



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

LUCRARE DE LICENȚĂ

Serenity Garden TD

Conducător Stiintific:
Lector Universitar Deaconu Adrian

Absolvent:
Opria Ion-Bogdan

Brașov, 2021

Contents

1 Introducere	5
1.1 Descrierea proiectului	5
1.2 Introducere in dezvoltarea jocurilor video	6
1.2.1 Scurta istorie	6
1.2.2 Industria curenta	6
2 Introducere in Unity	6
2.1 Prezentare generala	6
2.1.1 GameObject si Mesh	6
2.1.2 Collider	6
2.1.3 Transform si Rigidbody	6
2.1.4 API-ul de scriptare	6
2.2 Motivul alegerii pentru proiectul de licenta	6
2.3 Elemente specifice folosite	6
2.3.1 Shader Graph	6
2.3.2 VFX Graph	6
2.3.3 Photon Engine	6
3 Planificare Proiect	6
3.1 Faza de Proiectare	6
3.2 Tematica Jocului	7
3.3 Caracterele Jocului	7
3.3.1 Inamicii	8
3.3.2 Turnuri defensive	8
3.4 Sistemele Jocului	12
3.4.1 Harta de joc	12
3.4.2 Sistemul Turnurilor	14
3.4.3 Comandantul	14
3.4.4 Valul de inamici	14
3.4.5 Nivelele jocului	15
3.4.6 Lock-on	15
3.4.7 Magazinul de imbunatatiri permanente	15

3.4.8	Raid system	16
3.4.9	Atacurile monstrului de foc	16
3.5	Analiza proiectului din punctul de vedere al consumatorilor . .	17
3.5.1	Dificultatea jocului	17
3.5.2	Elemente de dependenta	18
3.5.3	Grupele de varsta vizate	18
4	Implementare proiect	18
4.1	Scena de lupta	18
4.1.1	Initializarea scripturilor	18
4.1.2	Procesarea comenzilor venite de la utilizator	20
4.1.3	Grid system	22
4.1.4	Sistemul de navigare	22
4.1.5	Interfete folosite	23
4.1.6	Ierarhia inamicilor	26
4.1.7	Ierarhia turnurilor defensive	27
4.1.8	Comandantul	30
4.1.9	Controlul turnurilor si starilor de joc	32
4.2	Selectarea nivelelor	32
4.2.1	Structura meniului	32
4.2.2	Generarea dinamica a meniului	32
4.3	Salvarea datelor	32
4.3.1	Salvarea automata	32
4.3.2	Criptarea si decriptarea datelor	32
4.4	Magazinul de imbunatatiri permanente	32
4.4.1	Structura meniului	32
4.4.2	Generarea dinamica a meniului	32
4.5	Sistemul co-op	32
4.5.1	Modul de lucru cu Photon Engine	32
4.5.2	Inregistrare si camera de asteptare	32
4.5.3	Sincronizarea datelor	32
4.5.4	Monstrul de foc	32
4.6	Metode de imbunatatire a proiectului	32

4.6.1	Arta imbunatatita	32
4.6.2	O poveste pentru joc	32
4.6.3	Monstrii multipli pentru modul co-op	32
5	Concluzii	32

1 Introducere

1.1 Descrierea proiectului

Jocurile video au inceput sa acapareze din ce in ce mai mult vietile noastre datorita usurintei de accesare si a experientei pe care o ofera. Odata cu evolutia hardware-ului, jocurile video au devenit din ce in ce mai realiste si prin urmare ofera experiente care nu poate fi gasite in niciun alt mediu existent, deoarece in nici un alt mediu nu putem controla si traii experientele unor caractere atat de indetaliat. Industria jocurilor video este o industrie care este in continua crestere si prin urmare este o ramura care merita explorata din perspectiva unui programator si 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 licenta sa fac un joc video. Jocul se numeste ”Serenity Garden” si este un tower defense 3D. Experienta jocului are loc in cateva nivele special definite, in care utilizatorul trebuie sa isi protejeze baza de operatii. Există multe tipuri diferite de inamici care vor sa distruga baza jucatorului, iar acesta, pentru a o proteja, trebuie sa construiasca un sistem defensiv. Jucatorul poate sa construiasca turnuri defensive care au diferite efecte asupra inamicilor. Inamicii ataca sistemul defensiv, sau in cazul in care lipseste, ataca direct baza jucatorului. Daca reușesc sa distruga baza, jucatorului, acesta pierde nivelul curent.

Jucatorul nu poate sa construiasca turnuri oriunde, ci are locuri predefinite unde poate sa le construiasca, locuri specificate de hexagoane colorate pe harta de joc. Odata plasat un turn, aceasta nu poate fi mutat, dar jucatorul are si un caracter invulnerabil pe care il poate muta oriunde doreste pe harta. Acest caracter va ataca inamicii automat cand nu se afla in miscare, si poate intra in turnuri pentru a le creste puterea/eficienta.

Modul care va diferenția acest joc de celelalte jocuri pe acest stil este modul de co-op. Doi jucatori se vor putea conecta prin retea si vor juca un nivel dificil care necesita cooperare si o planificare buna intre ei.

1.2 Introducere in dezvoltarea jocurilor video

1.2.1 Scurta istorie

1.2.2 Industria curenta

2 Introducere in Unity

2.1 Prezentare generala

2.1.1 GameObject si Mesh

2.1.2 Collider

2.1.3 Transform si Rigidbody

2.1.4 API-ul de scriptare

2.2 Motivul alegerii pentru proiectul de licenta

2.3 Elemente specifice folosite

2.3.1 Shader Graph

2.3.2 VFX Graph

2.3.3 Photon Engine

3 Planificare Proiect

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 ca o planificare buna a unui proiect inca de la inceput poate imbunatatii calitatea proiectului exponential, asigura usurinta de adaugare a unor elemente noi (fara sa fie necesara rescrierea/re-organizarea proiectului) si reduce frustrarea programatorilor care lucreaza la proiect.

