absolvent:  
Opria Ion-Bogdan

Brașov, 2021

# Cuprins

<b>1 Introducere</b>	<b>5</b>
1.1 Descrierea proiectului . . . . .	5
1.2 Introducere în dezvoltarea jocurilor video . . . . .	5
1.2.1 Scurtă istorie . . . . .	6
1.2.2 Industria curentă . . . . .	8
<b>2 Introducere în Unity</b>	<b>9</b>
2.1 Motivul alegerii programului Unity pentru dezvoltarea proiectului . . . . .	9
2.2 Prezentare generală . . . . .	9
2.2.1 GameObject și Mesh . . . . .	10
2.2.2 Collider . . . . .	11
2.2.3 Transform și Rigidbody . . . . .	11
2.2.4 Prefab . . . . .	12
2.2.5 Raycast . . . . .	12
2.3 Elemente specifice folosite . . . . .	12
2.3.1 Editoare personalizate . . . . .	13
2.3.2 Shader Graph . . . . .	13
2.3.3 VFX Graph . . . . .	13
2.3.4 Photon Engine . . . . .	13
<b>3 Planificarea Proiectului</b>	<b>14</b>
3.1 Faza de Proiectare . . . . .	14
3.2 Tematica Jocului . . . . .	14
3.3 Caracterele Jocului . . . . .	15
3.3.1 Inamicii . . . . .	15
3.3.2 Turnuri defensive . . . . .	17
3.4 Sistemele Jocului . . . . .	22
3.4.1 Harta de joc . . . . .	22
3.4.2 Sistemul Turnurilor . . . . .	23
3.4.3 Comandantul . . . . .	24
3.4.4 Valul de inamici . . . . .	24
3.4.5 Nivelele jocului . . . . .	24
3.4.6 Lock-on . . . . .	24
3.4.7 Magazinul de îmbunătățiri permanente . . . . .	25
3.4.8 Sistemul co-op . . . . .	25
3.4.9 Atacurile monstrului de foc . . . . .	26
3.5 Analiza proiectului din punctul de vedere al consumatorilor . .	26
3.5.1 Dificultatea jocului . . . . .	26

3.5.2	Elemente de dependență . . . . .	27
3.5.3	Grupele de vârstă vizate . . . . .	27
<b>4</b>	<b>Implementarea proiectului</b>	<b>27</b>
4.1	Scena de luptă . . . . .	27
4.1.1	Inițializarea scripturilor . . . . .	27
4.1.2	Procesarea comenzilor venite de la utilizator . . . . .	29
4.1.3	Harta de joc . . . . .	30
4.1.4	Sistemul de navigare . . . . .	35
4.1.5	Interfețe folosite . . . . .	36
4.1.6	Ierarhia inamicilor . . . . .	38
4.1.7	Ierarhia turnurilor defensive . . . . .	40
4.1.8	Comandantul . . . . .	42
4.1.9	Punerea pe pauză a jocului . . . . .	43
4.1.10	Valul de inamici . . . . .	46
4.1.11	Stagiile jocului . . . . .	47
4.1.12	Controlul jocului . . . . .	47
4.2	Selectarea nivelerelor . . . . .	49
4.2.1	Structura meniului . . . . .	49
4.2.2	Generarea dinamică a meniului . . . . .	50
4.2.3	Transmisia datelor între scene . . . . .	51
4.3	Magazinul de îmbunătățiri permanente . . . . .	53
4.3.1	Structura meniului . . . . .	53
4.3.2	Îmbunătățire permanentă . . . . .	54
4.3.3	Generarea dinamică a meniului . . . . .	55
4.4	Salvarea datelor . . . . .	56
4.4.1	Datele care trebuie salvate . . . . .	56
4.4.2	Salvarea automată . . . . .	56
4.4.3	Locația fișierelor salvate pentru dispozitive diferite . . . . .	58
4.5	Sistemul co-op . . . . .	59
4.5.1	Modul de lucru cu Photon Engine . . . . .	59
4.5.2	Înregistrare și camera de așteptare . . . . .	62
4.5.3	Sincronizarea datelor . . . . .	64
4.5.4	Monstrul de foc . . . . .	66
4.6	Metode de îmbunătățire a proiectului . . . . .	68
4.6.1	Artă îmbunătățită . . . . .	68
4.6.2	O poveste pentru joc . . . . .	69
4.6.3	Sunetele pentru joc . . . . .	69
4.6.4	Monștrii mulți pentru modul co-op . . . . .	70
4.6.5	Balansarea dificultății . . . . .	70

<b>5</b>	<b>Concluzii</b>	<b>71</b>
<b>6</b>	<b>Bibliografie</b>	<b>72</b>

# 1 Introducere

## 1.1 Descrierea proiectului

Jocurile video au început să acapareze din ce în ce mai mult viețile noastre, datorită ușurinței de accesare și a experienței pe care o oferă. Odată cu evoluția hardware-ului, jocurile video au devenit din ce în ce mai realiste și prin urmare, oferă experiențe care nu poate fi găsite în niciun alt mediu existent, deoarece, în nici un alt mediu nu putem controla și trai experiențele unor personaje atât de detaliat. Industria jocurilor video este o industrie în continuă creștere și prin urmare este o ramură care merită explorată din perspectiva unui programator și nu nu mai.

”Games are great to work on because they are as much about art as they are science.” [1]

Din aceste motive, am ales drept proiect pentru licență, crearea unui joc video. Jocul se numește ”Serenity Garden TD” și este un tower defense 3D. Experiența jocului are loc în câteva nivele special definite, în care utilizatorul trebuie să își protejeze baza de operații. Există multe tipuri diferite de inamici care vor să distrugă baza jucătorului, iar acesta, pentru a o proteja, trebuie să construiască un sistem defensiv. Jucătorul poate să construiască turnuri defensive, care au diferite efecte asupra inamicilor. Inamicii atacă sistemul defensiv, sau în cazul în care lipsește, atacă direct baza jucătorului. Dacă reușesc să distrugă baza, jucătorului, acesta pierde nivelul curent.

Jucătorul are locuri predefinite în care poate să își construiască turnurile, locuri specificate de hexagoane colorate. Odată plasat un turn, aceasta nu poate fi mutat, dar jucătorul are și un caracter invulnerabil, pe care îl poate muta oriunde dorește pe hartă. Acest caracter va ataca inamicii automat, atunci când nu se află în mișcare, și poate intra în turnuri pentru a le crește puterea/eficiența.

Modul care va diferenția acest joc de celealte jocuri pe acest stil este modul co-op. Doi jucători se vor putea conecta prin rețea și vor juca un nivel dificil care necesită cooperare și o planificare bună între aceștia.

## 1.2 Introducere în dezvoltarea jocurilor video

Dezvoltarea jocurilor video cuprinde o serie mare de domenii, care lucrează împreună, pentru a crea un produs cât mai bun. O parte din aceste

domenii sunt: programare, artă, sunete, management, etc. Datorită ariei largi de domenii care lucrează împreună pentru a produce un joc reușit, comunicarea și organizarea reprezintă un aspect foarte important pentru jocurile majore. Dacă această comunicare nu este stabilită încă de la începutul proiectului, jocul poate fi prelungit, anulat complet sau nu va ieși pe măsură așteptărilor.

Specific pentru această industrie au fost create și o serie de aplicații / tehnologii ajutătoare, pentru a ușura și grăbi procesul de dezvoltare. În continuare vom acoperi doar tehnologiile utile programatorilor din industrie.

OpenGL și DirectX sunt librării de randare 3D care ne permit să creăm scene complexe pentru jocurile noastre. O mare parte din industrie are la baza aceste tehnologii sau tehnologii similare cu acestea. OpenGL este o librărie care poate să ruleze pe majoritatea sistemelor de operare și majoritatea dispozitivelor, pe când DirectX a fost dezvoltat numai pentru sistemele de operare Windows. OpenGL este "open-source", motiv pentru care este mai bine documentat și mai ușor de învățat de către programatorii noi.

Dezvoltarea în aceste librării este destul de complexă și ocupă mult timp, motiv pentru care a apărut conceputul de motoare pentru dezvoltare (game engines). Acestea sunt aplicații specifice, care la bază se folosesc de OpenGL și/sau DirectX, dar în intermediul cărora sunt definite multe funcționalități utile care grăbesc procesul de dezvoltare. Cele mai populare motoare de dezvoltare în clipa curentă sunt: Unreal Engine, Unity, Godot, Game Maker Studio.

Pentru un programator începător, care dorește să între în industrie, este recomandat să învețe un motor de dezvoltare, pentru a putea construi o serie de jocuri și să acapareze cât mai multă experiență într-un timp cât mai scurt. Pe urmă, dacă dorește să se dezvolte și să ajungă la un cu totul alt nivel, este recomandat să învețe OpenGL/DirectX, deoarece acestea prezintă foarte multe concepte utile și deschid porțile către aplicații/interacțiuni cu mult mai complexe.

### 1.2.1 Scurtă istorie

Primul joc video dezvoltat pentru un monitor digital a fost "OXO", creat în anul 1952. În urmă acestuia au mai apărut câteva tentative de jocuri, dar dezvoltarea comercială a jocurilor video a început în anul 1970, când au apărut primele jocuri de tip arcadă. În anul 1972, Atari a publicat prima

versiune a jocului Pong, care a fost un succes imens și care a pus bazele acestei industriei.

A doua generație de jocuri o reprezintă apariția primelor console. După succesul jocurilor de tipul arcadă, au început să apară jocuri care rulau pe un micro-procesor, fapt care a dus la apariția unor console care puteau rula jocuri multiple pe același dispozitiv. Limitarea cea mai majoră a jocurilor de tipul arcadă era faptul că dispozitivul conținea un singur joc prestabilit și nu putea fi modificat cu usurință. Pentru consolele din această perioadă, jocurile erau salvate pe casete speciale, care puteau fi introduse în consolă pentru începerea jocului respectiv.

În 1978 a apărut prima versiune a jocului Space Invaders, care de asemenea, a fost un succes masiv. În același an au apărut și calculatoarele pe piață, fapt care a permis programatorilor individuali să își creze propriile lor jocuri. Acest lucru a pornit o perioadă în care programatorii dezvoltau jocurile de unii singuri și le publicau pe piață prin intermediul unor editori specifici.

Odată cu trecerea timpului și cu îmbunătățirea mediilor de dezvoltare, au început să se formeze echipe pentru crearea acestor jocuri. Organizarea pe echipe a crescut calitatea jocurilor, deoarece persoane specializate puteau să lucreze la anumite aspecte ale jocurilor, fapt care a îmbunătățit exponential calitatea acestora. Aceste echipe, în cele din urmă, au dus la crearea unor companii dedicate dezvoltării de jocuri, fapt care a pus bazele industriei curente.

Câteva titluri importante au fost:

- **Pacman** a fost o mașină de tipul arcadă, apărut în anul 1980 și a devenit o serie foarte populară, care, până și în zilele noastre, adaugă conținut nou prin desene animate/jocuri specifice seriei.
- **Final Fantasy** este jocul care a salvat compania Square Enix, una din cele mai mari companii din zilele noastre. Compania a dezvoltat o serie de jocuri care nu au avut succes și era pe punctul să intre în faliment. În această perioadă, dezvoltatorii au decis să încerce pentru ultima dată să facă un joc de succes, și au numit acest joc Final Fantasy. Jocul a apărut în anul 1987 și a creat conceptul de jocuri RPG (Role-Playing Game).
- **Wolfenstein3D** a fost primul joc care a atins conceptul de jocuri 3D. Mașinile de dezvoltare din această perioadă nu puteau permite obiecte

3D, motiv pentru care, toate jocurile erau 2D, inclusiv Wolfenstein3D. Hărțile acestui joc erau 2D, dar folosindu-se de Raytracing și alte concepte ingenioase, au reușit să păcălească ecranele să deseneze scene 3D.

- **Tomb Rider** a apărut în anul 2001 și este primul joc care a avut drept caracter principal un personaj feminin. În acea perioadă, industria era foarte părtinitoare, în sensul în care majoritatea credeau că jocurile video sunt concepute doar pentru băieți. Acest joc este primul care combate aceste ideologii, iar datorită popularității a reușit să aducă pe piață o serie de jocuri foarte populare: Life is Strange, The Last of Us, Remember Me, etc.

### 1.2.2 Industria curentă

În zilele noastre, industria jocurilor video a devenit o industrie imensă, care acaparează o mare parte din viețile noastre. În anul 2020, industria valora 152 miliarde de dolari, și este în continuă creștere.

Industria are 2 arii largi în care se încadrează majoritatea dezvoltatorilor:

- Aria Indie este o industrie pentru dezvoltatori mai mici de jocuri. Numărul de dezvoltatori este de obicei mic și numărul de resurse este limitat, motiv pentru care, aceștia caută soluții ingenioase, care produc jocuri de calitate în ciuda forței de muncă redusă. Această arie acaparează cele mai multe jocuri de pe piață, deoarece, oricine are talentele necesare poate intra în această arie, dacă își dorește acest lucru. O serie de companii foarte populare din această arie sunt: Moon Studios, Team Cherry, Blitworks, etc.
- Aria AAA este industria în care se încadrează firmele majore de dezvoltare de jocuri. Acestea au un număr imens de angajați și au la dispoziție foarte multe resurse pentru dezvoltare. De obicei, își organizează angajații în echipe pe diverse proiecte, o parte din proiecte fiind jocurile AAA. Un joc AAA este un joc la care lucrează cel puțin 100 de persoane, a cărui durată de dezvoltare se întinde pe o perioadă de câțiva ani, și care în momentul publicării pe piață, poate devenii unul din cele mai bune jocuri din acea perioadă, datorită devotamentului și resurselor folosite. Exemple de astfel de jocuri sunt: God of War, Red Dead Redemption, Dark Souls, The Last of Us, Persona 5, Final Fantasy, etc. Câteva firme majore din această arie sunt: Santa Monica

Studios, Square Enix, Bethesda, Blizzard, Riot, ID Software, Platinum Games, Bandai Namco, etc.

Industria curentă încă este în continuă dezvoltare, apar de la an la an tehnologii complet noi precum: realitate virtuală, realitate augmentată, senzori de simțuri pentru realitatea virtuală, inteligență artificială în jocuri, RTX, concepe de optimizari imense pentru jocuri, etc.

## 2 Introducere în Unity

### 2.1 Motivul alegerii programului Unity pentru dezvoltarea proiectului

Pentru acest proiect am ales să folosesc Unity, în locul altor motoare de dezvoltare sau librării speciale (OpenGL, DirectX), dintr-o serie de motive:

1. Scopul meu pentru acest proiect a fost să realizez un joc complet cu foarte multe interacțiuni și scenarii definite. Astfel, am avut nevoie de o viteză de dezvoltare cât mai mare, ceea ce este valabil în cazul Unity-ului.
2. Un alt motiv al acestei alegeri este experiența mea personală în Unity. Lucrez în acest motor de dezvoltare sub formă de hobby de 6 ani, timp în care am învățat o mare parte din funcționalitățile pe care le pune la dispoziție.
3. Pe lângă asta, am avut nevoie de câteva elemente specifice, care îmbunătățesc calitatea jocului exponențial. Aceste elemente vor fi discutate în capitolul [2.3](#).

### 2.2 Prezentare generală

Unity este un motor de dezvoltare care a apărut în anul 2005 și care este în continuă dezvoltare până și în zilele noastre. Momentan a ajuns la ediția 2020.3 LTS și a reușit să schimbe complet industria jocurilor datorită modului ușor de lucru. În decursul timpului a adăugat atât de multe funcționalități încât dezvoltarea de jocuri a devenit mai simplă că niciodată (s-a ajuns în punctul în care pot fi create jocuri simple fără să fie nevoie de programare).

Unity conține o multitudine de moduri de lucru, diverse librării grafice, suport pentru o serie largă de dispozitive, pe care putem să ne exportăm

jocurile și o comunitate foarte ajutătoare. Acesta, față de Unreal Engine, este considerat motorul de dezvoltare al industriei indie, deoarece grafica rezultată nu este la fel de realistă ca în Unreal Engine, dar libertatea de dezvoltare este cu mult mai mare.

Câteva jocuri populare dezvoltate până acum în Unity sunt: Osiris New Dawn, Hollow Knight, Cuphead, Pokemon GO, Wastelands 3, League of Legends Wild Rift, Ori and the Will of the Wisps, etc.

În continuare vom trece peste câteva concepte importante despre dezvoltarea în Unity, pentru a ușura înțelegerea implementării proiectului, discutată în capitolul 4.

### 2.2.1 GameObject și Mesh

GameObject este clasa de bază a tuturor obiectelor din scenă. Acesta acționează drept un container, în sensul în care, pe un GameObject putem să adăugăm orice script dorim. Printre proprietățile lui se numără:

- **Transform** reprezintă clasa spațială a tuturor obiectelor din scenă. Acesta definește poziția, rotația și scala la care se află un obiect în scenă.
- **Tag** este o proprietate pe care o putem defini pentru anumite obiecte. Este utilă pentru a face diferențierea între obiecte în anumite situații.
- **Layer** este o proprietate similară cu tag-ul, singura diferență fiind abilitatea de a ignora anumite interacțiuni între layere. De exemplu putem să ignorăm coliziunile între 2 layere sau să nu afișăm deloc pe ecran obiectele aflate pe un anumit layer.
- **Numele** obiectului care este vizibil în scenă. Este important să denumim obiectele corespunzător pentru a le găsi mai ușor în scenă.

Un GameObject nu trebuie neapărat să aibă o reprezentare grafică. Poate să fie doar un obiect gol care deține mai multe scripturi și/sau a cărui informații sunt folosite de alte scripturi. În cazul în care dorim să adăugăm un obiect 3D în scenă, intervine componenta Mesh.

Mesh reprezintă un obiect 3D, cu toate informațiile necesare pentru renderarea acestuia pe ecran: vertexurile, triunghiurile formate de acestea, coordinatele texturilor, etc. Pentru a randa un obiect pe ecran, este nevoie de 2 componente:

1. **MeshFilter** poate să citească datele din Mesh și să le organizeze în aşa fel încât să poată fi folosite de MeshRenderer.
2. **MeshRenderer** este componenta care se ocupă propriu-zis de randarea obiectului pe ecran. Aceasta definește materialele pe care să le folosească și interacțiunea obiectului cu mediul (dacă primește sau generează umbre, modul de randare, etc.).

### 2.2.2 Collider

Este componenta care definește modul de interacțiune între obiectele scenei. Există mai multe tipuri de collidere: Box Collider, Sphere Collider, Capsule Collider, Mesh Collider, etc. Aceste collidere pot să fie collidere normale sau declanșatoare. În cazul în care folosim sistemul de fizică definit de Unity (discutat în capitolul următor) iar 2 obiecte cu collidere normale s-au ciocnit unul de celălat, ele se vor bloca exact ca în lumea reală. În cazul în care sunt definite drept declanșatoare, acestea vor trece unul prin altul, dar vor apela metoda `OnTriggerEnter()` din scripturile de pe aceste obiecte.

Colliderele normale sunt folosite pentru a nu permite obiectelor/caracterelor să treacă prin alte obiecte (inclusiv prin podea), iar cele declanșatoare sunt utile pentru a activa anumite evenimente (de exemplu: un caracter se așează pe o platformă, moment în care platforma începe să se mișe în sus și să acționeze drept un lift).

În principiu, este de dorit să existe un singur collider pe obiect. În cazul în care dorim mai multe collidere, putem crea copii ai obiectului curent și să așezăm colliderele pe acești copii. Colliderele nu funcționează dacă niciunul din obiectele care fac coliziune nu deține componenta `RigidBody`.