Din acest motiv, inca de la inceputul proiectului am incercat sa imi planific cat mai indetaliat continutul acestuia si modul in care il voi implementa.

In prima faza, mi-am scris un document de proiectare care continea descrierea si elementele jocului, privite din multe perspective.

Introducerea proiectului a fost realizata la inceputul acestui document si nu va fi reluată aici.

3.2 Tematica Jocului

Jocul are loc in viitorul departat. Planeta ta este supra-populata, iar nivelul de poluare a ajuns la un nivel critic. Umanitatea a trecut de mult de punctul in care mai pot salva aceasta planeta. Intreaga planeta va deceda in cateva decenii, iar oamenii daca nu isi gasesc o alta locuinta intre timp, vor muri si ei impreuna cu ea. In acest scop, o armata speciala a fost formata, care are ca scop explorarea altor planete si gasirea uneia care poate suporta rasa umana.

Universul este totusi foarte periculos. Unele planete au viata extraterestra foarte agresiva, iar altele au fost distruse de experimente cibernetice esuate. Jucatorul este comandantul suprem al acestei armate care are ca scop protejarea oamenilor de stiinta care vor analiza aceste planete.

Prima faza a proiectului va contine o singura planeta care poate fi populata dar care contine multi inamici periculosi.

Planeta mama trimite resurse din cand in cand, dar numai un numar limitat de resurse pot fi trimise deodata, iar acestea dureaza mult timp sa ajunga, asa ca nu putem depinde de ele in timpul unei lupte.

3.3 Caracterele Jocului

Comandantul este protagonistul jocului. El este caracterul a carei perspective o vom trai pe tot parcursul jocului. In trecut a fost cobai pentru un experiment menit sa creeze super-soldati, iar in urma acestui experiment a primit o putere speciala. Poate sa isi ascunda complet prezenta fata de alti oameni sau alte creaturi. Din acest motiv, el este caracterul invulnerabil pe care il putem misca pe harta in timpul luptei.

3.3.1 Inamicii

- **Inamicul de lupta apropiata** este un tip de inamic care poate ataca turnurile doar cand sta fix in fata lor. Are viata mai multa decat inamicii de lupta de la distanta.
- **Inamicul de lupta de la distanta** este un tip de inamic care poate ataca turnurile de la distanta. Are viata putina, dar pot ataca repede.
- **Inamicul zburator** este un inamic care poate fi lovit doar de un numar limitat de turnuri.
- **Inamicul bombardier** este un tip de inamic care ignora turnurile intalnite in cale. Aceasta se misca pe cel mai scurt drum catre baza jucatorului, iar daca ajunge deasupra acesteia, va lansa bombe care ii vor scadea drastic viata, dupa care vor pleca de pe mapa. Este un tip de inamic zburator care poate fi lovit doar de un numar limitat de turnuri.

Acesti inamici (cu exceptia bombardierului) se vor misca pe harta calculand cel mai scurt drum catre baza jucatorului. Daca acestia intalnesc in cale un turn, se vor opri si vor ataca acel turn. Cand turnul este distrus, acestia isi vor relua drumul.

Pentru fiecare inamic vor fi 3 tipuri de variatii, fiecare mai puternic decat cel precedent.

3.3.2 Turnuri defensive

In continuare voi scrie o descriere scurta pentru fiecare turn din joc. Toate proprietatile despre care urmeaza sa vorbesc sunt in comparatie cu celelalte turnuri defensive.

- **Serenity** este baza jucatorului. Are viata foarte multa daca este distrusa jucatorul pierde nivelul curent. Aceasta poate ataca inamicii care se apropie de ea. Are raza de atac medie, putere de atac mediu, poate ataca orice tip de inamic, iar rata de reincarcare intre atacuri este mica (dureaza mult timp intre atacuri)

- **Mitraliera automata** (fig: 1) Are viata mica, poate ataca orice tip de inamic, are raza medie, putere de atac mica si rata de atac este mare (ataca foarte des)
- **Gardul electric** (fig: 2) este turnul de protectie. Are viata foarte mare, poate ataca doar inamicii de lupta apropiata, puterea de atac este medie, iar rata de reincarcare este mica. Scopul acestui turn este sa stea in calea inamicilor pentru a fi atacat de acestia, permitand celorlalte turnuri sa atace inamicii blocati.
- **Vulkan** (fig: 3) Este turnul special construit pentru a lupta impotriva inamicilor zburatori. Are viata putina, raza de atac mare si putere de atac mare dar poate ataca doar inamicii zburatori.
- **Aruncatorul de flacari** (fig: 4). Cand inamicii intra in raza acestui turn, el va imprastia flacari catre inamici. Toti inamicii care intra in aceste flacari primesc damage in fiecare clipa in care stau in flacari. Are viata medie, ataca instant si puterea de atac este mica dar distribuita in fiecare clipa in care inamicii stau in flacarile acestui turn.
- **Laser** (fig: 5). Este un turn similar cu aruncatorul de flacari in sensul in care ataca in fiecare clipa in care un inamic este in raza acestuia. Ataca cu un laser puternic care scade treptat viata inamicilor. Are viata medie, ataca instant si puterea de atac este mica.
- **Excavator** (fig: 6) este un tip special de turn deoarece nu poate ataca inamicii. Acesta aduna resurse in timpul luptei, resurse care pot fi convertite in bani de joc. Are viata multa, si pentru ca nu poate ataca, trebuie sa fie aparat de inamici.



Figure 1: Mitraliera automata

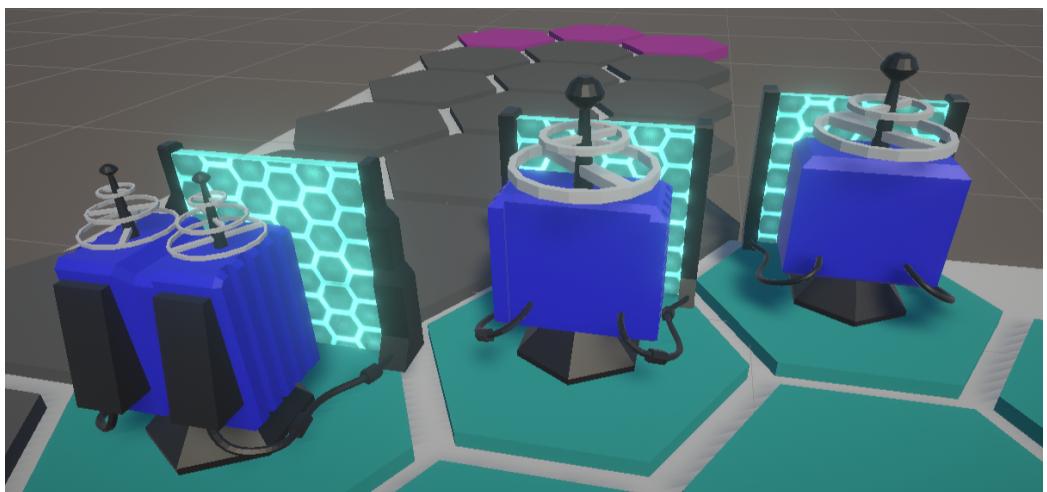


Figure 2: Gardul electric



Figure 3: Vulkan

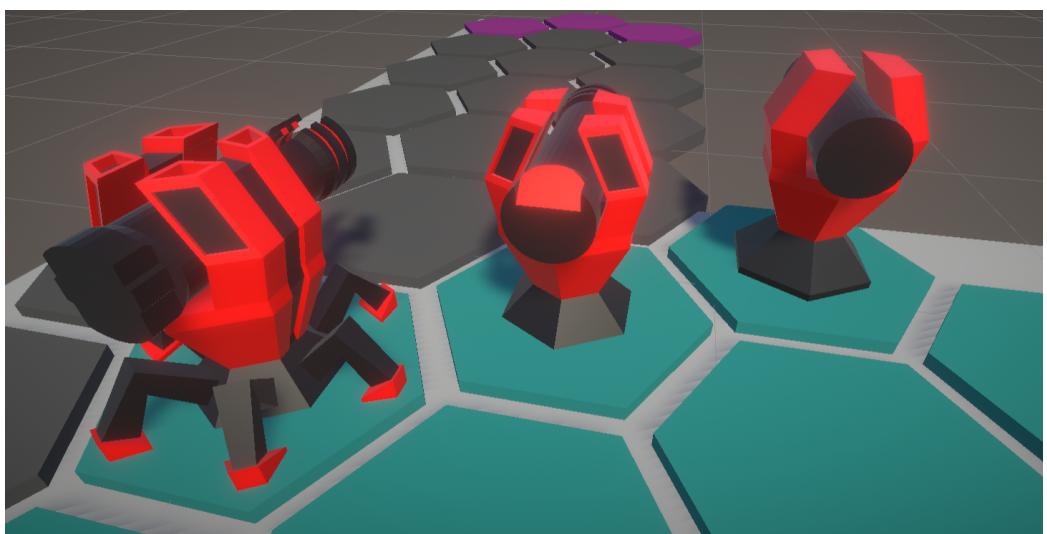


Figure 4: Aruncatorul de flacari

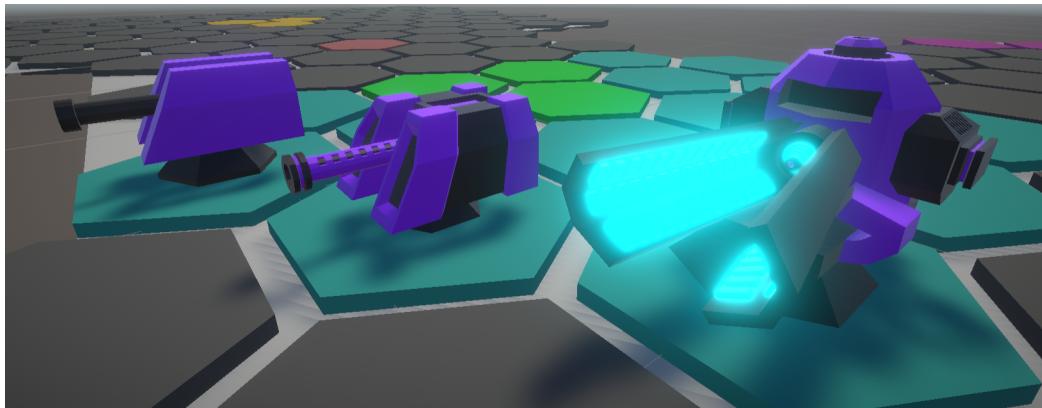


Figure 5: Laser

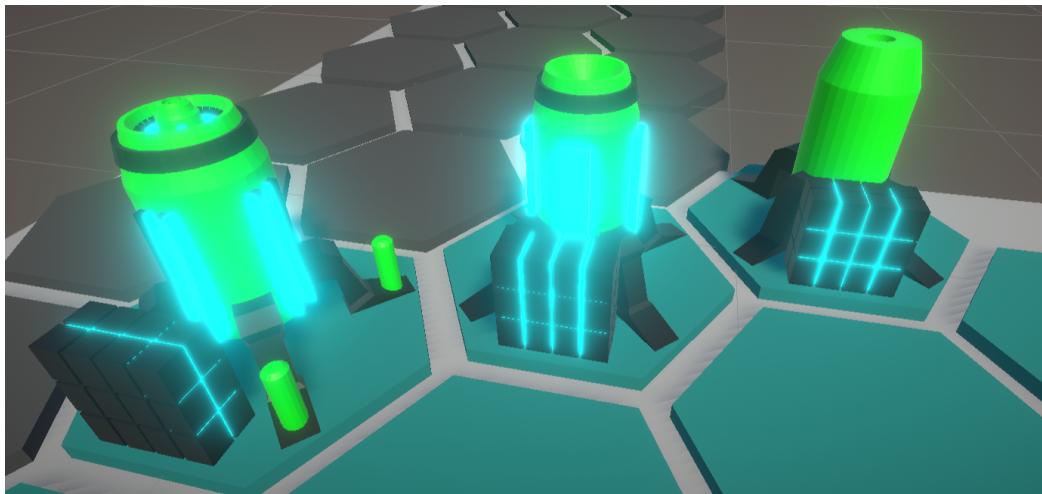


Figure 6: Excavator

3.4 Sistemele Jocului

3.4.1 Harta de joc

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

- Hexagoane goale (cele gri din (fig: 7)).
- Hexagoane pe care putem construi turnuri de atac. (cele albastre din (fig: 7))

- Hexagoane pe care putem construi turnul de extragere de resurse. (cele verzi din (fig: 7))
- Hexagonul pe care va incepe si se va afla comandantul. (cel roz din (fig: 7))
- Hexagoanele bazei. Baza va ocupa 3 hexagoane de acest tip. (cele galbene din (fig: 7))
- Hexagonul ocupat. Fiecare hexagon care este ocupat de o anumita structura devine un astfel de hexagon. (are culoarea rosie)