### 2.2.3 Transform și Rigidbody

Precum a fost discutat și la începutul capitolului, componenta `Transform` reține locația, rotația și scara la care se află obiectul în lume. Această componentă deține și o serie de metode utile pentru a modifica aceste proprietăți (să mutăm obiectul într-o anumită direcție, să îl rotim pe o anumită axă, sau în jurul altui punct din scenă, etc.). Componenta deține 2 tipuri de coordinate spațiale: cele raportate la lume și cele raportate la obiect. Coordonatele spațiale raportate la obiect reprezintă unde se află obiectul respectiv, indiferent dacă este copilul altui obiect sau nu. Coordonatele spațiale raportate la obiect sunt coordonatele pe care le ocupă obiectul în comparație cu părintele

acestuia.

Pentru a ușura înțelegerea acestui concept o să urmărim un exemplu al modului de lucru. Să presupunem că avem un obiect A, care este poziționat la coordonatele  $(10, 0, 0)$  în lume. Acest obiect are un copil B care se află la coordonatele  $(-5, 2, 0)$  raportate la părintele lui. Obiectul B are coordonatele  $(5, 2, 0)$  când este raportat la lume.

Rigidbody este componenta care definește sistemul de fizică din Unity. Acesta definește concepte precum: velocitate, gravitație, velocitate unghiulară, masa obiectelor, forța de frecare, etc. Aceste proprietăți pot fi modificate cu ușurință sau chiar dezactivate în funcție de necesitate.

#### 2.2.4 Prefab

Prefab ușurează modul de lucru cu obiecte în mai multe scene sau chiar și într-o singură scenă. Prefab este un fel de şablon pe care îl putem defini pentru obiectele din scenă. După ce am definit acest şablon, este foarte ușor să creăm mai multe obiecte de acest tip, în mai multe scene de joc. Avantajul apare în situația în care dorim să modificăm toate obiectele de acest tip din scenele jocului. Nu trebuie să le modificăm individual, ci putem modifica direct şablonul, astfel actualizându-se toate obiectele de acest tip din toate scenele.

#### 2.2.5 Raycast

Raycast reprezinta un mod foarte ușor de a detecta anumite lucruri în cod. Un raycast conține o origine, o direcție în care se îndreaptă și o distanță maximă admisibilă. Folosindu-ne de sistemul de fizică din Unity și de Raycast, putem detecta, dacă pe o anumită direcție, lovim un anumit obiect din scenă. Aceasta este util în cazul în care dorim să creăm un sistem în care oprim coliziunile anumitor obiecte, înainte ca acestea să se ciocnească, sau să găsim obiectele dintr-o anumită direcție (ex: când dăm click pe ecran, să găsim obiectul pe care l-am selectat). Pentru a funcționa, este necesar ca obiectele din scenă să aibă definite collideri. Rigidbody nu este necesar pentru aceste interacțiuni.

### 2.3 Elemente specifice folosite

### **2.3.1 Editoare personalizate**

Pentru a ușura modul de lucru pentru anumite sisteme, a trebuit să creez editoare personalizate pentru o serie de scripturi. Prin intermediul acestor editoare personalizate, informația afișată în editor poate fi organizată într-un mod diferit față de cel de baza care este definit de Unity. Pe lângă acest lucru, pot fi rulate anumite funcționalități ale scripturilor direct din editor, fapt ce normal ar fi posibil doar la rularea aplicației.

Pentru crearea unui astfel de editor personalizat, trebuie creat un nou script care se așează într-un folder numit Editor. În caz contrar, aplicația nu va putea construi un executabil pentru joc. Acest script trebuie să moștenească clasa Editor și să definească deasupra clasei, tipul de clasa pentru care creăm un editor personalizat. Pe urmă, în metoda OnInspectorGUI() adăugăm funcționalitățile dorite.

### **2.3.2 Shader Graph**

Shader Graph este o componentă din Unity care ne permite să creăm shadere specializate dintr-o interfață grafică. Aceasta funcționează pe un sistem de noduri similar cu multe alte aplicații care se ocupă cu definirea materialelor 3D (Blender, Substance Alchemist, etc.). Interfața grafică este ușor de utilizat, grăbește foarte mult timpul de dezvoltare a acestor shadere specializate, și poate defini proprietăți care să varieze aspectul final al shaderului. În urma definirii acestor shadere, putem crea cu ușurință materiale care să le folosească.

### **2.3.3 VFX Graph**

Este o componentă nouă în Unity, care îmbunătățește sistemul de particule. VFX graph poate să genereze foarte multe tipuri de efecte speciale, pune la dispoziție multe proprietăți care pot fi modificate pentru a atinge aspectul dorit, și este foarte bine optimizat, până la punctul în care poate suporta milioane de particule în același timp.

### **2.3.4 Photon Engine**

La acest proiect am decis să adaug și o componentă co-op, care va fi discutată în detaliu în capituloare ce urmează. În acest scop, am avut nevoie de un server la care să pot conecta aplicația. Photon Engine este un sistem dezvoltat de cei de la Exit Games și pune la dispoziție un server gratuit care

permite până la 20 de utilizatori conectați în același timp la aplicație.

Acesta oferă diverse funcționalități care ajută stabilirea conexiunii cu serverul și între clienți, comunicarea între clienți și sincronizarea datelor. Mai multe detalii vor fi discutate în capitolul 4.5.

## 3 Planificarea Proiectului

### 3.1 Faza de Proiectare

”Almost anything that you can be good at can become a useful skill for a game designer.” [2]

Din proiecte precedente, am realizat că o planificare bună a unui proiect încă de la început, poate îmbunătății calitatea proiectului exponential, asigură ușurința de adăugare a unor elemente noi (fără să fie necesară rescrierea/reorganizarea proiectului) și reduce frustrarea programatorilor care lucrează la proiect.

Din acest motiv, încă de la începutul proiectului am încercat să planific cât mai detaliat conținutul acestuia și modul de implementare.

În prima fază, am scris un document de proiectare, care conținea descrierea și elementele jocului, privite din multe perspective.

Introducerea proiectului a fost prezentată la începutul acestei lucrări și nu va fi reluată aici.

### 3.2 Tematica Jocului

Jocul are loc în viitorul îndepărtat. Planeta este supra-populată, iar nivelul de poluare a ajuns la un nivel critic. Umanitatea a trecut de mult de punctul în care mai putea salva această planetă. Întreaga planetă va deceda în câteva decenii, iar dacă oamenii nu își găsesc o altă locuință între timp, vor muri și ei împreună cu aceasta. Astfel, a fost formată o armată specială cu scopul de a explora alte planete și de a găsi una care poate suporta rasa umană.

Universul este totuși foarte periculos. Unele planete au viață extraterestră foarte agresivă, iar altele au fost distruse de experimente cibernetice

eșuate. Jucătorul este comandantul suprem al acestei armate care are drept scop protejarea oamenilor de știință, care vor analiza aceste planete.

Prima fază a proiectului va conține o singură planetă care poate fi populată, dar pe care există o serie de inamici periculoși.

Planeta mama trimitre resurse din când în când, dar numai un număr limitat de resurse pot fi trimise deodată. Datorită distanței imense, livrarea resurselor durează mult timp, motiv pentru care nu putem depinde de ele în timpul unei lupte.

### 3.3 Caracterele Jocului

**Comandantul** este protagonistul jocului. El este caracterul a cărui perspectivă o vom trăi pe tot parcursul jocului. În trecut, a fost cobai pentru un experiment menit să creeze super-soldați, iar în urma acestui experiment a primit o putere specială. Poate să își ascundă complet prezența față de alți oameni sau alte creaturi. Din acest motiv, el este caracterul invulnerabil pe care îl putem mișca pe hartă în timpul luptei.

#### 3.3.1 Inamicii

- **Inamicul de luptă apropiată** (fig: 1) este un tip de inamic care poate ataca turnurile doar când stă fix în fața lor. Are viață mai multă decât inamicii de luptă de la distanță.
- **Inamicul de luptă de la distanță** (fig: 2) este un tip de inamic care poate ataca turnurile de la distanță. Are viață puțină, dar pot ataca des.
- **Inamicul zburător** (fig: 3) este un inamic care poate fi lovit doar de un număr limitat de turnuri.
- **Inamicul bombardier** (fig: 4) este un tip de inamic care ignoră turnurile întâlnite în cale. Acesta se mișcă pe cel mai scurt drum către baza jucătorului, iar dacă ajunge deasupra acesteia, va lansa bombe, care îi vor scădea drastic viață, după care va pleca de pe hartă. Este un tip de inamic zburător care poate fi lovit doar de un număr limitat de turnuri.

Acești inamici, cu excepția bombardierului, se vor mișca pe hartă, calculând cel mai scurt drum către baza jucătorului. Dacă aceștia întâlnesc în

cale un turn, se vor opri și vor ataca acel turn. Când turnul este distrus, aceștia își vor relua drumul.

Pentru fiecare inamic vor fi 3 tipuri de variații, fiecare mai puternic decât cel precedent.



Figure 1: Inamicul de luptă apropiată



Figure 2: Inamicul de luptă de la distanță

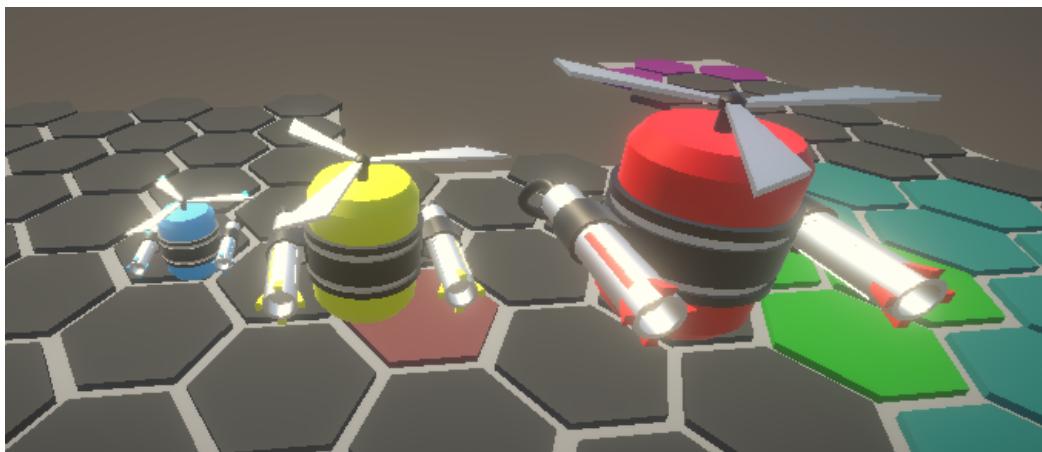


Figure 3: Inamicul zburător

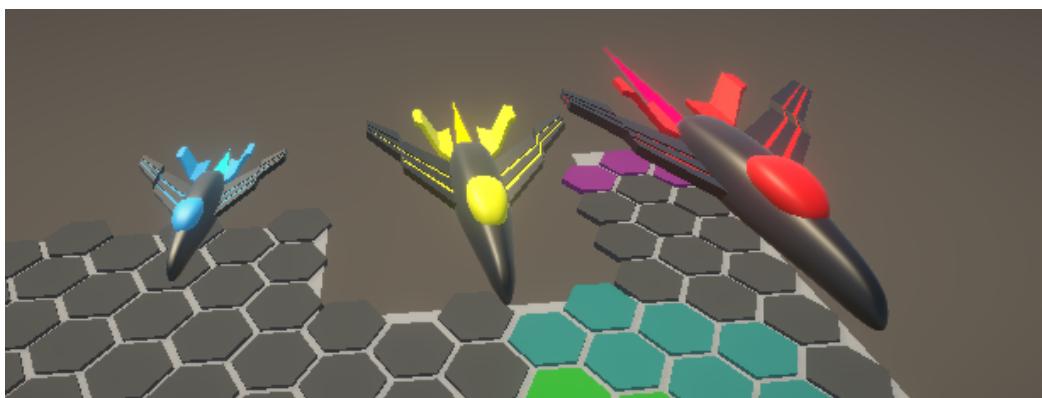


Figure 4: Inamicul bombardier

### 3.3.2 Turnuri defensive

În continuare voi prezenta o descriere scurtă pentru fiecare turn din joc. Toate proprietățile despre care urmează să vorbesc sunt în comparație cu celealte turnuri defensive.

- **Serenity** (fig: 5) este baza jucătorului. Are viață foarte multă, iar dacă este distrusă jucătorul pierde nivelul curent. Aceasta poate ataca inamicii care se apropiie de ea. Are rază de atac medie, putere de atac medie, poate ataca orice tip de inamic, iar rata de reîncărcare între atacuri este mică (durează mult timp între atacuri)

- **Mitralieră automată** (fig: 6) Are viață puțină, poate ataca orice tip de inamic, are rază medie, putere de atac mică, iar rata de atac este mare (atacă foarte des)
- **Gardul electric** (fig: 7) este turnul de protecție. Are viață foarte multă, poate ataca doar inamicii de luptă apropiată, puterea de atac este medie, iar rata de reîncărcare este mică. Scopul acestui turn este să stea în calea inamicilor pentru a fi atacat de aceștia, permitând celorlalte turnuri să atace inamicii blocați.
- **Vulkan** (fig: 8) Este turnul special construit pentru a lupta împotriva inamicilor zburători. Are viață puțină, rază de atac mare și putere de atac mare, dar poate ataca doar inamicii zburători.
- **Aruncătorul de flăcări** (fig: 9). Când inamicii intră în raza acestui turn, el va împrăștia flăcări către inamici. Toți inamicii care intră în aceste flăcări primesc daune pentru fiecare clipă în care stau în flăcări. Are viață medie, atacă instant, iar puterea de atac este mică dar distribuită pentru fiecare clipă în care inamicii stau în flăcările acestuia.
- **Laser** (fig: 10). Este un turn similar cu aruncătorul de flăcări în sensul în care atacă în fiecare clipă în care un inamic este în raza acestuia. Atacă cu un laser puternic care scade treptat viața inamicilor. Are viață medie, atacă instant și puterea de atac este mică.
- **Excavator** (fig: 11) este un tip special de turn, deoarece nu poate ataca inamicii. Acesta adună resurse în timpul luptei, resurse care pot fi convertite în bani de joc. Are viață multă, și pentru că nu poate ataca, trebuie să fie aparat de inamici.