Inamicii vor calcula drumul pe care merg in functie de toate aceste tipuri de hexagoane. Din acest motiv este nevoie si de cel gol.

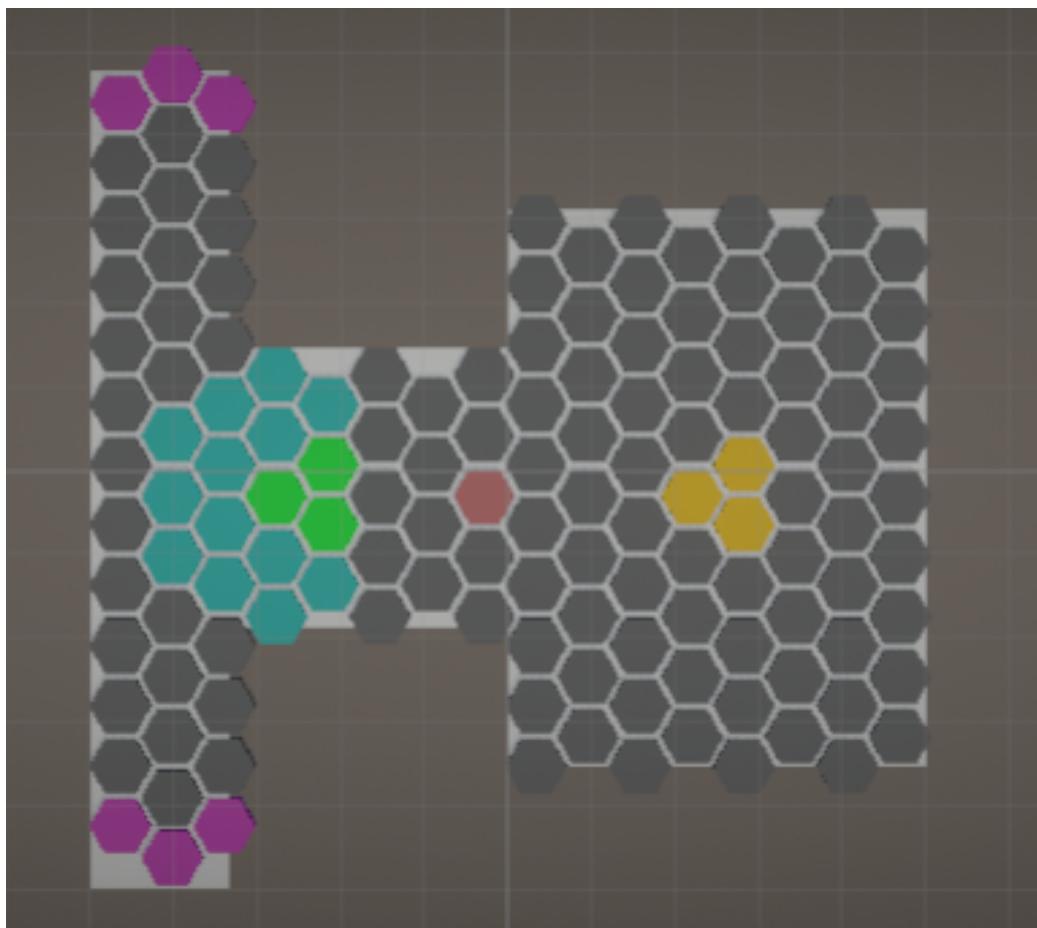


Figure 7: Harta de joc

3.4.2 Sistemul Turnurilor

Turnurile pot fi plasate doar pe hexagoanele de constructie. Odata plasate, acestea nu pot fi mutate. Singurele metode de a goli acel hexagon sunt sa fie distrus de un inamic sau sa vindem tureta respectiva. Daca vindem o tureta, vom primii inapoi o parte din banii investiti, bani cu care putem sa imbunatatim turnurile. Toate turnurile au urmatoarele proprietati:

- Viteza de atac
- Putere de atac
- Viata
- Raza de atac
- Inamicii pe care pot sa ii atace

Daca viata turetei ajunge la 0, va fi distrusa automat. Atata timp cat o tureta nu este distrusa, o putem repară, dar reparatiile vor fi executate intr-un anumit timp, iar in acest timp, tureta nu va putea sa atace dar poate sa fie atacata in continuare.