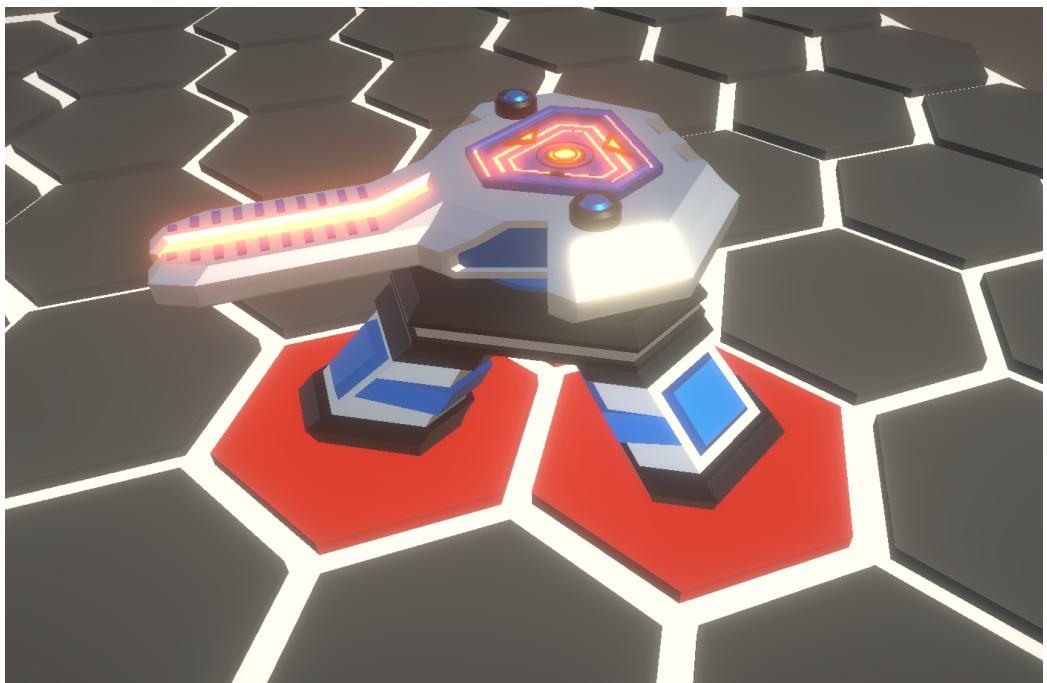


Figure 5: Serenity



Figure 6: Mitraliera automata

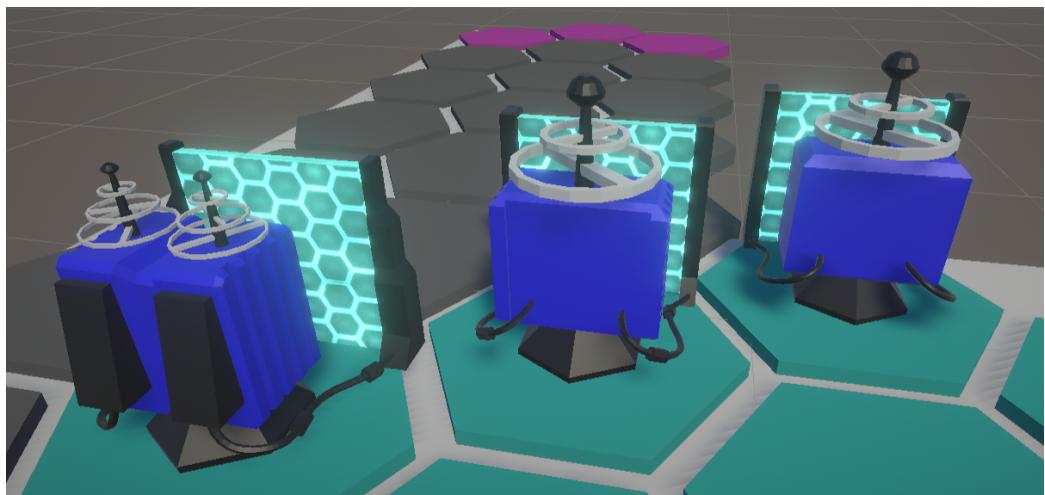


Figure 7: Gardul electric



Figure 8: Vulkan

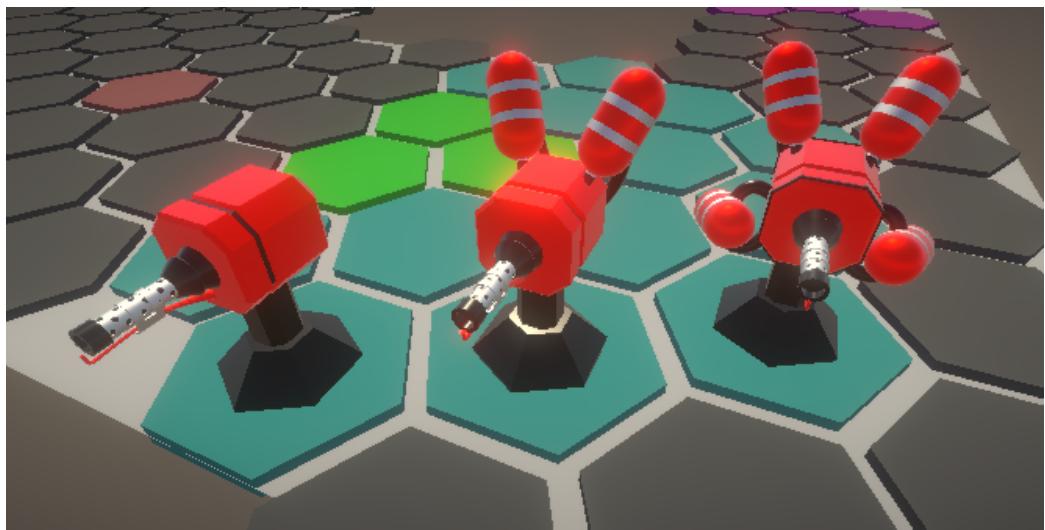


Figure 9: Aruncatorul de flacari

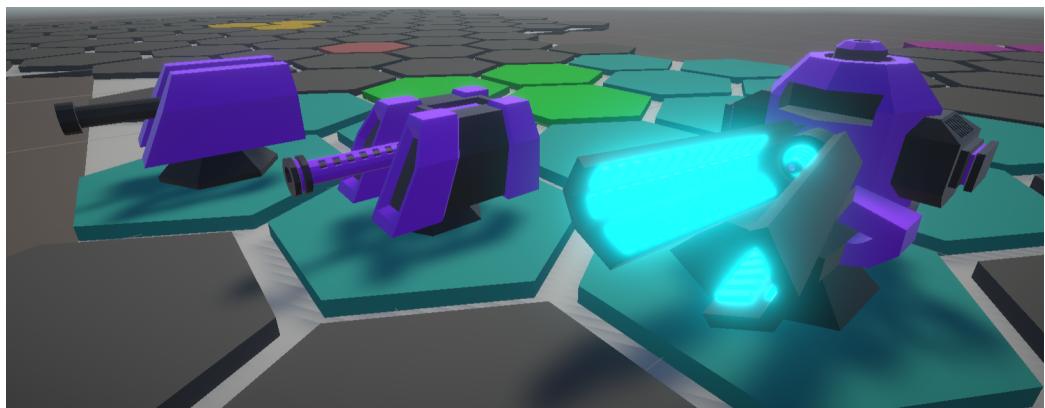


Figure 10: Laser

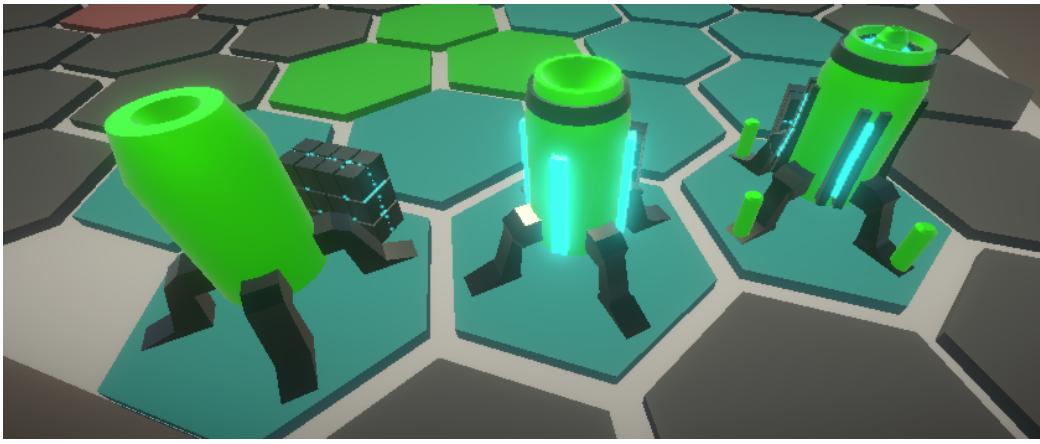


Figure 11: Excavator

## 3.4 Sistemele Jocului

### 3.4.1 Harta de joc

Pe harta jocului vor fi generate automat obiecte hexagonale. Aceste obiecte pot fi de mai multe tipuri:

- Hexagoane goale (cele gri din (fig: 12)).
- Hexagoane pe care putem construi turnuri de atac. (cele albastre din fig: 12)
- Hexagoane pe care putem construi turnul de extragere de resurse. (cele verzi din fig: 12)
- Hexagonul pe care va începe și se va afla comandanțul. (cel roz din fig: 12)
- Hexagoanele bazei jucătorului. Baza va ocupa 3 hexagoane de acest tip. (cele galbene din fig: 12)
- Hexagonul ocupat. Fiecare hexagon care este ocupat de o anumită structură devine un astfel de hexagon. (are culoarea roșie)

Inamicii vor calcula drumul pe care merg în funcție de toate aceste tipuri de hexagoane. Din acest motiv este nevoie și de cel gol.

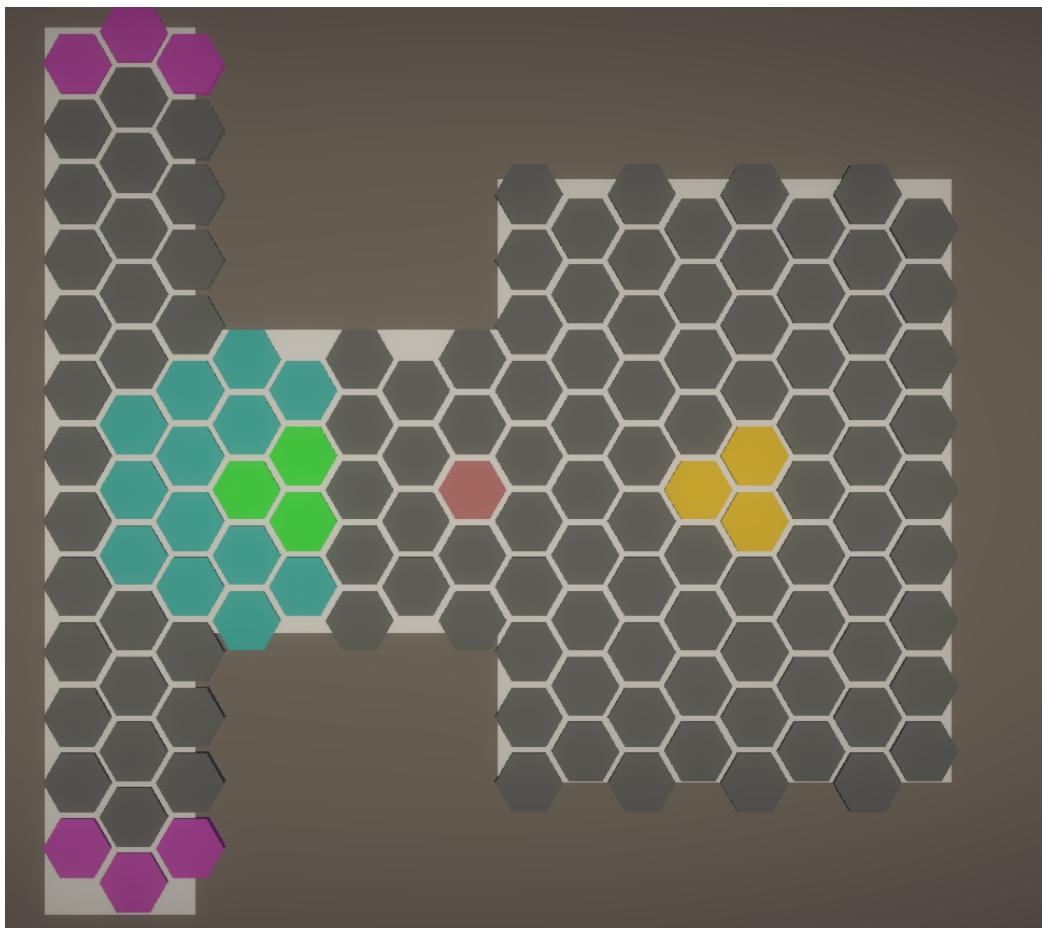


Figure 12: Harta de joc

### 3.4.2 Sistemul Turnurilor

Turnurile pot fi plasate doar pe hexagoanele de construcție. Odată plasate, acestea nu pot fi mutate. Singurele metode de a goli acel hexagon sunt să fie distrus de un inamic sau să vindem turnul respectiv. Dacă vindem un turn, vom primi înapoi o parte din banii investiți, bani cu care putem să îmbunătățim celelalte turnuri. Toate turnurile au următoarele proprietăți:

- Viteză de atac
- Putere de atac
- Viață
- Rază de atac

- Tipul inamicilor pe care pot să îi atace

Dacă viața turnului ajunge la 0, va fi distrus automat. Atâtă timp cât un turn nu este distrus, îl putem repara, dar reparațiile vor fi executate într-un anumit interval de timp, timp în care turnul nu va putea ataca, dar va putea fi lovit în continuare de inamici.

### 3.4.3 Comandantul

Comandantul poate fi mutat pe orice hexagon care nu este deja ocupat. După ce ai ales un hexagon, acesta va începe să se miște către acea locație. Cât timp este în mișcare, nu poate să atace inamicii. Destinația poate fi schimbată doar după ce a ajuns la destinația selectată. Are rază medie, rată de atac medie, putere de atac mediu și poate ataca orice tip de inamic. Acesta poate intra în turnuri, îmbunătățindu-le exponențial proprietățile.

### 3.4.4 Valul de inamici

Jocul are mai multe nivele, care pot fi selectate dintr-un meniu, acestea fiind cât mai diversificate posibil. Inamicii au mai multe puncte din care pot apărea, în funcție de nivelul ales. Aceștia vin în grupuri de inamici, între fiecare grup existând o perioada de repaus, în care jucătorul își poate repara turnurile. În caz că nu are nevoie de reparații, poate selecta să sară peste perioada de repaus, astfel primind bani extra.

### 3.4.5 Nivelele jocului

Nivelul este terminat când toți inamicii au fost omorâți. Fiecare nivel va avea un scor de maxim 3 stele, care specifică cât de bine s-a descurcat la acest nivel. Scorul este calculat în funcție de viața rămasă a bazei la finalul nivelului. Fiecare stea câștigată va da bani extra, bani care pot fi folosiți în magazinul de îmbunătățiri permanente, despre care vom discuta în scurt timp.

Jucătorul poate să joace din nou nivalele, dar nu va mai primi atât de mulți bani ca prima dată. Aceasta va câștiga bani extra de pe stelele obținute doar dacă a primit mai multe stele decât turele anterioare.

### 3.4.6 Lock-on

Jucătorul poate să apese pe inamici, astfel setând acel inamic drept o prioritate. Fiecare turn care poate ataca acest tip de inamic îl va prioritiza față de alții inamici. De îndată ce este omorât, se pierde această prioritate.

### 3.4.7 Magazinul de îmbunătățiri permanente

Între nivele, putem să cumpărăm piese de schimb pentru turnurile noastre pentru a le face mai puternice. Aceste piese se pot cumpăra doar cu banii câștigați de pe urmă nivelelor. Îmbunătățirile pentru fiecare turn în parte sunt următoarele:

- **Mitraliera automată:** atac mai puternic, rază de atac mai mare, rată de atac mai mare.
- **Gardul electric:** mai multă viață, atac mai puternic, rată de atac mai mare.
- **Vulkan:** atac mai puternic, rază de atac mai mare, atac mai rapid
- **Aruncătorul de flăcări:** atac mai puternic, rază de atac mai mare, viață mai multă.
- **Laser:** atac mai puternic, rază de atac mai mare, viață mai multă.
- **Excavator:** mai mulți bani, timpul între livrările de bani mai scurt, viață mai multă.

Fiecare îmbunătățire va avea mai multe nivele, fiecare costând din ce în ce mai mulți bani, dar vom avea un nivel maxim la care putem duce aceste îmbunătățiri.

### 3.4.8 Sistemul co-op

Când un jucător alege acest mod, este transferat la o pagină de login, în care trebuie să își specifice numele sau să îl păstreze pe cel generat automat. După ce intră în cont, trebuie să intre într-o cameră de așteptare. În acest scop, poate să creeze una nouă, să aleagă dintr-o listă cu toate camerele existente sau să intre automat într-una existentă, care este aleasă aleator. În cazul ultimei opțiuni, dacă nu există nici o cameră libera, aceasta se creează automat.

În urma intrării în cameră, poate să selecteze dificultatea nivelului, iar de îndată ce toți jucătorii din cameră sunt pregătiți, pot să inceapă nivelul.

În centrul hărții se află un monstru imens de foc. Scopul nivelului este să învingi monstrul împreună cu coechipierul tău, înainte ca monstrul să distrugă baza vreunui dintre jucători.

Fiecare jucător poate păstra turnuri oriunde, putând să îmbunătățească, repară sau vinde doar turnurile construite de el. Amândoi jucătorii au caracterele lor comandanți, dar aceștia nu pot intra în același turn în același timp. Pentru sistemul de lock-on, doar turnurile jucătorului care a setat lock-on-ul va ataca inamicul prioritizat.

### 3.4.9 Atacurile monstrului de foc

Monstrul de foc are o serie de atacuri predefinite, care devin mai puternice în dificultățile avansate.

- **Abilitatea automată:** crește puterea de atac pe măsură ce se pierde din viață.
- **Ploaia de meteoriți:** ridică o mână sus și generează meteoriți unul după altul. Fiecare meteorit, când este construit complet, alege un turn și pornește către direcția acestuia. Când se izbește de turn, explodează și scade din viață turnului în funcție de dificultatea selectată.
- **Gheara învărtitoare:** se învârte 360 grade și lovește toate turnurile de pe hartă cu mâna lui dreaptă.
- **Meteoritul suprem:** când ajunge la un anumit procentaj de viață începe acest atac. Alege una dintre bazele jucătorilor (în caz că sunt mai mulți), se rotește spre baza selectată și începe să încarce un meteorit imens. Încărcarea acestuia durează 30 de secunde, iar dacă în acest timp jucătorul reușește să îi scadă suficient de mult viață monstrului, atacul este întrerupt și devine confuz timp de 10 secunde, timp în care poate fi atacat neîntrerupt. Dacă jucătorii nu reușesc să îi scadă suficient de mult viață, va lovi puternic cu meteoritul, baza selectată și turnurile din jur.

Toate interacțiunile acestui nivel sunt sincronizate în rețea prin utilizarea modulului Photon Engine.

## 3.5 Analiza proiectului din punctul de vedere al consumatorilor

### 3.5.1 Dificultatea jocului

Jocul va fi relativ dificil. Jucătorul trebuie să se gândescă unde să construiscă anumite turnuri pentru a folosi resursele cât mai bine. La început, jocul va fi ușor, dar pe măsură ce progresează, nivelele vor deveni din ce în

ce mai dificile. Jucătorul trebuie să se gândescă și dacă poate rezista să sară peste perioada de repaus între valurile de inamici.

Sistemul co-op are un nivel de dificultate cu mult mai ridicat, pentru că trebuie să se gândească cum să se coordoneze cât mai bine cu coechipierul.

### 3.5.2 Elemente de dependență

În caz că nu putem depăsi un anumit nivel, putem juca nivelele anterioare și să ne asigurăm că obținem 3 stele la fiecare nivel. Această metodă de a obține 3 stele la fiecare nivel poate fi unul din motivele care îi determină pe jucători să continue să joace jocul. Un alt factor ar putea fi sistemul de îmbunătățiri permanente.

### 3.5.3 Grupele de vârstă vizate

Acest joc nu are limitări de vârstă, poate fi jucat de oricine, dar va fi apreciat cel mai mult de copii și de adolescenții care caută provocări în jocurile de strategie. Jocul este, în principiu, făcut pentru jucătorii ocazionali de jocuri video.

## 4 Implementarea proiectului

### 4.1 Scena de luptă

#### 4.1.1 Inițializarea scripturilor

În Unity scripturile au câteva metode care ajută la inițializare.

**Awake** este metoda care este apelată la începutul programului și înainte de restul metodelor.

**Start** este metoda care este apelată după ce au fost apelate metodele "Awake" de la toate scripturile.

**Update** este apelat la fiecare cadru al jocului, aici intervenind o mare parte din logica jocurilor. Apelarea update-ului începe după ce toate metodele "Start" au fost apelate.

Pentru "Start" și "Awake", Unity decide automat în ce ordine să le apeleze, fără ca noi să putem controla acest lucru. În multe proiecte mai

complexe, se ajunge în momentul în care, pentru initializea anumitor sisteme, este necesar ca alte sisteme să fie deja inițializate. Cum nu putem controla ordinea de initializare, se ajunge în punctul în care putem primi erori aleatorii din cauza inițializării haotice.

Pentru a combate acest lucru, am definit un sistem care va permite controlarea inițializării tuturor acestor procese cu ușurință.

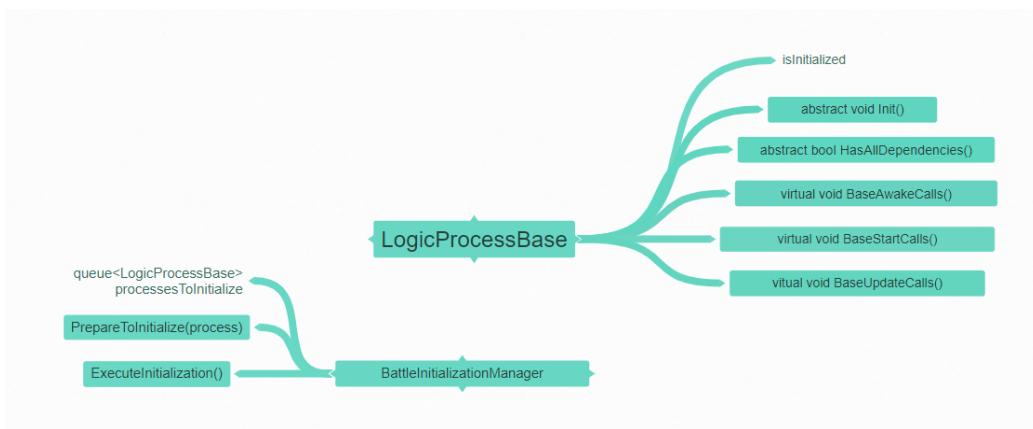


Figure 13: Inițializarea scripturilor

Am creat clasa LogicProcessBase care ajută în acest proces. Fiecare script care necesită o inițializare mai organizată trebuie să o moștenească. Aceasta conține:

- **isInitialized** care specifică dacă scriptul curent a fost inițializat sau nu încă.
- **Init()**. În această metodă trebuie să scriem tot codul de inițializare a clasei care moștenește.
- **HasAllDependencies()**. Este o metodă care returnează un boolean. În această metodă trebuie să punem toate dependințele de care are nevoie clasa care moștenește.
- **BaseAwakeCalls()**. În această metodă apelăm ”PrepareToInitialize()” din clasa ”BattleInitializationManager”, despre care vom vorbi în scurt timp.
- **BaseStartCalls()** și **BaseUpdateCalls()** sunt metode în care trebuie să scriem tot codul care normal ar fi venit în Start/Update. În Unity,

nu putem defini Start/Update drept virtual și să modificăm funcțiile lor pe urmă. Din acest motiv, tot codul din clasele dintr-o ierarhie mai complexă trebuie scris în aceste metode, iar în clasa cel mai jos din ierarhie trebuie să le apelăm în Awake/Start/Update.

**BattleInitializationManager** este a doua clasa de care avem nevoie. Ea este responsabilă pentru a executa inițializarea propriu-zisă a proceselor.

**PrepareToInitialize()** este apelată de fiecare proces în metoda lor de Awake, iar această metodă va adăuga procesul respectiv într-o coadă.

**ExecuteInitialization()** parcurge toate procesele din coadă și verifică dacă acestea pot fi inițializate, apelând "HasAllDependencies()". Dacă procesul curent poate fi inițializat, atunci apelează metoda Init() și setează "isInitialized = true". Dacă nu îl poate inițializa, îl adaugă la finalul cozii pentru a fi inițializat la final. Acest proces se repetă până când coadă este goală sau s-a parcurs o dată coada cap-coadă și nu s-a inițializat nici un proces. În acest caz, înseamnă că aveau dependințe circulare și oprim procesul de inițializare, afișând un mesaj de eroare. Această metodă trebuie apelata în Awake, Start și Update, deoarece, pe parcurs pot apărea noi procese care trebuie inițializate, iar programul trebuie să poată detecta astfel de cazuri.

#### 4.1.2 Procesarea comenzielor venite de la utilizator

O altă problema des întâmpinată în proiectele mari, este procesarea comenzielor venite de la utilizator. În cazul în care procesăm comenzi în mai multe script-uri, când dorim să schimbăm modul în care funcționează comenzi sau să schimbăm platforma pe care rulează jocul, va trebui să schimbăm toate scripturile în care procesăm comenzi utilizatorului. În proiectele mari, acest lucru duce la mult timp pierdut și din acest motiv am definit o metodă de a organiza mai bine acest sistem.

Ideea propusă de mine este să păstrăm toată procesarea de comenzi ale utilizatorului într-un singur script. Acest script are definite evenimente/delegate pentru toate tipurile de input specifice. Alte scripturi se pot abona la aceste evenimente, iar când utilizatorul apasă tasta respectivă, toate metodele abonate la evenimentul respectiv vor fi apelate.

Pe lângă asta, este necesar să știm și anumite informații legate de comanda primită de la utilizator. Informațiile adiționale de care am avut nevoie sunt:

- Poziția pe ecran la care a fost apăsat mouse-ul/ecranul (în cazul rulării pe Android)
- Poziția pe ecran la care a fost ridicată apăsarea mouse-ului/ecranului (în cazul rulării pe Android)
- Dacă în clipa curentă este sau nu apăsat mouse-ul/ecranul
- Timpul la care a fost apăsat mouse-ul/ecranul
- Timpul la care a fost ridicată apăsarea mouse-ului/ecranului
- Obiectul pe care utilizatorul a dat click în scenă. În momentul când apăsăm click, se creează un Raycast care are ca origine poziția mouse-ului pe ecran și care se îndreaptă către scenă în perspectiva în care se află camera la clipa curentă. Acest Raycast detectează dacă a fost lovit un obiect, și dacă da, reținem care obiect a fost lovit pentru ca alte scripturi să se poată folosi de această informație.
- Elementul cel mai de sus din ierarhia obiectului lovit. Datorită faptului că anumite modele au o structură complexă, uneori nu ne este deloc util să știm în mod direct ce obiect a fost lovit. De exemplu, în cazul în care apăsăm peste un turn și Raycast-ul lovește pușca turnului, nu ne va ajuta deloc să știm că am lovit o pușcă în scenă. În anumite cazuri va fi cu mult mai util să stim tipul turnului a cărui pușcă am lovit-o. Din acest motiv, în clipa în care găsim obiectul pe care am dat click, reținem și părintele cel mai de sus din ierarhie.

Această organizare a procesării comenziilor ne permite să schimbăm tastele folosite cu ușurință și să adăugăm suport pentru dispozitive noi foarte ușor. În cazul dispozitivelor noi, putem folosi directive de Unity pentru a separa codul specific pentru calculator de cel specific pentru Android sau orice alt dispozitiv. Procesarea comenziilor în cele 2 cazuri vor fi diferite, dar datorită faptului că se invocă evenimentele deja definite, alte script-uri nu trebuie modificate absolut deloc.

#### 4.1.3 Harta de joc

Acțiunea jocului se petrece pe o hartă care este formată din mai multe blocuri hexagonale. Această hartă este generată automat în funcție de ce tip de hartă alegem pentru nivelul respectiv. Înainte să începem să investigăm detaliat sistemul hărții de joc, trebuie mai întâi să definim ce reprezintă un

bloc hexagonal pe hartă.

### Blocul hexagonal

Pentru a reține informații specifice pentru un bloc, am definit clasa HexagonalBlock. În interiorul clasei am folosit două enum-uri: **SpawnPointsID** și **HexagonType**.

HexagonType definește tipul blocului hexagonal și interacțiunile posibile cu acesta. Tipurile definite în acest enum sunt:

- **Walkable**. Blocul primește culoarea gri și toate creaturile pot să se deplaseze pe acesta.
- **TurretBuildable**. Blocul primește culoarea albastră și reprezintă zonele de pe harta unde putem să construim turnurile noastre.
- **ResourceExtraction**. Blocul primește culoarea verde și reprezintă zonele de pe harta în care putem construi turnul excavator.
- **Occupied**. Blocul primește culoarea roșie și reprezintă zonele în care este construit un turn sau locul în care se află comandantul.
- **SpawnPoint**. Blocul primește culoarea purpurie și reprezintă locul din care pornesc inamicii.
- **Impassable**. Blocul primește culoarea roșu închis și reprezintă locurile prin care nu pot să treacă inamicii. Sistemul de navigare va ignora toate aceste blocuri și va găsi rute alternative.
- **PlayerBase**. Blocul primește culoarea galbenă și reprezintă locul în care va fi creată baza jucătorului. Pentru ca un nivel să fie considerat valid, trebuie să existe 3 astfel de blocuri unul lângă altul.
- **CommanderSpawn**. Blocul primește culoarea roz și reprezintă locul în care va începe comandantul. Pentru ca un nivel să fie considerat valid, trebuie să existe exact o instanță a acestui bloc pe hartă.

SpawnPointsID definește o serie id-uri pentru blocurile de tip SpawnPoint. În fișierul de configurare al valului de inamici, discutat în capitolul [4.1.10](#), putem specifica id-ul blocului de la care vor începe inamicii. În cazul în care, pentru un val de inamici specificăm ca inamicii să începă de la o locație aleatoare, atunci se va alege un bloc aleatoriu dintre toate blocurile de pe hartă.

Blocul hexagonal definește și o listă de materiale, care sunt folosite pentru a schimba culoarea blocului în momentul în care tipul acestuia se schimbă.

Are definite și 2 metode ajutătoare și anume PlaceHexagon(), care așează blocul pe hartă în funcție de poziția și diametrul specificat în parametrii, și UpdateMaterial() care actualizează culoarea blocului, prin schimbarea materialului pe care îl folosește.

Pentru a pune în funcțiune schimbarea de culoare a blocului hexagonal direct din editor, a trebuit să creez clasa specială HexagonalBlockEditor. Această clasă moștenește clasa Editor, pentru a putea schimba interfața grafică a editorului din Unity. Această clasa, în metoda OnInspectorGUI care este apelată de fiecare dată când obiectul este selectat, verifică dacă s-a schimbat tipul blocului, și dacă da, atunci apelează metoda UpdateMaterial() de pe blocul selectat.

### Harta propriu-zisă

Clasa specială definită pentru harta de joc este HexagonalGrid. Această clasa are câteva proprietăți publice care definesc dimensiunea hărții: diameter (care definește cât de mare este fiecare hexagon), offset (care este aplicat în așa fel încât hexagoanele vor fi poziționate la distanțe egale între ele) și mapScaleOffset (harta de joc este scalată automat la dimensiunea ecranului; acest offset ne ajută să scalăm harta mai mult sau mai puțin decât ar fi fost scalată automat).

Scriptul are definit o listă de tipul HexagonalBlock, care reține toate blocurile hexagonale de pe hartă și mai are definit o variabilă publică care reprezintă tipul de hartă pe care o folosim. Tipul de harta este reprezentat de un model 3D care va fi folosit pentru a poziționa hexagoanele.

Clasa moștenește LogicProcessBase, iar în metoda Init(), găsește fișierul de configurare al nivelului curent, discutat în capitolul 4.1.11, șterge toate blocurile existente de pe hartă și citește fișierul de configurare al hărții (definit în fișierul de configurare al nivelului și explicit în secțiunea următoare). După ce a citit conținutul fișierului, apelează metoda SpawnAndScaleMap() și metoda LoadPresetGrid().

SpawnAndScaleMap() creează o instanță a hărții de joc (modelul 3d care este definit drept variabilă publică), găsește limitele acesteia în coordonate 3D, și în funcție de dimensiunea ecranului, scalează harta în așa fel încât să încapă harta în întregime pe ecran. Pe urmă, în funcție de mapScaleOffset,

scaleaza din nou harta în funcție de dorința persoanei care a definit nivelul.

Înainte să vorbim despre funcția LoadPresetGrid() trebuie să discutăm despre CreateGrid(). Această funcție se folosește de modelul 3D al hărții pentru a genera și poziționa toate blocurile hexagonale necesare. Aceasta parcurge întreagă dimensiune a modelului 3D, și folosindu-se de fizica definită de Unity (Physics.Raycast), verifică dacă în poziția curentă de pe model a reușit să lovească suprafața. În caz că a reușit să lovească, atunci generează un bloc hexagonal, îl adaugă în listă și caută în continuare până când a reușit să parcurgă întreg modelul 3D. La final o să avem generate pe hartă blocuri hexagonale care sunt poziționate în toate zonele de pe modelul 3D al hărții, fiecare bloc fiind de tipul Walkable.

LoadPresetGrid() preia conținutul fișierului configurabil, apelează metoda CreateGrid(), după care începe să itereze peste fiecare element din blocurile generate. La fiecare iterare, initializează blocul curent în funcție de datele din fișierul configurabil al hărții de joc, și în caz că întâlnește blocuri de tipul PlayerBase sau CommanderSpawn, generează și poziționează pe harta comandantul și baza jucătorului. În caz că fișierul este corrupt, nu a găsit blocurile pentru comandant și pentru baza jucătorului, sau a găsit prea multe din acestea, afișează mesaje de eroare pentru fiecare caz în parte.

### **Fișierul de configurare pentru harta de joc**

Pentru a putea crea cu ușurință nivele cât mai variate, trebuie să reținem anumite date în fișiere configurabile. Aceste fișiere configurabile trebuie să fie făcute în aşa fel încât prin încărcarea unui simplu fișier, putem genera un nivel de joc complet funcțional.

În acest scop am definit clasa GridSaveData, care reține proprietățile cele mai importante ale hărții de joc. Această clasa reține diametrul blocurilor, offset-ul și mapScaleOffset, pentru a putea încărca harta la aceeași scara la care era configurația inițială. Pe lângă asta, reține o listă cu tipurile blocurilor hexagonale de pe hartă și id-urile setate pentru blocurile de pe care încep inamicii.

Această clasa este definită urmând o structură JSON, pentru a ușura serializarea și deserializarea datelor în momentul scrierii și citirii pe disk.

Pe lângă această clasa, am avut nevoie și de o clasa ajutătoare care să se ocupe de salvarea și încărcarea propriu-zisă a datelor. Clasa se numește GridDataSaver și este definită cu 2 metode statice simple: SaveData() și

LoadData().

### **Editorul special pentru crearea rapidă a nivelelor**

Pentru a ușura generarea de nivele, am creat un editor cu funcții speciale. Clasa se numește HexagonalGridEditor și mosteneste clasa Editor din Unity. Folosind această clasă, am suprascris editorul implicit pentru clasa HexagonalGrid, și în loc am adăugat câteva proprietăți ajutătoare. Am definit câmpuri publice pentru modelul 3D al hărții pentru care dorim să creăm un nivel, diametru, offset, mapScaleOffset și locația fișierului de configurare în care să salveze configurarea hărții și din care poate să încarce harta pentru a o modifica.

Pe lângă aceste câmpuri, am definit și câteva butoane care au drept rol să apeleze funcționalități din scriptul HexagonalGrid. Fără aceste butoane, nivelele ar fi putut fi create numai în timpul rulării aplicației, fapt ce nu era de dorit, deoarece este cu mult mai ușor să creăm hărți de joc direct din editor. Butoanele definite sunt: Scale Map, Reset Scale, Generate Grid, Clear Grid, Load Preset, Save Preset. Fiecare buton apelează funcționalitatea cu același nume din HexagonalGrid, exceptând Load Preset și Save Preset, care încarcă și salvează configurarea actuală a hărții în locația specificată prin câmpul public, folosindu-se de GridDataSaver.

Datorită acestui editor special, procesul de creare a hărților jocului a devenit foarte rapid. Pentru a crea o hartă nouă, trebuie să încarcăm un model 3D simplist definit într-un program de modelare 3D și să tragem acel model 3D în câmpul public de pe editor. Modificăm câmpurile de scalare a hărții și apăsăm pe butonul de Generate Grid repetat până când suntem mulțumiți de configurarea actuală. După acest lucru, selectăm blocurile hexagonale la care dorim să le schimbăm tipul. Datorită editorului HexagonalBlockEditor, culoarea acestora se schimbă automat, fapt care ne ușurează muncă enorm. După ce am definit toate tipurile blocurilor, specificăm locația la care dorim să salvăm fișierul de configurare al hărții și apăsăm pe butonul Save Preset.

În cazul în care dorim să modificăm configurarea unei hărți, specificăm locația fișierului de configurare a hărții respective. Setam câmpul modelului 3D cu modelul 3D pe care a fost generată inițial harta și apăsăm pe butonul Load Preset. După ce harta s-a încărcat cu succes, putem modifica blocurile hexagonale în modul dorit, iar când am terminat, apăsăm din nou pe butonul Save Preset pentru a suprascrie configurarea hărții.

#### 4.1.4 Sistemul de navigare

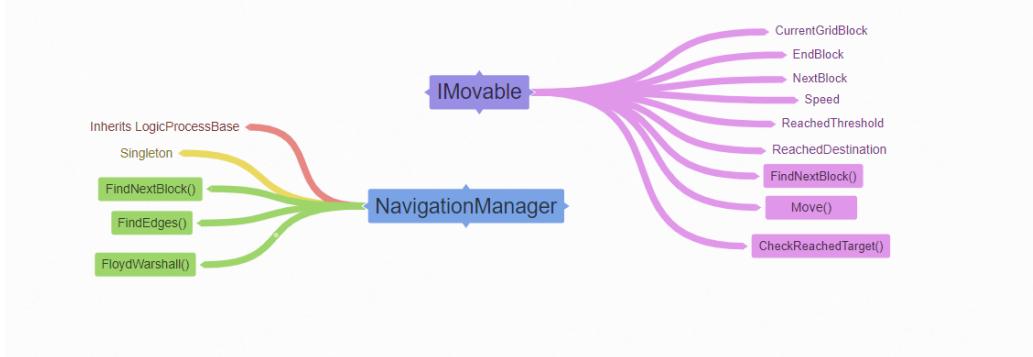


Figure 14: Sistemul de navigare

Caracterele jocului trebuie să știe pe unde pot să se deplaseze ca să ajungă la o anumită destinație. În acest scop a trebuit să aleg un algoritm care să găsească cel mai scurt drum posibil. Am avut de ales între mai mulți algoritmi specifici, precum: Dijkstra, A\*, Floyd-Warshall, etc.

În cele din urmă am ales să folosesc Floyd-Warshall. Acest algoritm are complexitatea  $O(n^3)$  și presupune construirea celor mai scurte drumuri posibile pornind de la oricare nod către oricare alt nod din rețea. Deoarece construirea drumului se realizează doar la începutul jocului și reconstruirea acestui drum se realizează aproape instantaneu, am ales să folosesc acest algoritm.

”Considerăm rețeaua orientată  $G = (N, A, b)$  reprezentată prin matricea valoare adiacentă  $B = (b_{ij}), i, j \in N$  cu

$$b_{ij} = \begin{cases} b(i, j) & \text{daca } i \neq j \text{ și } (i, j) \in A; \\ 0 & \text{daca } i = j; \\ \infty & \text{daca } i \neq j \text{ și } (i, j) \notin A. \end{cases}$$

Algoritmul Floyd-Warshall determină matricea distanțelor  $D = (d_{ij}), i, j \in N$  și matricea predecesor  $P = (p_{ij}), i, j \in N$ .“ [3]

```

function FLOYD-WARSHALL
    for i ← 1 to n do
        for j ← 1 to n do
             $d_{ij} \leftarrow b_{ij};$ 
            if i ≠ j and  $d_{ij} < \infty$  then

```

```

 $p_{ij} = i;$ 
else
 $p_{ij} = 0;$ 
end if
end for
end for
for k  $\leftarrow$  1 to n do
    for i  $\leftarrow$  1 to n do
        for j  $\leftarrow$  1 to n do
            if  $d_{ik} + d_{kj} < d_{ij}$  then
                 $d_{ij} = d_{ik} + d_{kj};$ 
                 $p_{ij} = p_{kj};$ 
            end if
        end for
    end for
end for
end function

function RECONSTRUIRE DRUM
    k = n;
     $x_k = j$ 
    while  $x_k \neq i$  do
         $x_{k-1} = p_{ix_k};$ 
        k = k - 1
    end while
end function

Drumul minim este  $D_{ijp} = (x_k, x_{k+1}, \dots, x_{n-1}, x_n) = (i, x_{k+1}, \dots, x_{n-1}, j)$ 

```

#### 4.1.5 Interfețe folosite

Pentru că o serie de obiecte din joc trebuie să urmeze aceleași concepte, am decis să mă folosesc de interfețe pentru a le defini modul în care trebuie să funcționeze.

##### **IMovable**

Deoarece inamicii și comandanții se pot mișca pe hartă, am creat această interfață pentru a defini aceleași concepte pentru toate caracterele care au nevoie să se miște pe hartă. Interfața se folosește de sistemul de navigare definit anterior și conține următoarele:

- Nodul curent la care se află.
- Nodul destinație.

- Nodul următor la care trebuie să se mute ca să se apropie de destinație.
- Viteza de deplasare.
- Distanța față de următorul nod pentru care considerăm că am ajuns la acesta. Deoarece lucrăm cu obiecte în spațiu 3D, este nevoie de o asemenea valoare.
- Un boolean care definește dacă am ajuns sau nu la destinație.
- O metodă care setează nodul următor la care trebuie să ajungem ca să ne apropiem de destinație. Aceasta face apel la matricea construită de sistemul de navigare.
- O metodă care mișcă propriu-zis obiectul către următorul nod.
- O funcție care verifică dacă am ajuns sau nu la destinație.

### **IAttacker**

Este o interfață pe care toate obiectele/caracterele care doresc să atace alte obiecte trebuie să o moștenească. Aceasta conține următoarele date:

- Obiectul pe care trebuie să îl atace. Tipul este definit la moștenirea clasei datorită şablonului definit pentru interfață.
- Puterea de atac.
- Raza de atac.
- Durata între atacuri.
- Timpul la care s-a executat ultimul atac.
- Intervalul de timp între care se caută un nou inamic.
- Metoda care afisează/ascunde raza obiectelor din scenă. Raza acestora a fost definită într-un shader special creat în Shader Graph.
- O metodă care caută cel mai apropiat inamic dacă există în rază.
- Metoda de atac pe care fiecare obiect o implementează diferit.

### **IDestroyable**

Fiecare obiect care are o viață anume și care poate fi distrus trebuie să moștenească această interfață. Aceasta conține următoarele date:

- Viața obiectului.
- Banii primiți când obiectul este distrus
- O metodă care distrugă obiectul respectiv când viața lui a ajuns la 0.

### IRrecoverable

Turnurile pot să își refacă viața, iar ca un concept pentru viitor, ar putea să existe și anumiți inamici care refac viața altor inamici. Interfața conține următoarele date:

- Câtă viață se reface în fiecare secundă când efectul este aplicat.
- Costul refacerii în cazul în care este de dorit un cost.
- Un boolean care verifică dacă se reface sau nu în clipă curentă.
- O metodă care pornește procesul de refacere.