3.4.3 Comandantul

Comandantul poate fi mutat pe orice hexagon care nu este deja ocupat. Dupa ce ai ales un hexagon, acesta va incepe sa se miste catre acesta. Cat timp este in miscare, nu poate ataca inamicii. Destinatia poate fi schimbata doar dupa ce a ajuns la destinatie. Are raza medie, rata de atac medie, putere de atac mediu si poate ataca orice tip de inamic.

Acesta poate intra si in turete, imbunatatindu-le exponential.

3.4.4 Valul de inamici

Jocul va avea mai multe nivele, care pot fi selectate dintr-un meniu, acestea fiind cat mai diversificate posibil. Inamicii au mai multe puncte din care pot sa vina in functie de nivelul ales. Acestia vin in grupuri de inamici. Intre fiecare grup exista o perioada de repaus in care jucatorul isi poate repară

turnurile. In caz ca nu are nevoie de reparatii, poate selecta sa sara peste perioada de repaus, astfel primind bani extra.

3.4.5 Nivelele jocului

Nivelul este terminat cand toti inamicii au fost omorati. Fiecare nivel va avea un scor de maxim 3 stele, care specifica cat de bine s-a descurcat la acest nivel. Scorul este calculat in functie de viata ramasa a bazei la finalul nivelului. Fiecare stea castigata va da bani extra, bani care pot fi folositi in magazinul de imbunatatiri permanente, despre care vom discuta in scurt timp.

Jucatorul poate sa joace din nou nivelele, dar nu va mai primii atat de multi bani ca prima data. Aceasta va castiga bani extra de pe stelele castigate doar daca a primit mai multe stele decat turele anterioare.

3.4.6 Lock-on

Jucatorul poate sa apese pe inamici, astfel setand acel inamic drept o prioritate. Fiecare turn care poate ataca acest tip de inamic il va prioritiza fata de alti inamici. De indata ce este omorat, se pierde aceasta prioritate.

3.4.7 Magazinul de imbunatatiri permanente

Intre nivele putem sa cumparat piese de schimb pentru turnurile noastre pentru a le face mai puternice. Aceste piese se pot cumpara doar cu banii castigati de pe urma nivelelor. Im bunatatirile pt fiecare tureta in parte sunt urmatoarele:

- Mitraliera automata: atac mai puternic, raza de atac mai mare, costuri mai mici.
- Gardul electric: mai multa viata, atac mai puternic, costuri mai mici.
- Vulkan: atac mai puternic, range mai mare, atac mai rapid
- Aruncatorul de flacari: atac mai puternic, range mai mare

- Laser: atac mai puternic, range mai mare
- Excavator: da bani mai multi, timpul intre livrarile de bani mai scurt

Fiecare imbunatatire va avea mai multe nivele, fiecare costand din ce in ce mai multi bani, dar vor avea un nivel maxim la care putem duce aceste imbunatatiri.

3.4.8 Raid system

Cand un jucator alege acest mod, este dus la o pagina de login in care trebuie sa isi specifiche numele sau sa il lase pe cel generat automat. Dupa ce intra in cont, trebuie sa intre intr-un lobby. Ca sa intre intr-unlobby poate sa creeze el un lobby, sa aleaga un lobby dintr-o lista cu toate lobbyurile existente sau sa intre automat intr-un lobby existent. In cazul ultimei optiuni, in caz ca nu exista nici un lobby, acesta se creeaza automat. La aceasta pagina poate sa selecteze dificultatea nivelului si daca coechipierul este pregatit. De indata ce amandoi jucatorii sunt gata, pot sa inceapta nivelul.

In centrul mapei se afla un monstru imens de foc. Scopul nivelului este sa invingi monstrul impreuna cu coechipierul tau, inainte ca monstrul sa distruga baza vreunuia dintre jucatori.

Fiecare jucator poate pласa turnuri oriunde, putand sa imbunatateasca, repară sau vinde doar turnurile construite de ei. Amândoi playerii au caracterele lor comandanți, dar aceștia nu pot intra în același tureta în același timp. Pentru sistemul de lock-on, doar turnurile jucatorului 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 dificultatile avansate.

- Abilitate automata: creste puterea de atac pe masura ce pierde din viata

- Ploaia de meteoriti: ridica o mana sus si genereaza meteoriti unul dupa altul. Fiecare meteorit cand este construit complet alege un turn si porneste catre directia acestuia. Cand se izbeste de turn explodeaza si scade din viata turnului in functie de dificultatea selectata.
- Gheara invartitoare: se invarte 360 grade si loveste toate turnurile de pe harta cu mana lui dreapta.
- Meteoritul suprem: cand ajunge la un anumit procentaj de viata incepe acest atac. Alege una dintre bazele jucatorilor (in caz ca sunt mai multi), se roteste spre ea si incepe sa incarce un meteorit imens. Incarcarea acestuia dureaza 30 de secunde. Daca in acest timp jucatorul reuseste sa ii scada suficient de mult viata monstrului, atacul este intrerupt si el devine confuz timp de 10 secunde, timp in care poate fi atacat neintrerupt. Daca jucatorii nu reusesc sa ii scada suficient de mult viata, va lovii puternic cu meteoritul baza selectata si turetele de prin jur.