#### 4.1.6 Ierarhia inamicilor

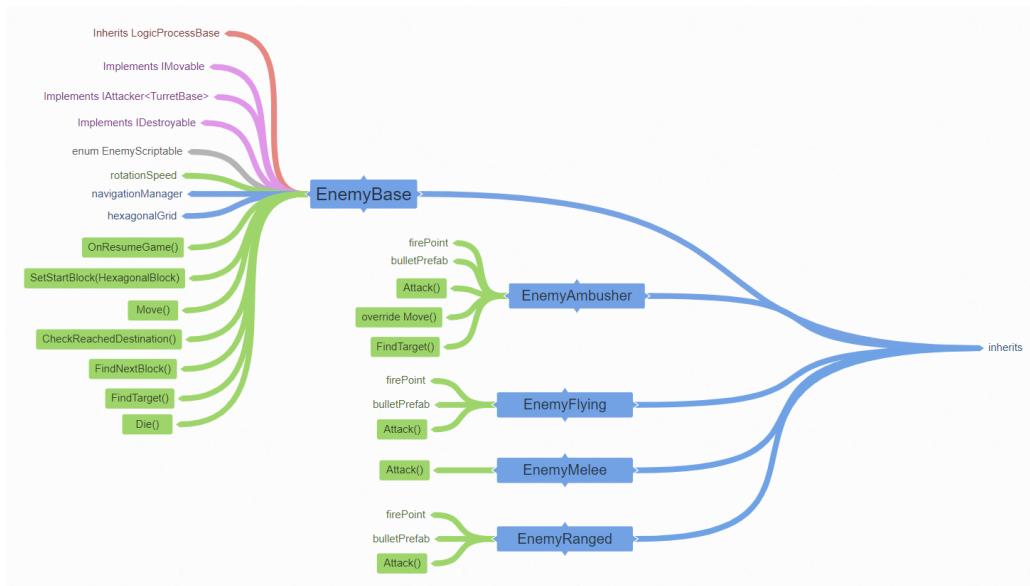


Figure 15: Ierarhia inamicilor

În fig: 15 am definit într-un mod simplu și colorat modul în care trebuie structurată ierarhia inamicilor din joc.

Clasa EnemyBase este clasa părinte care definește modul principal în care inamicii trebuie să funcționeze. Această clasă implementează 3 interfețe utile, și anume IMovable, IDestroyable și IAttacker<TurretBase>. Precum am explicat și anterior, interfața IAttacker definește un şablon care specifică tipul de obiect pe care poate să îl atace, în cazul nostru, clasa TurretBase care va fi explicată în următoarea secțiune.

Această clasă face referire și la sistemul de navigare și la sistemul hărții hexagonale, și din acest motiv este nevoie să își facă inițializarea doar după ce aceste două sisteme și-au terminat inițializarea lor. În acest scop, putem să moștenim clasa LogicProcessBase, care a fost descrisă în capitolul 4.1.1.

Funcțiile implementate de această clasa sunt în mare parte funcțiile moștenite după implementările interfețelor descrie, cu câteva excepții referitoare la alte sisteme din joc. Un astfel de sistem este cel de a pune pe pauză jocul. Cand jocul este pus pe pauză, toate scripturile care se folosesc de timpul actual al jocului nu vor acționa corespunzător după reluarea jocului. Din acest motiv este nevoie să implementăm funcția OnResumeGame() care va fi apelata când se reia jocul, iar în această funcție trebuie să setam toti temporizatorii din script la valorile necesare.

Această clasa nu este suficientă pentru a defini toate tipurile de inamici, deoarece face dificilă diferențierea între inamici pentru anumite sisteme și anumiți inamici acționează diferit față de standard. Din acest motiv este necesar să definim o serie de clase pentru fiecare inamic în parte, care să moștenească această clasa de bază, și care să își adauge propria lor logică la standardul definit de EnemyBase.

**EnemyAmbusher** nu se folosește de sistemul de navigare. Este inamicul bombardier, care zboară prin aer direct către baza jucătorului. Când ajunge în contact cu aceasta, lansează bombe care provoacă multe daune, după care își continuă drumul, părăsind scenă de luptă. Din acest motiv, metodele Move() și FindTarget() trebuie să fie suprascrise.

**EnemyFlying** și **EnemyRanged** sunt inamici care urmează în totalitate logica definită de EnemyBase. Singura excepție la aceste două clase este că anumite turnuri vor face diferențiere între ele. O parte din turnuri pot să atace doar inamicii de sol (EnemyRanged și EnemyMelee), iar alte turnuri pot să atace doar inamicii zburători (EnemyFlying și EnemyAmbusher).

**EnemyMelee** este un inamic care poate să atace alte turnuri doar când

a ajuns lângă acestea. Din acest motiv metoda lui de atac trebuie să fie suprascrisă puțin.

#### 4.1.7 Ierarhia turnurilor defensive

Ierarhia turnurilor este puțin mai complexă și va fi descrisă pe bucăți.

##### TurretBase

Urmând aceeași logică ca la inamici, am definit o clasa de bază pentru modul în care trebuie să acționeze un turn defensiv. Acesta implementează două interfețe: IDestroyable și IAttacker<EnemyBase>. Şablonul de la IAttacker definește faptul că un turn defensiv poate să atace obiecte de tipul EnemyBase, care este clasa din care moștenesc toți inamicii jocului.

TurretBase are și ea nevoie de o initializare controlată și din acest motiv trebuie să moștenească clasa LogicProcessBase. TurretBase definește proprietățile turnurilor, cum ar fi viață, puterea de atac, etc. Pe lângă asta, majoritatea turnurilor pot fi îmbunătățite, proprietățile crescând exponențial când acest lucru se întâmplă. Din acest motiv a fost nevoie să creez fișiere configurabile care definesc valori prentru proprietățile turnurilor. Această clasa deține o instanță a unui astfel de fișier, și definește o funcție ”SetLevelProp(int level)”, care citește fișierul configurabil și setează toate proprietățile în funcție de acesta. Pe lângă asta, când turnurile sunt îmbunătățite, le schimbă modelul 3D pentru a avea o schimbare vizuală.

Clasa definește și alte metode care au fost abordate în capitolele anterioare, precum: Die(), OnResumeGame(), Init(), etc. Funcția DrawRange() afișează/ascunde raza de atac a turnului, rază care a fost construită într-un shader special definit în Shader Graph.

##### PlayerBase

PlayerBase este o clasa care moștenește TurretBase, și reprezintă baza de operații a jucătorului. Aceasta implementează o parte din acțiunile definite în interfețe, precum: Attack(), FindTarget(), etc.

Pe lângă asta, când baza jucătorului rămâne fără viață și este distrusă, jocul trebuie să se termine. Acest lucru poate fi realizat foarte ușor prin suprascrierea metodei Die(). Când baza rămâne fără viață, încearcă managerul jocului, iar apoi acesta își pornește procesul de încheiere al nivelului. Mai multe detalii vor fi discutate în capitolul 4.1.12.

## **BuildableTurret**

BuildableTurret este clasa care moștenește TurretBase și definește funcționalitățile pentru toate turnurile care pot fi construite pe harta de joc. Această clasa implementează IRecoverable, care definește modul în care turnurile pot să își refacă viața. De îndată ce turnul a fost lovit de un inamic, se calculează un cost necesar pentru a reduce turnul la viață maximă. În cazul în care jucătorul are suficienți bani și dorește să refacă turnul, acesta va începe procesul refacere. Pe parcursul câtorva secunde, se va reface treptat pentru viață lipsă în momentul începerii acestui proces. În acest timp, turnul nu poate să atace inamicii, dar poate să fie lovit în continuare de aceștia, deci nu este garantat că va reveni la viață maximă după terminarea procesului.

O altă funcționalitate majoră a turnurilor este opțiunea de a îmbunătăți turnurile. Când utilizatorul selectează un turn, se caută în fișierul configurabil costul necesar pentru a îmbunătăți turnul la nivelul următor, în caz că nu a ajuns încă la nivelul maxim. În cazul în care jucătorul are suficienți bani, se apelează metoda Upgrade(), care verifică dacă s-a ajuns la nivelul maxim și dacă nu, dezactivează toate stările defavorabile ale turnului (cum ar fi raza), crește nivelul și apelează metoda SetLevelProp() pentru a citi din nou fișierul de configurație și pentru a actualiza starea turnului în funcție de nivelul nou.

Turnul poate fi vândut de jucător în caz că este în criză de bani sau vrea să mute turnul în altă locație. În acest caz se calculează banii pe care îi primește jucătorul, se dezactivează toate stările și funcționalitățile turnului și se distrug referințele specifice. Tote acestea se întâmplă în metoda Sell-Turret() implementată în această clasă.

Ultima funcționalitate adăugată de această clasa este cea de a permite turnului să fie controlat de comandantul jocului. Această funcționalitate va fi descrisă în detaliu în secțiunea [4.1.8](#)

## **Turnurile specifice**

Cel mai jos nivel din ierarhie presupune definirea claselor specifice pentru fiecare tip de turn în parte, fiecare dintre acestea moștenind clasa Buildable-Turret și adăugând/modificând logica stabilită până în acest punct în funcție de caz.

- **TurretElectricFence.** Este turnul care are viață foarte mare dar rază de atac foarte mică. Poate să atace doar inamicii de luptă apropiată și

acționează drept un zid care păzește toate celelalte turnuri de atacurile inamicilor. Acest turn modifică funcționalitatea prin care se găsește ce inamic trebuie să atace și modul în care atacă inamicul.

- **TurretExcavator.** Acest turn este cel mai diferit de normă, în sensul în care nu poate ataca deloc inamicii. Este un turn care la un anumit interval de timp obține resurse, pe care le convertește în bani de joc. Din acest motiv a fost nevoie să scape complet de căutarea inamicilor, iar metoda de atac a fost suprascrisă în funcționalitatea de obținere de bani.
- **TurretFlamethrower.** De îndată ce un inamic intră în raza acestui turn, lansează flăcări violente în direcția inamicului, flăcări care rănesc toți inamicii care stau în ele. Turnul priorizează inamicii de atac de aproape, dar în cazul în care nu există astfel de inamici, atacă inamicii de la distanță. Nu poate să atace inamicii zburători sau pe cei bombardieri.
- **TurretMachineGun.** Este turnul universal care poate să atace toți inamicii jocului. Acesta lansează gloanțe foarte rapid către inamici, dar gloanțele lansate individual nu provoacă foarte multe daune. Fiecare nivel presupune o viteză de atac mai mare.
- **TurretLaser.** Acest turn atacă inamicii cu un laser foarte puternic care distrugă compozitia moleculară a inamicului în fiecare clipă în care aceștia sunt în raza turnului. Poate ataca inamicii de atac apropiat și cei de la distanță, dar îi priorizează pe cei de la distanță. Nu poate ataca inamicii zburători sau pe cei bombardieri.
- **TurretVulkan.** Este turnul special împotriva inamicilor zburători și a celor bombardieri. Nu poate ataca inamicii de sol, dar în schimb lansează rachete către cei zburători, rachete care explodează când ajung în contact cu aceștia.

#### 4.1.8 Comandantul

Comandantul este un caracter special care poate fi mișcat oriunde dorim pe hartă și care atacă inamicii din raza lui de atac, cât timp nu se deplasează către o nouă locație.

În acest scop, clasa Commander implementează interfețele IMovable și IAttacker<EnemyBase>. Comandantul nu poate să fie distrus și din acest

motiv nu este necesară implementarea interfeței IDestroyable.

Comandantul are și el un fișier de configurare care specifică statusurile lui, cum ar fi viața, viteza de mișcare, puterea de atac, etc. Pentru a se putea deplasa pe hartă, are nevoie să facă referire la sistemul de navigare și la cel al hărții de joc. Din acest motiv este necesar să moștenească LogicProcessBase, ca să își execute inițializarea doar după ce celelalte sisteme au reușit să își termine pe a lor.

Majoritatea funcțiilor sunt similare cu cele ale turnurilor (Attack, DrawRange, FindTarget, etc.) așa că nu vor fi reluate aici.

Pe lângă aceste funcționalități, comandantul poate să intre în turnuri și să le controleze, mărindu-le astfel productivitatea. Când acesta intră în turnuri, le mărește raza, viteza, puterea de atac și în cazul excavatorului de resurse, crește banii primiți la fiecare livrare de resurse. Ca să poată intra în turnuri, trebuie să se deplaseze către acel turn la comanda jucătorului, iar când turnul este distrus/vândut, acesta ieșe din turn și se mișcă către cel mai apropiat nod din graful hărții de joc care nu este deja ocupat. Comandantul poate să iasă din turn și fără ca acesta să fie distrus, dar acest lucru se întâmplă numai la comanda jucătorului.

#### 4.1.9 Punerea pe pauză a jocului

Jucătorul are la dispoziție opțiunea de a pune pe pauză jocul. Unity pune la dispoziție mai multe moduri de a pune pe pauză jocul. Unul din aceste moduri este de a modifica o variabilă din clasa globală Time, și anume Time.timeScale. Această variabilă are în mod normal valoarea 1, dar dacă o modificăm, toate sistemele jocului vor rula la o altă viteză. Dacă îi setăm o valoare de 0.5, majoritatea sistemelor vor încetini (animării, sisteme de particule, rata de apelare a metodelor update, etc.). În caz că setăm această variabilă la 0, toate sistemele care sunt dependente de timp se vor opri din funcționa până când se schimbă această variabilă.

Este o metodă validă de a crea un sistem care să pună pe pauză jocul, dar deoarece oprește toate funcțiile update din a mai fi apelate și deoarece nu putem controla ce sisteme sunt opriți, nu este o metodă folosită foarte des. Un motiv principal pentru care nu este de recomandată această soluție este pentru că va opri inclusiv sunetele din joc, fapt ce nu este de dorit în cele mai multe cazuri.

O altă metodă de a implementa un sistem de pauză, este să avem un script care ține referințe către toate celealte script-uri din scenă și care, în momentul în care jucătorul pune pe pauză jocul, oprește scripturile dorite din a mai fi executate. Unity are o clasă specială numită MonoBehaviour, care ne dă toate funcțiile discutate anterior (Start, Update, Coroutine, OnTriggerEnter, etc.) și multe alte proprietăți utile. Toate scripturile care moștenesc această clasa pot fi puse pe obiecte 3D din scenă, iar scripturile care nu implementează clasa MonoBehaviour nu pot fi puse pe obiecte din scenă. Acestea din urmă vor fi rulate numai când creăm obiecte de tipul acelor clase, sau în cazul claselor statice, când apelăm metode din ele. Clasele care moștenesc MonoBehaviour și care sunt puse pe obiecte 3D din scenă au o bifă lângă numele scriptului. În caz că bifă este dezactivată, scriptul nu va rula, iar această bifă poate fi setată și direct din cod, atât timp cât avem referință la acest script.

Revenind la ideea principală, script-ul care definește sistemul de pauză ar avea referințe la toate scripturile din scenă și decide pe care dintre acestea să le dezactiveze și pe care să le lase active în clipa în care jocul se pune pe pauză. Acest sistem nu presupune o implementare foarte frumoasă, deoarece în scene diferite putem avea scripturi diferite, iar acest lucru ne-ar determina să scriem mai multe scripturi de sistem de pauză, câte unul pentru fiecare scenă diferită în parte. Pe lângă asta, obținerea tuturor referințelor către scripturile din scenă este un proces greu de automatizat. Unity oferă câteva metode pentru găsirea scripturilor din scenă (FindObjectByType<type>(), FindObjectsByType<type>(), FindObjectByName<type>()), dar acestea nu pot găsi scripturile care sunt dezactivate. Alternativa ar fi să definim o lista sau variabile publice în care să punem scripturile dorite, dar acest proces presupune o pierdere de timp pentru programator și poate produce erori foarte ușor în caz că acesta uită să adauge scriptul în lista specifică.

Varianta pe care am implementat-o în cele din urmă este de a defini un script care pune pe pauză jocul și care are o serie de proprietăți care pot fi accesate de alte scripturi. Celealte scripturi din scenă, în funcție de aceste proprietăți își vor modifica stările/sistemele în clipa în care jocul este pus pe pauză sau se reia. Această clasa am denumit-o GamePauseManager iar structura acesteia o vom explora în continuare.

GamePauseManager definește un Singleton, care este un concept de programare introdus în C#. Singleton în Unity funcționează pe principiul că există o singură instanță a clasei în scenă, și din acest motiv, orice script poate face referire cu ușurință la această clasa, fără să mai fie nevoie să

definim variabile publice și să tragem scriptul în acele variabile publice. Singleton este menit pentru sistemele majore ale jocurilor/aplicațiilor, sisteme care au o singură instanță în scenă și care influențează o mare parte din scripturi.

În Unity ca să definim un Singleton, în clasa dorită, definim o variabilă statică care are drept tip, clasa din care face parte (de cele mai multe ori variabila este numită instance). În metoda de Awake verificăm dacă acest instance este null, și dacă este, atunci setăm variabilă instance să fie egală cu pointer-ul this al clasei respective. Acest lucru setează variabila statică să fie egală cu prima instanță a scriptului din scenă, astfel putem accesa scriptul direct din tipul clasei, fără a fi necesară căutarea acestuia (FindObjectOfType<type>()) sau crearea unei noi instanțe.

În caz că nu există o instanță a clasei în scenă, GamePauseManager.instance va rămâne null, ceea ce poate produce erori, motiv pentru care trebuie să ne asigurăm că adăugăm o instanță a clasei în fiecare scenă. Tot în metoda Awake, trebuie să verificăm și dacă instance este diferit de null, caz care se întâmplă când avem mai mult de o singură instanță a scriptului în scenă. Acest caz poate produce erori, motiv pentru care trebuie afișat un warning și distrusă instanța respectivă a scriptului (se șterg doar intantele scripturilor duplicate până în punctul în care rămânem cu o singură instanță a scriptului în scenă).

Pe lângă asta, scriptul definește câteva proprietăți publice care pot fi accesate direct de alte scripturi datorită singletonului implementat. Aceste proprietăți sunt următoarele:

- **GamePaused** boolean privat care specifică dacă în clipa curentă jocul este pus pe pauză sau nu, și care are implementat doar un getter ca să nu poată fi modificat de alte scripturi.
- **PausedTime** float privat care reprezintă cât timp a fost pus pe pauză jocul. Este util pentru a reactualiza temporizatorii sistemelor. De asemenea, are doar un getter implementat pentru a nu permite altor scripturi să modifice proprietatea.
- **PauseStartTime** float privat care reprezintă timpul la care a fost pus pe pauză jocul. Este util pentru scripturile care se folosesc de Coroutine ca să își poată calcula corect timpul care trebuie așteptat.
- **EventOnPauseGame** este un delegate definit special pentru a înștiința alte scripturi când jocul este pus pe pauză. Toate scripturile care

doreșc, se pot abona la acest eveniment, iar în clipa în care este pus pe pauză jocul își pot opri/modifica sistemele care rulează.

- **EventOnResumeGame** este de asemenea un delegate definit pentru a înștiința scripturile când jocul se reia. Scripturile care doreșc se pot abona la acest eveniment pentru a își reactualiza stările și a reporni sistemele în clipa în care jocul revine la normal.

Clasa are și câteva metode definite pentru a procesa interacțiunea utilizatorului cu UI-ul, deoarece UI-ul de pauză de joc rămâne neschimbăt între scenele jocului. Când jocul este pus pe pauză, pe ecran apare un mesaj care înștiințează jucătorul, și mai apar 2 butoane: "Exit Level" și "Quit Game". În caz că apasă pe vreunul din ele, este întrebat dacă este sigur de alegerea dorită și în caz că răspunde afirmativ, este scos din nivel/joc în funcție de caz. Pentru a relua jocul, trebuie să apese din nou pe butonul de pauză, meniul va dispărea și jocul se va relua.

#### 4.1.10 Valul de inamici

Pe parcursul nivelului, inamicii vin în mai multe valuri. Un val de inamici constituie o serie de inamici diferiți sau identici care apar într-o ordine prestabilită sau la întâmplare. Metoda de implementare care mi s-a părut cea mai ușoară a fost să definesc fișiere de configurare care conțin structura valurilor de inamici. Aceste fișiere configurabile le-am definit folosind un concept specific Unity, și anume ScriptableObject. Aceste obiecte sunt instanțe ale unei clase a cărei structuri o putem defini. Obiectele sunt obiecte globale care pot fi modificate direct din editor și care permit accesarea valorilor din interiorul obiectului cu ușurință, de către scripturile care au nevoie de această informație. Utilizând acest concept, am creat un Scriptable Object numit WaveScriptable, care conține următoarele date:

- SpawnRandomly este un boolean care specifică dacă inamicii apar în ordinea prestabilită sau la întâmplare.
- ID-ul punctului de pe hartă din care vor apărea.
- O listă de perechi de câte două valori. Prima valoare din pereche reprezintă inamicul care va apărea pe hartă, iar a două valoare reprezintă căți inamici de acest tip vor apărea unul după altul, în caz că spawnRandomly este fals, sau aleatoriu în caz contrar.

Folosind această structură, putem crea o multitudine de valuri de inamici, numărul de inamici și ordinea acestora fiind diferită.

#### 4.1.11 Stagiile jocului

Valurile de inamici nu reprezintă suficientă informație pentru a acoperi un nivel, deoarece majoritatea nivelelor nu vor conține un singur val. În acest scop am creat o structură similară cu cea a valului de inamici, dar la un nivel mai înalt.

StageScriptable este un Scriptable Object care definește care valuri de inamici apar la stagiul curent, și multe alte proprietăți pentru a face nivelele cât mai variate.

- StageName reprezintă numele pe care îl definim noi pentru stagiul respectiv. Este folosit de multe sisteme, și anume: încărcarea configurării hărții de joc pe baza căutării unui fișier cu numele specificat de stageName, crearea UI-ului dinamic în scenă în care selectăm nivelul pe care dorim să îl jucăm, lucru care va fi discutat în capitolul 4.2
- O listă de tipul WaveScriptable, în care putem pune toate obiectele create pentru valurile de inamici. Astfel putem crea o serie de nivele variante. O parte din valurile de inamici se pot repeta între stagii, dar stagii trebuie să fie unice.
- IsBossStage este un boolean care ne ajută să identificăm dacă suntem în modul co-op sau nu. Mai multe detalii vor fi discutate în capitolul 4.5
- Banii pe care îi primim dacă reușim să câștigăm nivelul. Banii sunt folosiți pentru sistemul de îmbunătățiri permanente, discutat în capitolul 4.3
- Numărul de stele câștigate până acum la stagiul respectiv. Acest concept va fi discutat în capitolul următor.

#### 4.1.12 Controlul jocului

În punctul acesta am reușit să creez fișiere configurabile pentru o multitudine de nivele diferite, dar este necesară definirea unui sistem care poate citi aceste fișiere configurabile și care controlează ordinea în care apar inamicii și stările jocului.

WaveManager este script-ul care citește propriu-zis fișierul configurabil (StageScriptable) și care controlează modul în care apar inamicii pe hartă. Acesta definește o cortină care primește ca parametru fișierul configurabil al

valului de inamici (preluat din lista de valuri din fișierul stagiului). Corutina ne ajută să așteptăm o perioadă anume între aparițiile inamicilor pe hartă. La începutul corutinei, toți inamicii sunt adăugați într-o lista în ordinea în care sunt specificați în fișierul configurabil. În caz că spawnRandomly este adevărat, reordonăm lista într-un mod aleator folosindu-ne de metoda Shuffle(). Pe urmă, pornim un loop peste fiecare element din lista. La fiecare element creăm un obiect de tipul inamicului respectiv, îl pozitionăm pe hartă în locul dorit, pornim procesul de initializare al acestuia și așteptăm un anumit număr de secunde până la următoarea interacție.

După ce toți inamicii au fost astfel instantiați, se așteaptă o perioadă de 30 de secunde, după care se trece la valul următor. În acest timp jucătorul este liber să își refacă sau îmbunătățească turnurile, sau, în caz că se simte încrăzător, poate să sară peste această perioadă, primind contravaloarea timpului în bani de joc. După ce toate valurile au fost instantiate, controlul este preluat de scriptul BattleStageStateManager, despre care vom discuta în continuare.

BattleStageStateManager este scriptul care controlează finalul nivelului, final obținut prin distrugerea bazei jucătorului sau prin omorârea tuturor inamicilor. În metoda Die() din clasa PlayerBase, discutată în capitolul 4.1.7, apelăm metoda GameOver() din clasa BattleStageManager. Această metodă afișează pe ecran meniul de pierdere a jocului și pune pe pauză jocul pentru a nu permite anumitor sisteme să nu mai funcționeze corespunzător (din cauza lipsei bazei jucătorului) și a nu permite inamicilor să se deplaseze de bună voie pe hartă. La acest meniu de pierdere a jocului afișăm jucătorului că a câștigat 0 stele, afișăm banii câștigați în urma distrugerii inamicilor până în punctul actual și așteptăm ca acesta să apese ecranul, moment în care este trimis la scena de selectare a nivelelor, discutată în următorul capitol.

Celălalt mod în care jocul poate fi terminat este după instanțierea tuturor valurilor de inamici. După ce toate valurile au fost instanțiate, WaveManager încearcă BattleStageStateManager de acest lucru, moment în care acesta verifică o dată la câteva cadre de joc dacă mai sunt inamici în viață pe hartă. În clipa în care nu mai sunt inamici pe hartă, se afișează meniul de câștigare a jocului și se așteaptă ca jucătorul să apese ecranul, moment în care este trimis la scena de selectare a nivelelor, discutată în următorul capitol. Pe meniul de câștigare a jocului se afișează banii și numărul de stele câștigate.

Steile reprezintă cât de bine s-a descurcat jucătorul în a își apăra baza

de operații pe parcursul nivelul. În cazul în care aceasta are mai mult de 90% din viață maximă, jucătorul s-a descurcat excelent și primește 3 stele. În cazul în care baza are peste 40%, atunci jucătorul s-a descurcat la un nivel acceptabil și primește 2 stele. În cazul în care baza are sub 40% atunci a reușit să supraviețuiască nivelul la limita și primește o singură stea drept consecință. Stelele influențează numărul de bani primiti la terminarea nivelului. Fiecare stea câștigată pentru prima dată la nivelul respectiv aduce o recompensă monetară pentru jucător, recompensă care poate fi folosită să își îmbunătățească turnurile.

În caz că jucătorul se blochează la un moment dat și nu poate trece de un anumit nivel, este recomandat să se întoarcă la nivelele anterioare, să câștige 3 stele la toate aceste nivele, să își îmbunătățească turnurile și să se întoarcă mai puternic că niciodată la nivelul care îi provoca dificultăți.

## 4.2 Selectarea nivelelor

### 4.2.1 Structura meniului

La pornirea aplicației putem vedea un meniu cu mai multe opțiuni. Dacă utilizatorul apasă pe butonul "Start Game", va fi dus la un meniu din care poate să selecteze nivelul pe care dorește să îl joace.

În partea stângă putem vedea un buton care ne duce înapoi la pagina principală. Pe lângă aceasta, în colțul din dreapta sus avem un alt buton intitulat "Shop" care ne va duce la magazinul de îmbunătățiri permanente, care va fi discutat în capitolul 4.3. În colțul din dreapta jos putem observa banii adunați de jucător până în punctul curent. La începutul jocului are 0 bani, iar pe măsură ce câștigă nivele și stele la nivelele respective, numărul de bani va crește.

Esența acestui meniu este panoul din centrul ecranului. În partea dreapta a panoului putem observa o lista de nivele. Fiecare nivel are un titlu anume și 3 stele negre. După ce câștigăm un nivel, stelele câștigate la nivelul respectiv vor apărea în locul stelelor negre pentru a ne înștiința progresul făcut la nivelul respectiv.

Dacă apăsăm pe unul din nivele, meniul din stânga va fi umplut cu informații. Prima informație pe care o putem vedea este că apare o imagine cu harta de joc pe care va lua loc nivelul. Sub această imagine sunt scrise informații legate de nivelul respectiv:

- Numele stagiu selectat.

- Inamicii întâlniți și numărul fiecărui tip de inamic.
- Banii primiți în cazul câștigării nivelului. Aceștia sunt banii de baza, pe care îi va primi jucătorul indiferent de numărul de stele câștigate. Stelele aduc o contravaloare în plus față de această sumă, contravaloare plătită numai în clipa în care steaua respectivă este câștigată pentru prima dată. Dacă prima dată când jucăm un nivel câștigăm 2 stele și a două oară când jucăm același nivel câștigăm tot 2 stele, nu vom primi bani extra pe cele 2 stele câștigate. În caz că a două oară când jucăm nivelul primim 3 stele, vom primi contravaloarea unei singure stele, deoarece doar aceasta este steaua care a fost câștigată pentru prima dată.

Putem să mai observăm și un buton intitulat ”Start Stage”, care va începe nivelul respectiv. Acest buton salvează nivelul selectat într-un obiect indestructibil între scene, concept care va fi discutat în capitolul 4.2.3, după care încarcă scena de luptă. Odată ajuns în scena de luptă se preiau datele nivelului încărcat în obiectul indestructibil, se generează automat harta de joc, se initializează toate celelalte scripturi necesare și se așteaptă ca jucătorul să apese pe butonul de începere a jocului.

#### 4.2.2 Generarea dinamică a meniului

Acest meniu este generat automat la începutul acestei scene, în funcție de fișierele configurabile ale stagiilor de luptă create (StageScriptable).

Pentru generarea dinamică a meniului a fost necesară crearea unui prefab ajutător: StageUIBlock. Acest prefab conține butonul pe care poate apăsa jucătorul și 6 stele: 3 dintre stele sunt negre, iar celelalte 3 stele sunt galbene și sunt suprapuse cu cele negre. Când câștigăm un nivel tot ce trebuie să facem este să activăm aceste stele galbene, urmând ca ele să fie randate în fața celor negre. Acest prefab conține și un script care controlează ce nume este afișat pe buton și care stele galbene trebuie să fie activate. Acest lucru se întâmplă în metoda InitializeBlock(StageScriptable stage), care initializează toate proprietățile prefab-ului în funcție de nivelul pe care îl reprezintă. Acest script generează și descrierea nivelului, descriere care conține: numele nivelului, inamicii, numărul de inamici întâlniți din fiecare tip și banii primiți în urma câștigării nivelului.

Pe lângă acest prefab, a fost necesară scrierea unui script care să creeze, poziționeze și initializeze clone ale acestui prefab, în funcție de fișierele de

configurare care au fost definite pentru joc. Scriptul se numește StageUIS-pawner.

Ultima clasa relevantă din această scenă este StageSelectionManagement, care controlează obiectele StageUIBlock selectate și informația afișată de pe acestea. În momentul apăsării unui buton de pe obiectul StageUIBlock, se înștiințează managerul că obiectul respectiv a fost afișat, urmând ca managerul să preia datele necesare de pe obiect și să le afișeze corespunzător pe interfața utilizatorului. Aceasta este responsabil și pentru schimbarea între scene.

#### 4.2.3 Transmisia datelor între scene

Obiectul amintit mai sus se numește SceneDataRetainer. În Unity, în mod normal, când schimbăm între scene, toate obiectele din scena curentă vor fi distruse și vor fi înlocuite cu cele din scena pe care dorim să o încărcăm. Acest lucru include și scripturile de pe obiecte, fapt care face transmisia de date între scene să fie dificilă. Există totuși câteva variante prin care pot fi transmise datele între scene, variante care vor fi discutate în continuare.

Una dintre variantele valide și foarte des folosite de a transmite datele între scene este să salvăm datele pe disk înainte să încărcăm scena, iar după ce am încărcat scena să citim acel fișier, să procesăm datele din acesta și să inițializăm scripturile dorite în funcție de datele procesate. Unity a definit și o clasa specială pentru acest mod de lucru și anume PlayerPrefs. În clasa PlayerPrefs putem salva orice date dorim și putem să le accesăm din alte scene cu ușurință, prin simpla apelare a câtorva metode simple. Clasa PlayerPrefs în spate funcționează tot prin salvarea și citirea anumitor fișiere pe disk, iar metodele expuse de clasă apelează acele funcționalități specifice. Această metodă este utilă când vrem să salvăm date simple între scene, dar nu și când avem obiecte complexe, deoarece de cele mai multe ori ar trebui să creăm din nou obiectul respectiv după ce îl citim din fișier. O altă limitare este că nu pot fi stocate date și fișiere foarte mari între scene, deoarece citirile și scrierile pe disk ar dura foarte mult timp, fapt ce ar întrerupe jocul. Ultima limitare relevantă este că, pe anumite dispozitive, nu avem acces să citim/scriem oriunde dorim noi, fapt ce ar determina PlayerPrefs să nu funcționeze corespunzător pe acele dispozitive. Mai multe lucruri vor fi discutate despre acest subiect în capitolul [4.4.3](#)

Cealaltă metodă de lucru este cea pe care am ales să o folosesc, și anume folosirea unui obiect nemuritor. Pentru a defini un astfel de obiect, în metoda

Awake sau Start din scriptul respectiv, trebuie să apelăm metoda DontDestroyOnLoad() și să îi dăm ca parametru obiectul pe care se află scriptul respectiv. Acest lucru va înștiința Unity-ul că obiectul respectiv nu trebuie să fie șters la încărcarea între scene. Astfel transmisia de date între scene devine foarte ușoară, trebuie să definim variabile publice sau private cu geteri și seteri pentru a permite salvarea și citirea corectă a datelor. SceneDataRetainer de asemenea are un Singleton implementat pentru a permite accesarea datelor între scene cu ușurință. În această clasa am ales să salvez următoarele date:

- SelectedStage este o variabilă de tipul StageScriptable care are definiti getteri și setteri. În această variabilă salvăm nivelul selectat de jucător, pentru a putea inițializa scena de luptă în momentul incarcării acesteia.
- PermanentUpgrades este o variabilă de tipul TurretPermanentUpgrades[] care are de asemenea definiți getteri și setteri. Această variabilă reține coeficientii de îmbunătățire a turnurilor, în urma achiziționării lor din magazinul de îmbunătățiri permanente. Mai multe detalii în capitolul 4.3.
- SelectedBossDifficulty este o variabilă de tipul BossScriptableObject. Această variabilă reține proprietățile monstrului din scena co-op, care va fi discutată în capitolul 4.5.

Acest script generează totuși o problema. În caz că punem scriptul acesta direct pe un obiect din scenă, în prima iteratie va funcționa corespunzător. Vom putea încărca un nivel, să jucăm normal prin acel nivel, iar la finalul lui revenim la scena de selectare a nivelelor jocului. În acest punct vom avea 2 instanțe ale scriptului: unul din scripturi este cel nemuritor pe care l-am păstrat între scene, iar al doilea este cel nou creat când am încărcat scena de selectare a nivelelor. Deoarece el era setat din start în scena, de fiecare dată când reîncărcăm scenă, va fi creată o nouă instanță a scriptului. Acest lucru poate să strice sistemele existente, din acest motiv este necesară scrierea unui alt script.

SceneIndependentScriptsSpawner este un script care la începutul jocului verifică dacă avem scripturi nemuritoare în scenă. În caz că avem, atunci se distrug singur fără să mai facă nimic. În caz că nu avem scripturi nemuritoare, creează obiectele nemuritoare dorite, le initializează, după care se distrug singur pentru a opri orice altă funcționalitate. Acest simplu script rezolvă problema duplicării de obiecte nemuritoare între scene.

## 4.3 Magazinul de îmbunătățiri permanente

### 4.3.1 Structura meniului

Poate fi accesat din meniul selectării nivelului de joc. La acest meniu putem vedea în colțul din dreapta jos, banii pe care i-a adunat jucătorul până în clipa curentă. În stânga sus avem un buton care ne duce înapoi la meniul selectării nivelului de joc. În centrul ecranului putem vedea butoane cu numele turnurilor noastre. În caz că apăsăm pe unul din butoane, în dreapta acestora va apărea o nouă interfață grafică.

Putem observa că au apărut 3 rânduri, fiecare rând conținând o serie de informații:

- Numele proprietății care este îmbunătățită pentru turnul curent.
- Zece dreptunghiuri care pornesc de la mic la mare. Aceste blocuri vor fi umplute cu diverse culori pe măsură ce îmbunătățim proprietatea specifică a turnului selectat.
- Un buton de plus care cumpără un nivel nou pentru proprietatea respectivă. Nivelul maxim care poate fi cumpărat este 10.
- Sub butonul de plus apare costul cumpărării pentru îmbunătățirea respectivă. Acest cost se calculează în funcție de nivelul actual al jucătorului. În caz că nu are suficienți bani, textul este colorat cu roșu, și dacă apasă pe butonul de plus nu se va întâmpla nimic. La nivelul maxim, în loc de un cost, va apărea cuvântul ”MAX” în acest loc.

Îmbunătățirile pentru fiecare tip de turn în parte sunt:

- Gardul electric: rată și putere de atac mărită, viață extra
- Excavatorul: rată și putere de atac mărită, viață extra
- Aruncătorul de flăcări: putere și rază de atac mărită, viață extra
- Mitralieră automată: rată, putere și rază de atac mărită
- Baza jucătorului: rată și putere de atac mărită, viață extra
- Laserul: putere și rază de atac mărită, viață extra
- Vulkan: rată, putere și rază de atac mărită

Toate aceste elemente au fost generate dinamic la încărcarea scenei, fapt pe care îl vom investiga în detaliu în cele ce urmează. Ca să putem discuta despre generarea dinamică a meniului, trebuie mai întâi să parcurgem cum este reprezentată o îmbunătățire permanentă și cum este folosită în alte scripturi.

#### 4.3.2 Îmbunătățire permanentă

Îmbunătățirea permanentă este un simplu coeficient cu care sunt înmulțite toate proprietatile turnurilor la începutul unui nivel de luptă. Totuși, pentru folosirea acestui coeficient a fost necesară folosirea mai multor informații, motiv pentru care am creat o clasa specială, PermanentUpgrade, care are următoarele proprietăți:

- UpgradeType este un enum care definește tipul îmbunătățirii. Acesta poate fi una din următoarele: viteza, putere, rază de atac și viață extra.
- Valori de minim și maxim între care se poate afla coeficientul îmbunătățirii respective. Acest lucru ne ajută să calculăm cu ușurință coeficientii respective la fiecare nivel al îmbunătățirii.
- Coeficientul propriu-zis cu care înmulțim proprietățile turnurilor.
- Costul de start pentru a cumpăra prima îmbunătățire.
- Un coeficient cu care este înmulțit prețul curent al îmbunătățirii, în momentul cumpărării acesteia. Rezultatul operației este salvat drept costul pentru următoarea îmbunătățire
- Nivelul curent al îmbunătățirii.
- Nivelul maxim la care poate fi dusă îmbunătățirea. Momentan a fost setat la 10.

Cu această clasa am reușit să definim cum ar trebui să arate o îmbunătățire, dar un turn are 3 îmbunătățiri, iar de preferat ar fi să putem modifica cu ușurință aceste valori. Din acest motiv am creat o nouă clasa care moștenește ScriptableObject și care definește o serie de proprietăți:

- Tipul turnului la care se aplică îmbunătățirea.
- O lista de îmbunătățiri posibile pentru turnul respectiv. Lista conține elemente de tipul PermanentUpgrade, care a fost explicat mai sus.

- O metodă prin care putem obține coeficientul. Aceasta primește ca și parametru tipul de îmbunătățire dorit. Funcția caută în lista de îmbunătățiri dacă există acea îmbunătățire dorită, și dacă există, atunci returnează coeficientul acestieia. În caz contrar returnează 1 pentru a nu modifică proprietățile turnurilor.

#### 4.3.3 Generarea dinamică a meniului

Pentru generarea dinamică a meniului am creat un prefab cu un buton și un text pe el, care reprezintă butoanele cu numele turnurilor. Aceste butoane vor fi generate automat folosind prefab-ul respectiv. Pe fiecare din aceste butoane se adaugă un delegate care va fi apelat în momentul apăsării pe buton. Acest delegate schimbă index-ul turnului selectat și actualizează informația de pe ecran în funcție de noul index.