Toate interactiunile acestui nivel sunt sincronizate in retea prin utilizarea Photon Engine-ului.

3.5 Analiza proiectului din punctul de vedere al consumatorilor

3.5.1 Dificultatea jocului

Jocul va fi relativ dificil. Jucatorul trebuie sa se gandescă unde să construiască anumite turnuri pentru a folosi resursele cât mai bine. La început jocul va fi usor, dar pe măsură ce progresează, nivelele vor deveni din ce în ce mai dificile. Trebuie să se gandescă și dacă poate rezista să sără peste perioada de repaus între grupurile de inamici.

Raid system-ul are un nivel de dificultate cu mult mai ridicat, pentru că trebuie să se gandească cum să se coordoneze cât mai bine cu coechipierul.

3.5.2 Elemente de dependenta

In caz ca nu putem depasii un anumit nivel, putem juca nivelele anterioare si sa ne asiguram ca luam 3 stele la fiecare nivel. Aceasta metoda de a obtine 3 stele la fiecare nivel poate fi unul din motivele care ii determina pe jucatori sa continue sa joace jocul. Un alt factor ar putea fi sistemul de imbunatatiri permanente.

3.5.3 Grupele de varsta vizate

Acest joc nu are limitari de varsta, poate fi jucat de oricine, dar va fi apreciat cel mai mult de copii si de adolescentii care cauta provocari in jocurile de strategie. Jocul este in principiu facut pentru "casual gamers".

4 Implementare proiect

4.1 Scena de lupta

4.1.1 Initializarea scripturilor

In Unity scripturile au cateva metode care ajuta la initializare.

Awake este metoda care este apelata la inceputul programului si inainte de restul metodelor.

Start este metoda care este apelata dupa 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 incepe dupa ce toate metodele "Start" au fost apelate.

Pentru "Start" si "Awake", unity decide automat in ce ordine sa le apeleze, fara ca noi sa putem controla acest lucru. In multe proiecte mai complexe se ajunge in momentul in care, pentru initializarea anumitor sisteme este necesar ca alte sisteme sa fie deja initializate. Cum nu putem controla ordinea de initializare, se ajunge in punctul in care putem primi erori aleatorii din cauza initializarii haotice.

Pentru a combate acest lucru, m-am gandit la un sistem care va permite controlarea initializarii tuturor acestor procese cu usurinta.

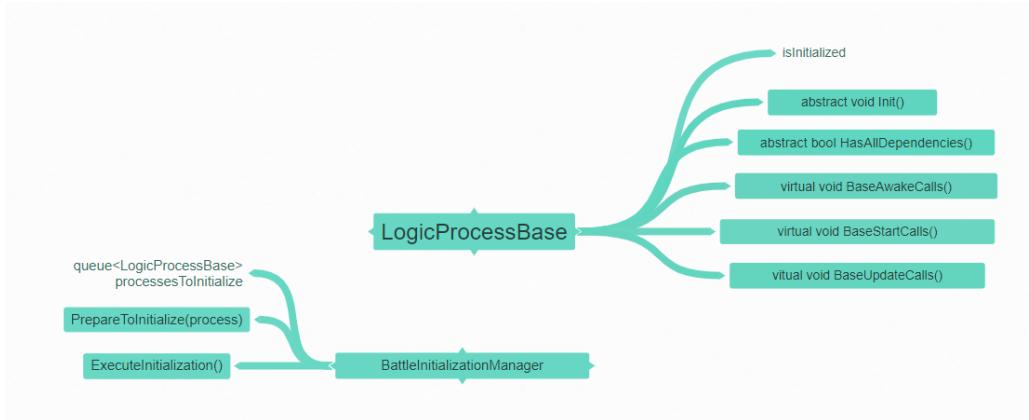


Figure 8: Initializarea scripturilor

Am creat clasa LogicProcessBase care ajuta in acest proces. Fiecare script care necesita o initializare mai organizata trebuie sa o mosteneasca. Aceasta contine:

- **isInitialized** care specifica daca scriptul curent a fost initializat sau nu inca
- **Init()**. In aceasta metoda trebuie sa scriem tot codul de initializare a clasei care mosteneste.
- **HasAllDependencies()**. Este o metoda care returneaza un boolean. In aceasta clasa trebuie sa punem toate dependintele de care are nevoie clasa care mosteneste.
- **BaseAwakeCalls()**. In aceasta metoda apelam ”PrepareToInitialize()” din clasa ”BattleInitializationManager”, despre care vom vorbi in scurt timp.
- **BaseStartCalls()** si **BaseUpdateCalls()** sunt metode in care trebuie sa scriem tot codul care normal ar fi venit in Start/Update. In unity, nu putem defini Start/Update drept virtual si sa modificam functiile

lor pe urma, din acest motiv tot codul din clasele dintr-o ierarhie mai complexa trebuie sa fie scris in aceste metode, iar in clasa cel mai jos din ierarhie trebuie sa le apelam in Awake/Start/Update.

BattleInitializationManager este a doua clasa de care avem nevoie. Ea este responsabila pentru a executa initializarea propriuzisa a proceselor.

PrepareToInitialize() este apelata de fiecare proces in metoda lor de awake, iar aceasta metoda va adauga procesul intr-o coada.