O altă clasa folosită este UIUpgradeRow, care controlează afișarea și cumpărarea îmbunătățirii permanente (rândul din dreapta care apare în momentul selectării turnului pe care dorim să îl îmbunătățim). Acest script este activat și inițializat în momentul în care apăsăm pe unul din butoanele explicate anterior, moment în care activează dreptunghiurile de pe ecran în funcție de nivelul actual al îmbunătățirii. Pe lângă asta, calculează și costul cumpărării următorului nivel pt îmbunătățirea respectivă, afișează acest cost pe ecran, iar în cazul în care jucătorul nu are suficienți bani, colorează cu roșu acest cost. De asemenea, în momentul în care jucătorul cumpără o îmbunătățire, se apelează o metodă din această clasa, care calculează din nou costul proprietății, îl afișează pe ecran, și își întrează managerul magazinului că jucătorul dorește să cumpere îmbunătățirea respectivă. În acea clipă, managerul magazinului scade banii pe care ii deține jucătorul, actualizează coeficientul și datele îmbunătățirii și pornește salvarea datelor, care va fi discutată în capitolul următor.

Deoarece există 3 îmbunătățiri pentru fiecare turn, UIUpgradeRow se va afla pe fiecare din cele 3 rânduri, diferența dintre ele fiind proprietatea pe care o monitorizează. Această proprietate este setată în momentul în care apăsăm pe butonul cu numele turnurilor, UI-ul modificandu-se în funcție de proprietățile turnului selectat. La fiecare îmbunătățire, datele sunt salvate automat și în memorie dar și pe disk, astfel nefiind posibilă pierderea datelor.

## 4.4 Salvarea datelor

Salvarea datelor este un concept întâlnit în toate jocurile existente, dar și în alte tipuri de aplicații. Orice joc are nevoie să salveze anumite date: date cu progresul jucătorului, date cu configurații ale jucătorului, date colectate de dezvoltatori pentru a își îmbunătății jocul/aplicația, etc. Deoarece este un concept atât de important și am dorit ca jucătorul să aibă parte de o experiență cât mai plăcută, a trebuit să implementez un astfel de sistem și pentru joc.

Fișierele de configurație necesare pentru ca jocul să funcționeze corespunzător au fost discutate la fiecare capitol în parte unde erau necesare, motiv pentru care nu vor fi reluate aici.

### 4.4.1 Datele care trebuie salvate

În primul rând trebuie să discutăm despre ce date este necesar să fie salvate pentru ca jucătorul să poată continua de unde a rămas, în urmă întreruperii sesiunii de joc. În acest scop, am definit o clasa PlayerData a cărei structură definește tipurile de date care trebuie salvate pentru un anumit jucător. Clasa conține banii câștigați de jucător și o lista cu progresul acestuia la fiecare nivel al jocului.

Informația progresului pentru fiecare nivel am organizat-o într-o clasa separată pentru a ușura modul de lucru. Clasa se numește StageSaveData și conține două proprietăți: numele nivelului, pentru a putea fi identificat cu ușurință, și numărul de stele câștigate la nivelul respectiv.

Pe lângă aceste date, pentru un anumit jucător salvăm și îmbunătățirile cumpărate de jucătorul respectiv. Îmbunătățirile au structura prezentată în capitolul 4.3.

### 4.4.2 Salvarea automată

Salvarea propriu-zisă este controlată de un script nemuritor, definit cu tipul PlayerDataSaver. Acest script este creat și inițializat în același timp cu SceneDataRetainer, și are implementat și un singleton pentru a permite scripturilor din scenă să îl acceseze cu ușurință.

Scriptul are definit o metodă SaveData(), care preia toate datele utilizatorului, discutate mai sus și le convertește la o structură JSON, pentru a ușura procesarea datelor. După convertire, aplică o criptare simplistă peste

aceste date, pentru a nu permite jucătorului să modifice datele după bunul lui plac. Metoda de criptare aplică un operator XOR între fiecare caracter din fișierele JSON și o cheie definită în program. Această operație face ca mesajul JSON să fie inteligibil pentru oameni. După ce a terminat criptarea datelor, salvează mesajele rezultate în 2 fișiere: primul fișier conține datele jucătorului (banii și progresul la fiecare nivel în parte), iar al doilea fișier conține îmbunătățirile cumpărate de acesta.

Deoarece salvăm datele pe disk, trebuie să avem definită și o metodă pentru a prelua aceste date. Metoda este denumită simplist: LoadData și, folosind tratarea de excepții, încearcă să citească și să proceseze datele de pe disk. Metoda este împărțită în două faze și aplică tratarea de excepții în fiecare dintre faze.

Prima fază este cea de citire a datelor jucătorului (primul fișier salvat). În această situație, citește conținutul fișierului de pe disk, aplică din nou aceeași operație de criptare, care va readuce textul la starea lui inițială, urmând să îl convertească la un obiect de tipul PlayerData. Motivul principal pentru care am ales să folosesc formatul JSON, este pentru că există multe funcționalități deja implementate pentru serializare și deserializare de fișiere JSON. Serializarea presupune convertirea unui obiect din memorie la un string cu formatul specific JSON, pe când deserializarea presupune convertirea unui string în formatul JSON la un obiect în memorie. Aplicând procesul de deserializare pe string-ul rezultat în urma decriptării, obținem în memorie un obiect de tipul PlayerData.

Folosindu-ne de acest obiect, inițializăm toate celelalte date relevante din joc. În cazul în care fișierul lipsește sau este corupt, se resetează toate stările de joc la o stare de baza, în care jucătorul nu are progres la vreun nivel, nu deține bani în posesie și nu a cumpărat nici o îmbunătățire permanentă.

A două fază este cea de citire a îmbunătățirilor cumpărate de jucător. Acestea erau salvate într-un fișier separat, iar procesul este similar cu cel de mai sus. Citim fișierul de pe disk, îl decriptăm aplicând aceeași operație XOR cu aceeași cheie, deserializam obiectul, astfel obținând un obiect de tipul `TurretPermanentUpgrades[]`, urmând să modificăm datele jocului în funcție de aceste date citite de pe disk.

La fel ca și în cazul primei faze, în caz că lipsește fișierul de pe disk, sau fișierul respectiv a fost corupt, jocul se resetează la starea lui inițială. Fișierul poate deveni corupt din o serie de motive:

1. Programatorul a schimbat cheia folosită pentru decriptare între rulări ale aplicăției.
2. Jucătorul a găsit fișierele salvate pe disk și a încercat să le modifice.
3. Jucătorul a închis aplicația brusc cât timp se afla în procesul de salvare a datelor, rezultând într-un fișier incomplet. Acest caz este puțin probabil, deoarece datele care sunt salvate sunt puține, și salvarea se face numai în momentul în care s-a terminat de procesat toate datele. Dacă salvarea pe disk se aplică treptat, atunci era cu mult mai probabilă coruperea fișierului.

Cu ajutorul acestor operații, putem concepe cu ușurință un sistem de salvare automată, și anume: de fiecare dată când jucătorul modifică o parte din date (câștigă un nivel, cumpără o îmbunătățire), apelăm metoda SaveData din acest script, care va suprascrie datele jucătorului cu cele curente.

#### **4.4.3 Locația fișierelor salvate pentru dispozitive diferite**

Unity pune la dispoziție două metode utile pentru a controla locațiile în care se salvează datele, și anume:

- Application.streamingAssetsPath. Această proprietate returnează o locație din folderul jocului, în care sunt salvate toate resursele plasate în folderul StreamingAssets în editor.
- Application.persistentDataPath. Această proprietate returnează o locație în afara folderului jocului și diferă în funcție de platforma de pe care se rulează. Pe Windows, acest folder poate fi găsit de obicei în AppData/LocalLow/<companyname>/<productname>. Pe android, de obicei se află la locația /storage/emulated/0/Android/dată/<packagename>/files.

În cazul rulării aplicăției pe Windows, putem alege ambele variante, singura diferență este dacă dorim să păstrăm datele după ce stergem jocul sau nu. În caz că nu vrem să păstrăm datele, trebuie să folosim Application.streamingAssetsPath, iar în caz că vrem să fie păstrate, trebuie să folosim Application.persistentDataPath.

Pe Android este o cu totul altă poveste. Toate fișierele jocului sunt compresate într-un singur fișier .apk, motiv pentru care nu putem scrie în acest fișier. Putem doar să aplicăm citiri, motiv pentru care, pe android trebuie să folosim Application.persistentDataPath, care ne dă o locație în care putem

aplica și citiri și scrieri.

Din această cauză, am creat o clasă specială, FileManager, care controlează locațiile la care se aplică citiri și scrieri de fișiere. Acest script funcționează diferit în funcție de platformă pe care este rulat, datorită folosirii unor directive specifice, și are 2 metode definite: SetFileContents și GetFileContents. Ambele metode primesc ca și parametrii un boolean care definește dacă vrem să salvăm în streaming assets sau nu, în caz contrar se salvează la locația persistență. Al doilea parametru reprezintă string-ul pe care vrem să îl punem în fișier, iar ultimul parametru reprezintă numele fișierului în care salvăm sau din care citim conținutul.

## 4.5 Sistemul co-op

Pentru a putea diferenția jocul acesta față de alte jocuri pe acest stil din domeniu, am decis să implementez un mod co-op, în care joci un nivel dificil împreună cu un alt jucător. Momentan este implementat un singur nivel de acest tip, dar care are mai multe dificultăți care pot fi explorate. Pe lângă aceasta, majoritatea funcționalităților sunt deja scrise, motiv pentru care pot fi create cu o relativă ușurință și alte nivele de joc.

Sincronizarea a fost realizată cu ajutorul un framework numit Photon Engine, despre care vom vorbi în continuare.

### 4.5.1 Modul de lucru cu Photon Engine

Photon Engine este un framework dezvoltat de Exit Games. Acest framework are mai multe pachete care pot fi achiziționate. O parte dintre acestea sunt gratis, iar altele sunt plătite, dar oferă funcționalități extra față de cele gratis. Pentru acest proiect am folosit pachetul PUN 2 care este gratis. Acest pachet oferă un serviciu de găzduire a server-ului care suportă până la un maxim de 20 de utilizatori conectați la aplicație în același timp.

Pe lângă acest serviciu, pachetul oferă câteva clase și funcționalități utile în Unity care ajută la procesul de conectare la server, comunicare între dispozitive și sincronizare de date. În continuare vom discuta despre fiecare din aceste categorii în parte.

## Conecțarea la server

Pentru a putea crea un server de găzduire pentru aplicația noastră, trebuie să intrăm pe site-ul oficial de la Photon Cloud și să creăm o aplicație de tipul Photon PUN. În urma creării acestei aplicații în cloud, ne generează un id unic pentru aplicație, care este recomandat să fie păstrat confidențial. Pasul următor este să importăm pachetul PUN 2 - FREE într-un proiect de Unity. În urma importării fișierelor, apare o casuță de dialog în care trebuie să introducem id-ul generat anterior pentru aplicația noastră din cloud.

Setările serverului pot fi accesate din Unity din locația Window->Photon Unity Network->Highlight Server Settings. Aici putem vedea o serie de opțiuni:

- ID-ul aplicației care a fost setat anterior și care poate fi modificat.
- Versiunea aplicației. Aplicații cu versiuni diferite nu se vor putea conecta.
- Tipul de protocol care va fi folosit în transmiterea datelor.
- O multitudine de opțiuni folosite pentru a depăși aplicația.

În urma setării acestor opțiuni, pentru a ne conecta la server în joc, tot ce trebuie să facem este să apelăm metoda PhotonNetwork.ConnectUsingSettings. Clasa PhotonNetwork este o clasa cu foarte multe metode și proprietăți statice care sunt utile în a verifica starea în care se află conexiunea cu serverul, dar și starea curentă a utilizatorului (dacă se află într-o camera de așteptare, camera de joc sau este conectat direct la server, etc.).

## Comunicarea între dispozitive

Comunicarea între dispozitive poate fi realizată cu ușurință datorită unei clase speciale, numită Photon View. Această clasa este responsabilă de a identifica instanțele obiectelor între clienții aplicației (același obiect între clienți are același id care este folosit pentru a transmite și sincroniza datele). Photon definește și o serie de clase generale pentru a sincroniza proprietăți specifice Unity, dar acestea vor fi discutate amănunțit în capitolul [4.5.3](#).

Clasa Photon View știe să identifice automat care scripturi/proprietăți trebuie sincronizate în rețea, de pe obiectul pe care a fost adăugat scriptul, dar și în copii acestuia. În cazul în care vrem să marcăm un script scris de noi

să fie sincronizat în rețea, clasa respectivă trebuie să implementeze interfața IPunObservable. Această interfață pune la dispoziție metoda OnPhotonSerializeView() care se ocupă de sincronizarea între clienți.

Metoda OnPhotonSerializeView() primește un parametru de tipul PhotonStream care are 2 proprietăți utile: IsWriting și IsReading. Scriptul Photon View identifică și cine este deținătorul obiectului, iar în cazul în care utilizatorul curent este deținătorul, IsWriting va fi adevărat, iar IsReading va fi fals. În cazul în care nu deținem obiectul respectiv, IsWriting va fi fals, iar IsReading va fi adevărat.

Astfel putem separa cu ușurință logică care trebuie să fie executată de deținătorul obiectului de logică executată de ceilalți utilizatori. Pe ramura IsWriting putem să transmitem mesajele dorite de îndată ce acestea se întâmplă, lucru realizat prin metoda stream.SendNext(objectToSend). Pe ramura IsReading, putem primi datele folosind metoda stream.ReceiveNext(), urmând să procesăm și să actualizăm stările obiectului în funcție de mesajul primit de la deținător. Singurul aspect la care trebuie să avem grijă este că trebuie să primim exact atâtea mesaje câte au fost transmise de la deținător. În caz că avem mai puține, citirea va produce o eroare, iar în caz că avem mai multe, mesajele se vor pierde și stările de joc nu vor mai fi sincronizate.

Pe lângă această funcționalitate, Photon pune la dispoziție și două clase care extind MonoBehaviour, și anume MonoBehaviourPun și MonoBehaviourPunCallbacks. MonoBehaviourPun pune la dispoziție proprietatea photonView pentru a ușura accesul acestiei, iar MonoBehaviourPunCallbacks este puțin mai complexă:

- OnConnected() este apelat când conexiunea a fost stabilită pentru prima dată, înainte de a fi pregătit de a realiza comunicări cu serverul.
- OnConnectedToMaster() este apelat când clientul se conectează la server și este pregătit să între într-o cameră de așteptare sau direct într-o cameră de joc.
- OnLeftRoom() este apelată când jucătorul curent părăsește camera de joc. Este utilă în a transmite un ultim mesaj celorlalți jucători sau de a reseta anumite configurații ale camerei înainte de a părăsi jocul.
- OnJoinRoomFailed() este apelată când jucătorul nu a putut intra într-o cameră de joc din diverse motive.

- OnJoinedLobby() este apelată când jucătorul s-a conectat cu succes la o cameră de aşteptare. Logica din camera de aşteptare este diferită în funcție de aplicație, dar în general, este un loc unde se adună jucătorii înainte să înceapă jocul propriu-zis. Când toți jucătorii sunt gata, jucătorul care a creat camera, încarcă camera de joc pentru toți jucătorii din camera de aşteptare.
- OnCreatedRoom() este apelată când jucătorul a reușit să creeze o cameră de joc cu succes.
- OnDisconnected() este apelată când jucătorul a fost deconectat de la server. Acest lucru se întâmplă când are conexiunea foarte proastă, a opri fortat sau a pus pe pauză execuția aplicației ( acest lucru se întâmplă și când se încearcă depanarea aplicației cu puncte de intrerupere definite în Visual Studio), și o serie de alte situații posibile.
- OnPlayerEnteredRoom() este apelată când un jucător a reușit să se conecteze cu succes la camera de joc. Această metodă este utilă pentru a crea și inițializa reprezentarea grafică a jucătorilor pe măsură ce acestia intră în camera de joc.
- OnPlayerLeftRoom() este apelată când un jucător a părăsit camera de joc. Această metodă este utilă pentru a șterge reprezentarea grafică a jucătorului și obiectele deținute de acesta în momentul în care părăsește jocul. Poate fi folosită și pentru a reactualiza stările jocului în urma părăsirii unui jucător (un joc care avea neapărată nevoie de un alt jucător poate să instanțieze un bot controlat de calculator care să ajute jucătorul încă aflat în joc).

Photon pune la dispoziție multe alte funcționalități utile, dar acestea prezentate sunt cele mai des folosite în proiect.

#### 4.5.2 Înregistrare și camera de aşteptare

Pentru a începe experiența co-op, trebuie mai întâi să ne înregistram. Procesul de înregistrare este simplu, trebuie doar să ne introducem numele de joc pe care îl dorim. Inițial este generat automat un nume pe care putem să îl păstrăm sau să îl modificăm în funcție de preferință. În momentul apăsării butonului de login, salvăm în memorie numele ales de jucător. Nu este necesară salvarea numelui într-o baza de date sau în cloud deoarece datele cu progresul jucătorului sunt deja salvate pe disk. Acum nume este util doar să facă diferență între jucători cât timp joacă un nivel anume.

După înregistrare apar 3 opțiuni:

- Create Room. Această opțiune ne duce la o nouă pagină unde putem specifica numele camerei pe care o dorim. Crearea camerei de așteptare se realizează cu ajutorul apelării metodei PhotonNetwork.CreateRoom( roomName, roomOptions ). Opțiunile specificate în al doilea parametru sunt prestatibile și nu pot fi modificate de jucător. Cea mai importantă astfel de opțiune este limitarea numărului maxim de jucători. Deoarece este o experiență co-op, maxim doi jucători pot fi conectați la aceeași cameră și pot juca un nivel împreună. În funcție de aplicație acest număr poate fi modificat.
- Join Random Room. Această opțiune apelează metoda PhotonNetwork.JoinRandomRoom(), care în spate funcționează în modul următor: caută să vadă dacă există camere deja create și dacă există vreo cameră care să aibă un loc liber. În caz că există o astfel de cameră, se conectează la aceasta, iar în caz contrar, creează o cameră nouă cu aceleasi setări specificate mai sus. Această metodă este utilă când nu dorim să pierdem timpul creând camere sau căutând o cameră specifică.
- Show Room List. Această metodă afișează toate camerele de așteptare care există pe server, numele și numărul de locuri ocupate și disponibile la acestea. Căutarea camerelor de pe server se face cu ajutorul unei funcții definite în MonoBehaviourPunCallbacks, și anume OnRoomListUpdate(), care ne dă drept parametru o listă cu toate camerele existente pe server.

După ce jucătorul a intrat în cameră, de jos poate să selecteze dificultatea nivelului, iar în dreapta numelui lui are un buton "Ready?". În momentul în care un alt jucător se conectează la aceeași cameră, va apărea în această listă, iar în clipa când amândoi jucătorii au apăsat butonul "Ready?", în dreapta jos apare un buton care pornește jocul, buton care este vizibil doar pentru jucătorul care a creat camera.

Încărcarea nivelului se realizează folosind metoda PhotonNetwork.LoadLevel( levelName ), care încarcă numele specificat ca parametru pentru toți jucătorii din camera de așteptare (în cazul de față cei doi jucători). Pentru a putea fi încărcat nivelul, acesta trebuie să fie adăugat în setările de distribuire a aplicației File->Build settings.

#### 4.5.3 Sincronizarea datelor

Pentru a sincroniza corect datele de joc a trebuit să adaug o clasă numită NetworkPlayer și să modific o mare parte a scripturilor pentru a își sincroniza reciproc datele.

Network player este o clasă care este generată la conectarea fiecărui jucător în scenă. Aceasta conține o metodă SendNetworkEvent( eventname, eventParam ) care va trimite un eveniment către celălalt jucător, iar jucătorul care recepționează mesajul va actualiza anumite sisteme la recepționarea acestui eveniment. Această funcționalitate este necesară pentru obiectele scenice.

Descrierea de acum două capitole legată de sincronizarea datelor folosind Photon View funcționează doar pentru obiectele care sunt generate pe parcursul rulării aplicației. Obiectele care nu sunt generate ci sunt așezate în scenă înainte de începerea aplicației se numesc obiecte scenice. Acestea nu își pot sincroniza datele atât de ușor deoarece nu există un deținător ale acestor obiecte, motiv pentru care valorile metodelor din PhotonStream.IsWriting și PhotonStream.IsReading rămân întotdeauna false. O alternativă pentru a sincroniza aceste obiecte este să se folosească RPC-uri, dar acestea ar fi complicat destul de mult logica aplicației.

Din acest motiv, am ales să creez clasa Network Player, care poate primi evenimente de la orice obiect scenic, evenimente pe care le trimit în rețea, iar în momentul recepționării acestor mesaje, actualizează obiectele scenice ale clientului conform evenimentelor primite. Metodele principale care au fost transmise până în clipa actuală sunt următoarele: evenimentul de câștigare a jocului, evenimentul de punere pe pauză a jocului și cel de a determina comandanțul să părăsească turnul pe care îl controlează.

Pe lângă această clasă, majoritatea claselor deja existente au trebuit să fie extinse pentru a suporta sincronizarea datelor. Majoritatea claselor au trebuit să implementeze IPunObservable și să definească în metoda OnPhotonSerializeView() modul exact de sincronizare a datelor.

O parte dintre clase au trebuit să moștenească clasa IPunInstantiateMagicCallback, care adaugă funcționalitatea de a trimit date de initializare în momentul generării obiectelor. Un exemplu de o astfel de folosință este pentru turnurile de joc. În clipa în care construim un turn, trebuie să îi transmitem turnului, blocul hexagonal pe care a fost construit. La deținătorul

camerei de joc nu avem probleme cu acest proces, dar pe client, turnul va fi creat în urma apelării PhotonNetwork.Instantiate(), care generează doar o copie a obiectului pe server. Această copie, cum nu este creată de aceeași logică de construire cu care a fost construit pe partea deținătorului, nu va avea acces la blocul hexagonal pe care a fost construit. Din acest motiv, IPunInstantiateMagicCallbacks ne pune la dispoziție opțiunea de a transmite anumite date de initializare, copiei care va fi creată pe partea clientului. Pe partea clientului va fi apelată metoda OnPhotonInstantiate() care primește datele transmise și care poate inițializa corespunzător obiectul.

Photon pune la dispoziție și câteva clase care pot sincroniza tipurile de date specifice Unity, cum ar fi Transform și Rigidbody.

În cazul sincronizării proprietăților Transform, Photon a creat clasa PhotonTransformView, în care putem specifica direct din editor care dintre componente dorim să le sincronizăm (poziția, rotația sau scala obiectului).

În cazul sincronizării proprietăților Rigidbody, Photon a creat clasa PhotonRigidbodyView, în care putem specifica dacă dorim să sincronizăm velocitatea obiectului, velocitatea unghiulară, și dacă permitem teleportarea obiectelor în cazul în care conexiunea între jucători este proastă.

Acstea clase pot fi detectate automat de Photon View, și sunt doar o mică parte din ce pune la dispoziție Photon. Noble Whale Studios a creat un pachet foarte util pentru Photon, care permite sincronizarea cu mult mai bună a obiectelor. Acest pachet permite calcularea poziției obiectelor pe parcursul mai multor cadre de joc, astfel scăpând de efectul în care obiectele se teleportează în cazul conexiunii proaste. Din nefericire, acest pachet nu este gratis și nu am avut ocazia să îl folosesc la acest proiect.

#### 4.5.4 Monstrul de foc

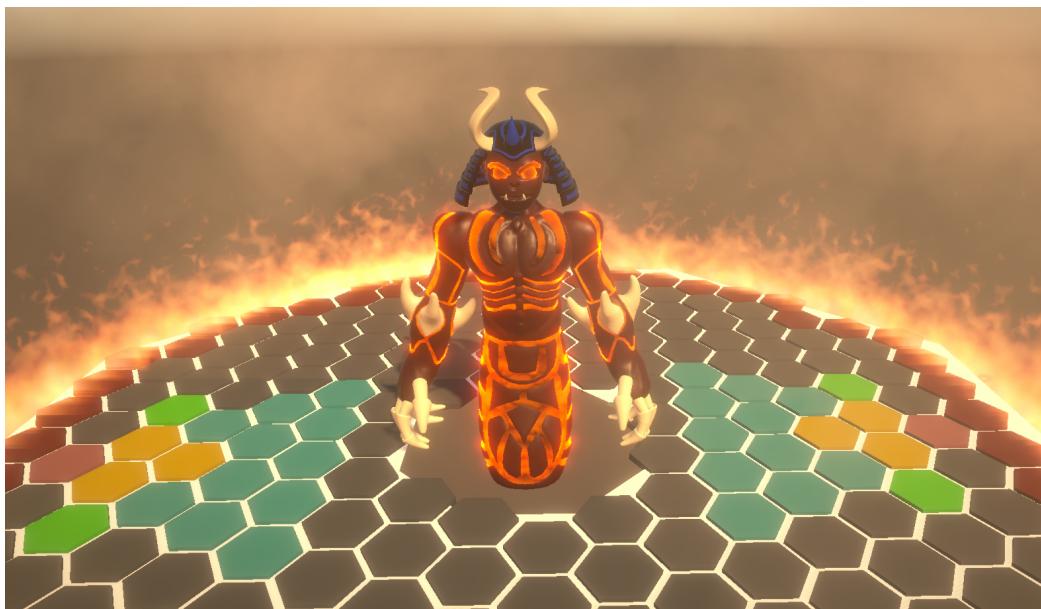


Figure 16: Monstrul de foc

La încărcarea scenei de luptă, în mijlocul hărții apare un monstru imens de foc. Scopul nivelului este să îngingem monstrul împreună cu coechipierul nostru, înainte ca acesta să ne distrugă baza.

Acest monstru, o dată la un anumit interval de timp alege să execute o acțiune din acțiunile lui deja stabilite. Acțiunile posibile ale acestui monstru sunt următoarele:

1. **Monstrul începe să se întindă.** Este o acțiune care ușurează puțin nivelul deoarece nu atacă jucătorul. Acesta doar arată o animație cum se întinde puțin pe hartă și îi permite jucătorului să se refacă după atacurile anterioare sau să atace monstrul cu tot ce are mai puternic.
2. **Distrugătorul de turnuri.** Pentru această acțiune, monstrul de foc ridica mâna dreaptă sus și începe să creeze meteoriți de foc. Acești meteoriți cresc în mărime până în punctul în care ajung la o dimensiune cât un turn obișnuit. În acest moment, monstrul de foc alege la întâmplare un turn de pe hartă, trimitete meteoritul către acest turn, după care începe să încarce următorul meteorit. În caz că nu găsește nici un turn pe hartă, atunci va ataca direct una din bazele jucătorilor, altfel, atât timp cât există cel puțin un turn pe hartă, nu poate să

atace bazele jucătorilor. Numărul de meteoriți folosiți diferă în funcție de dificultatea aleasă: 3 meteoriți în modul ușor, 5 meteoriți pe mediu și 7 meteoriți pe cea mai mare dificultate.

3. **Învârtirea mortală.** Monstrul de foc își scoate ghearele și începe să se învârtă 360 de grade și să lovească toate turnurile de pe hartă, inclusiv bazele jucătorilor. Pentru a diminua daunele primite, jucătorul poate să construiască garduri electrice pentru a proteja toate turnurile din spatele acestora. În caz că monstrul de foc întâlnește garduri electrice, va scădea jumate din viață care ar fi fost scăzută în mod normal. Această diminuare se aplică doar turnurilor care se află fix în spatele gardurilor electrice, cele care se află în lateralul acestora nu primesc acest beneficiu.
4. **Meteoritul suprem.** Acest atac nu poate fi ales în mod voit de către monstrul de foc. Este activat automat când monstrul de foc ajunge la 2/3 și 1/3 din viață lui maximă. Pentru acest atac, alege la întâmplare una din bazele jucătorilor, se învârte către aceasta și începe să încarce un meteorit uriaș deasupra capului. Pe ecran apare un cronometru, iar jucătorii au timp 30 de secunde ca să îi provoace suficiente daune monstrului. În caz că nu reușesc acest lucru, monstrul va lansa meteoritul uriaș (care este cât un sfert din hartă în punctul acesta) către baza jucătorului, lovind toate turnurile din cale. În cazul în care jucătorii reușesc să îl atace suficient de mult în acest timp, monstrul devine amețit timp de 10 secunde, timp în care jucătorii pot să îl atace sau să își refacă turnurile după bunul plac.

Pe lângă aceste atacuri, monstrul are și o abilitate automată: cu cât îi scade mai mult viață, cu atât devine mai puternic. Când este aproape de punctul de a muri, va fi de 1.5 ori mai puternic decât era la începutul nivelului.

Acest monstru are anumite proprietăți definite într-un fișier configurabil, care permite crearea cu ușurință a dificultăților multiple. Acest fișier configurabil reprezintă o clasa de tipul BossScriptableObject, care moștenește clasa ScriptableObject. Această clasa definește următoarele proprietăți: viață maximă, timpul dintre decizii, banii primiți de jucător pentru fiecare punct de viață luat, puterea de atac pentru fiecare dintre cele trei atacuri ( distrugătorul de turnuri, învârtirea mortală, meteoritul suprem ), numărul de meteoriți pentru atacul "distrugătorul de turnuri" și procentajul de viață necesar pentru a deveni amețit în momentul când execută atacul "meteoritul suprem".

Datorită acestei structuri, am definit 3 fișiere configurabile, câte unul pentru fiecare dificultate și am adăugat opțiunea de a permite jucătorului să aleagă pe care dintre ele le dorește la începerea jocului. Dificultatea poate fi modificată doar de jucătorul care detine camera, dar și celălalt jucător poate să vadă care dificultate este aleasă.

## 4.6 Metode de îmbunătățire a proiectului

Jocul a ajuns într-un punct destul de bun, în care poate să fie jucat cap coadă de jucători, iar modul co-op este complet funcțional, dar care este mai mult un exemplu de ce poate deveni. Jocul poate fi îmbunătățit într-o serie de moduri pe care, din nefericire, nu mai am timp să le implementez, dar pe care doresc totuși să le enumerez în acest capitol.

### 4.6.1 Artă îmbunătățită

În primul rând, arta este un factor major pentru care oamenii aleg să joace un joc. Artă actuală a jocului este mai mult conceptuală, nu este finisată deloc. Am încercat să definesc câteva turnuri finisate în Blender, dar pentru a crește calitatea jocului trebuie realizate o serie de lucruri legate de aspectul grafic al jocului:

1. Toate turnurile să fie schimbate cu modele și materiale realiste pe ele. Pe lângă asta, ar fi util să aibă și animații de construire a turnurilor, distrugere a acestora, și după caz, al atacului.
2. Hărțile de joc trebuie să fie făcute modele 3D adevărate, ci nu doar niște planșe care definesc zonele pe unde pot să circule inamicii. Aceste nivele ar trebui ideal să fie environmenturi cu o tonă de alte modele de umplutură. Câteva exemple de hărți realiste ar fi: o hartă pe dealuri, înconjurați de copaci, tufe și alte obiecte/creaturi găsite în natură. În centrul hărții ar fi o zona defrișată unde poți construi turnurile. O altă idee ar fi un oraș abandonat, cu o tematică a culorilor foarte sumbră. Clădirile din oraș pot să construiască un fel de labirint artificial, iar turnurile să poată fi construite pe clădiri, sau între aceste clădiri.
3. Fiecare nivel sau fiecare hartă să aibă o tematică a culorilor folosite. La început să fie culori deschise și usoare pentru ochi, iar pe final culorile să devină foarte intense.
4. Hexagonale pot fi ascunse, și să apară doar în momentul în care dorim să construim un turn. Modul în care apar ar fi folosit un shader transparent, poate chiar cel construit deja.

5. Efectele speciale pot fi îmbunătățite, și pot fi adăugate efecte speciale pentru acțiunile care nu au fost acoperite încă (îmbunătățirea turnurilor, distrugerea acestora)
6. Animații la toate UI-urile și tranzițiile între scene.



Figure 17: Exemplu de grafică îmbunătățită pentru un turn defensiv

#### 4.6.2 O poveste pentru joc

Am definit deja o poveste de bază pentru joc în capitolul 3.2, dar aceasta a rămas drept un concept neimplementat. Jocul ar deveni cu mult mai calitativ dacă ar avea o poveste care să descrie evenimentele care au loc. Pe lângă asta, ar fi de dorit un sistem de dialog în care comandantul discută cu soldații lui, cu centrul de operații de pe planeta mamă sau cu inamicii. Pentru toate aceste interacțiuni de dialog, ar putea fi desenate imagini cu caracterele jocului sau s-ar putea folosi direct modelele lor 3D, la care ar trebui realizate animații faciale, sincronizarea buzelor, și animații specifice în funcție de evenimentele întâmpilate (când sunt fericiți, când au pierdut o misiune, când sunt surprinși, etc.)

#### 4.6.3 Sunetele pentru joc

Desigur, nu există joc fără sunete, oricât de retro ar putea să fie jocul. Jocul ar trebui să aibă câteva melodii de fundal care să acapareze jucătorul. Ideal fiecare harta diferită ar avea o melodie diferită, dar acest lucru ar fi greu de realizat fără o persoană dedicată pe această parte. Sunete de atac și pentru efectele speciale ar fi și ele foarte de dorit, iar la finalul unui nivel ar putea fi adăugate sunete comice în cazul pierderii nivelului.

#### 4.6.4 Monștrii multipli pentru modul co-op

Inițial am venit cu mai multe concepte pentru monștrii care pot fi adăugați în modul co-op, dar am avut timp să implementez doar unul din aceștia, motiv pentru care o să îi înșirui pe restul aici:

1. **Necromancer.** Este un monstru mai mic decât monstrul de foc și se poate teleporta la diverse locații pe hartă. Are 3 atacuri: un atac în care alege o direcție și lansează o bară oblică care lovește tot ce îi stă în cale, un atac în care cheamă alți inamici pe hartă, inamici care vor ataca toate turnurile pe care le găsesc în cale. Atacul lui special este să se teleporteze în spatele unei baze și să provoace foarte multe daune.
2. **Monstrul corosiv.** În fiecare secundă de joc, toate turnurile vor primi daune mici, dar care se adună cu timpul. De asemenea are 3 atacuri: un atac în care scuipă acid într-o zonă largă peste turnuri și crește viteza cu care le scade viața. Atacul lui special ar fi că își poate separa corpul în mai multe bile mai mici, care se mișcă încet una spre alta. Dacă acestea se adună toate, monstrul revine la normal și își refac multă viață. Scopul jucătorilor este să distrugă cât mai multe astfel de bile în timpul alocat.
3. **Armadillo.** Asemănător vietății cu același nume, este un monstru imens care are foarte multă viață. Din când în când alege să se facă în formă de bilă și să se rostogolească peste toate turnurile de pe harta. Scopul, la fel ca și la monstrul de foc, este să îi provoace suficiente daune într-un timp cât mai scurt pentru a îl întrerupe din a ataca. Celelalte atacuri nu au fost definite încă.

#### 4.6.5 Balansarea dificultății

Desigur, pentru un joc reușit, proprietățile trebuie să fie balansate într-un mod cât mai ideal. Jocul nu trebuie să fie nici foarte dificil dar nici extrem de ușor. Jucătorul ar trebui să se simtă nevoit să conceapă strategii de plasare a turnurilor pentru a reuși să câștige nivelele, iar din când în când să fie nevoit să joace nivelele anterioare și să obțină 3 stele la fiecare, pentru a își îmbunătăți turnurile.

Balansarea dificultății este o problema serioasă în industria jocurilor, motiv pentru care există persoane specializate care se ocupă de acest lucru și sunt mulți testeri care parcurg jocul și își dau cu părerea despre modul în care poate fi îmbunătățit.

## 5 Concluzii

Jocurile video au început să acapareze din ce în ce mai mult viețile noastre, iar industria aceasta este în continuă dezvoltare, motiv pentru care este o arie care merită explorată din perspectiva unui programator și nu numai.

Acest proiect a fost realizat în Unity pentru a mări viteza de dezvoltare și a crește calitatea jocului rezultat. În plus, a fost necesară folosirea unor tehnologii specifice în Unity, precum: Photon Network, Shader Graph, VFX Graph, etc.

Dezvoltarea proiectului a necesitat o planificare cât mai organizată a funcționalităților și a modului de implementare, fapt care a ușurat exponential procesul de dezvoltare. Pe parcursul acestui proiect am documentat o serie de concepte utile de organizare a codului și a modului de lucru din industria profesionistă.

Jocul rezultat conține o multitudine de sisteme complexe care interacționează între ele pentru a crea experiențe cât mai interesante pentru utilizatori. Implementarea acestor sisteme a presupus utilizarea multor tehnologii inginoase, iar rezultatul final este pe măsură așteptărilor.

Implementarea experienței co-op a presupus o provocare datorită dificultății de sincronizare a datelor jucătorilor. Serverul folosit de aplicație este un server oferit gratuit de Photon Engine, iar sincronizarea datelor s-a realizat cu ajutorul funcționalităților definite de acesta. În ciuda faptului că sincronizarea tuturor obiectelor și interacțiunilor din scenă a presupus o dificultate pentru acest proiect, în cele din urmă a fost realizată cu succes.

Acest joc poate să fie îmbunătățit și extins într-o serie de moduri: prin adăugarea unei grafici îmbunătățite, adăugarea sunetelor și melodiilor pentru joc, adăugarea unei povești și a unui sistem de dialog, inamici multiplii pentru modul co-op și prin balansearea corespunzătoare a dificultății.

Proiectul a fost distractiv de conceput și recomand tuturor persoanelor care sunt interesate să învețe modul în care funcționează jocurile video, să încerce pe cont propriu un proiect similar cu acesta.

## 6 Bibliografie

- [1] Mike McShaffry, David Graham.  
*Game Coding Complete Fourth Edition.* 2012
- [2] Jesse Schell.  
*The Art of Game Design: A Book of Lenses 1st Edition.* 2008
- [3] Eleonor Ciurea, Laura Ciupala.  
*Algoritmi: Introducere în algoritmica fluxurilor în rețele.* 2006
- [4] Photon API: <https://doc.photonengine.com/en-us/pun/current/getting-started/pun-intro>
- [5] Unity API: <https://docs.unity3d.com/ScriptReference/>
- [6] <https://docs.unity3d.com/ScriptReference/GameObject.html>
- [7] <https://docs.unity3d.com/ScriptReference/Mesh.html>
- [8] <https://docs.unity3d.com/ScriptReference/Collider.html>
- [9] <https://docs.unity3d.com/ScriptReference/Transform.html>
- [10] <https://docs.unity3d.com/ScriptReference/Rigidbody.html>
- [11] <https://docs.unity3d.com/ScriptReference/Input.html>
- [12] <https://docs.unity3d.com/ScriptReference/Coroutine.html>
- [13] <https://docs.unity3d.com/ScriptReference/Physics.Raycast.html>
- [14] <https://docs.unity3d.com/Manual/StreamingAssets.html>
- [15] <https://docs.unity3d.com/ScriptReference/Application-persistentDataPath.html>
- [16] <https://www.reddit.com/r/gamedev/>
- [17] <https://www.youtube.com/c/Brackeys>
- [18] <https://www.youtube.com/playlist?list=PLPV2KyIb3jR4u5jX8za5iU1cqnQPmbzG0>