ExecuteInitialization() parcurge toate procesele din coada si verifica daca acestea pot fi initializate, apeland `"HasAllDependencies()"`. Daca procesul curent poate fi initializat, atunci apeleaza metoda `"Init()"` si seteaza `"isInitialized = true"`. Daca nu il poate initializa, il adauga la finalul cozii pentru a fi initializat la final. Acest proces se repeta pana cand coada este goala sau s-a parcurs o data coada cap-coada si nu s-a initializat nici un proces. In acest caz inseamna ca aveau dependinte circulare si oprim procesul de initializare, afisand un mesaj de eroare. Aceasta metoda trebuie apelata in Awake, Start si Update, deoarece, pe parcurs pot aparea noi procese care trebuie initializate, iar programul trebuie sa poata sa detecteze astfel de cazuri.

4.1.2 Procesarea comenziilor venite de la utilizator

O alta problema des intampinata in proiectele mari este procesarea comenziilor venite de la utilizator. In cazul in care procesam comenziile in mai multe script-uri, cand vrem sa schimbam modul in care functioneaza comenziile sau sa schimbam platforma pe care ruleaza jocul, va trebui sa schimbam toate scripturile in care procesam comenziile utilizatorului. In proiectele mari, acest lucru duce la mult timp pierdut si din acest motiv m-am gandit la o metoda cum sa imi organizez mai bine acest sistem.

Ideea propusa de mine este sa pastram toata procesarea de comenzi ale utilizatorului intr-un singur script. Acest script are definite evenimente/delegate pentru toate tipurile de input specifice. Alte scripturi se pot abona la aceste evenimente, iar cand utilizatorul apasa tasta respectiva, toate metodele abonate la evenimentul respectiv vor fi apelate.

Pe langa asta, este necesar sa stim si anumite informatii legate de comanda primita la utilizator. Informatiile aditionale de care am avut nevoie sunt:

- Pozitia pe ecran la care a fost apasat mouse-ul/ecranul (in cazul rularii pe android)
- Pozitia pe ecran la care a fost ridicata apasarea mouse-ului/ecranului (in cazul rularii pe android)
- Daca in clipa curenta este sau nu apasat mouse-ul/ecranul
- Timpul la care a fost apasat mouse-ul/ecranul
- Timpul la care a fost ridicata apasarea mouse-ului/ecranului
- Obiectul pe care utilizatorul a dat click in scena. In momentul cand apasam click, se creeaza un Raycast care are ca origine pozitia mouse-ului pe ecran si care se indreapta catre scena in perspectiva in care se afla camera la clipa curenta. Acest Raycast detecteaza daca a fost lovit un obiect, si daca da, retinem care obiect a fost lovit pt ca alte scripturi sa se poata folosi de aceasta informatie.
- Elementul cel mai de sus din ierarhia obiectului lovitur. Datorita faptului ca anumite modele au o structura complexa, uneori nu ne este deloc util sa stim in mod direct ce obiect a fost lovitur. De exemplu in caz ca apasam peste un turn si Raycast-ul loveste pusca turnului, nu ne va ajuta deloc in cazul in care ne intereseaza sa aflam ce tip de turn am lovitur. Din acest motiv, in clipa in care gasim obiectul pe care am dat click, retinem si parintele cel mai de sus din ierarhie.

Aceasta organizare a procesarii comenzilor ne permite sa schimab tastele folosite cu usurinta si sa adaugam suport pentru dispozitive noi foarte usor. In 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 comenzilor in cele 2 cazuri vor fi diferite, dar datorita faptului ca se invoca evenimentele deja devinute, alte script-uri nu trebuie modificate absolut deloc.

4.1.3 Grid system

4.1.4 Sistemul de navigare

Caracterele jocului trebuie sa stie pe unde pot sa mearga ca sa ajunga la o anumita destinatie. In acest scop a trebuit sa aleg un algoritm care sa gaseasca cel mai scurt drum posibil. Am avut de ales intre mai multi algoritmi specifici, precum: Dijkstra, A*, Floyd-Warshall, etc.

In cele din urma am ales sa folosesc Floyd-Wharshall. Acest algoritm are complexitatea $O(n^3)$ si presupune construirea celor mai scurte drumuri posibile pornind de la oricare nod catre oricare alt nod din retea. Deoarece construirea drumului se realizeaza doar la inceputul jocului si reconstruirea acestui drum se realizeaza aproape instantaneu, am ales sa folosesc acest algoritm.

”Consideram reteaua orientata $G = (N, A, b)$ reprezentata prin matricea valoare adiacenta $B = (b_{ij}), i, j \in N$ cu

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

Algoritmul Floyd-Warshall determina matricea distanteelor $D = (d_{ij}), i, j \in N$ si matricea predecesor $P = (p_{ij}), i, j \in N$.” [3]

```

function FLOYD-WARSHALL
    for i  $\leftarrow$  1 to n do
        for j  $\leftarrow$  1 to n do
             $d_{ij} \leftarrow b_{ij};$ 
            if i  $\neq$  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 Interfete folosite

Pentru ca o serie de obiecte din joc trebuie sa urmeze aceleasi concepte, am decis sa ma folosesc de interfete pentru a le definii modul in care trebuie sa functioneze.

IMovable

Deoarece inamicii si comandanțul se pot mișca pe mapa, am definit aceasta interfață pentru a defini aceleasi concepte pentru toate caracterelor care au nevoie de a se mișca pe mapa. Interfața se foloseste de sistemul de navigare definit anterior și conține urmatoarele:

- Nodul curent la care se află

- Nodul destinatie
- Nodul urmator la care trebuie sa se mute ca sa se apropie de destinatie
- Viteza de deplasare
- Distanța fata de urmatorul nod pt care consideram ca am ajuns la acesta. Deoarece lucram cu obiecte in spatiu 3D, este nevoie de o asemenea valoare.
- Un boolean care defineste daca am ajuns sau nu la destinatie
- O metoda care seteaza nodul urmator la care trebuie sa ajungem ca sa ne apropiem de destinatie. Aceasta face apel la matricea construita de sistemul de navigare.
- O metoda care misca propriuzis obiectul catre urmatorul nod
- O functie care verifica daca am ajuns sau nu la destinatie.

IAttacker

Este o interfata pe care toate obiectele/caracterele care vor sa atace alte obiecte trebuie sa o mostendeasca. Aceasta contine urmatoarele:

- Obiectul pe care trebuie sa il atace. Tipul este definit la mostenirea clasei datorita sablonului definit pentru interfata.
- Puterea de atac
- Raza de atac
- Durata intre atacuri
- Timpul la care s-a executat ultimul atac
- Intervalul de timp intre care se cauta un nou inamic
- Metoda care afisaza/ascunde raza obiectelor din scena. Raza acestora a fost definita intr-un shader special creat in Shader Graph

- O metoda care cauta cel mai apropiat inamic daca exista in raza
- Metoda de atac pe care fiecare obiect o implementeaza diferit.

IDestroyable

Fiecare obiect care are o viata anume si care poate fi distrus trebuie sa mosteneasca aceasta interfata. Aceasta contine urmatoarele:

- Viata obiectului.
- Banii primiti cand obiectul este distrus
- O metoda care distruge obiectul respectiv cand viata lui a ajuns la 0

IRecoverable

Turnurile pot sa isi refaca viata, iar ca un concept pentru viitor, ar putea sa existe si anumiti inamici care refac viata altor inamici. Interfata contine urmatoarele:

- Cata viata se reface in fiecare secunda cand efectul este aplicat.
- Costul refacerii in cazul in care este de dorit un cost.
- Un boolean care verifica daca se reface sau nu in clipa curenta.
- O metoda care porneste procesul de refacere

4.1.6 Ierarhia inamicilor

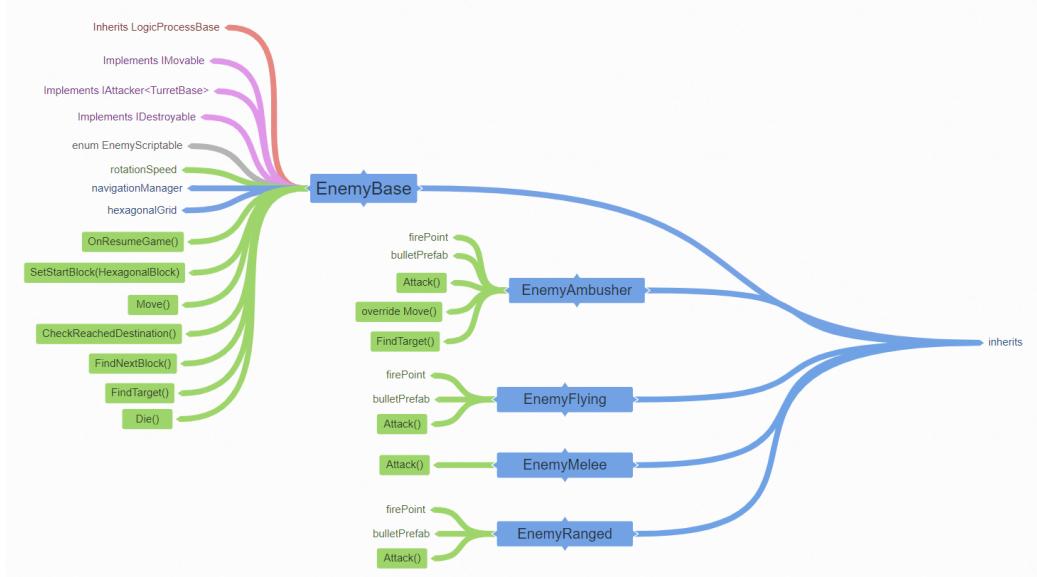


Figure 9: Ierarhia inamicilor

In 9 am definit intr-un mod simplu si colorat modul in care trebuie structurata ierarhia inamicilor din joc.

Clasa EnemyBase este clasa parinte care defineste modul principal in care inamicii trebuie sa functioneze. Aceasta clasa implementeaza 3 interfete utile, si anume IMovable, IDestroyable si IAttacker< TurretBase >. Precum am explicat si anterior, interfata IAttacker defineste un sablon care specifica tipul de obiect pe care poate sa il atace, in cazul nostru, clasa TurretBase care va fi explicata in urmatoarea sectiune.

Aceasta clasa face referire si la sistemul de navigare si la sistemul hartii hexagonale, si din acest motiv este nevoie sa isi faca initializarea doar dupa ce aceste doua sisteme si-au terminat initializarea lor. In acest scop, putem sa mostenim clasa LogicProcessBase, care a fost descrisa in capitolul 4.1.1.

Functiile implementate de aceasta clasa sunt in mare parte functiile mostenite dupa implementarile interfetelor descrie, cu cateva exceptii referitoare la alte sisteme din joc. Un astfel de sistem este cel de a pune pe pauza jocul. Cand jocul este pus pe pauza, toate scripturile care se foloseau de timpul

actual al jocului nu vor actiona in modul asteptat dupa ce reluam jocul. Din acest motiv este nevoie sa implementam functia OnResumeGame() care va fi apelata cand se reia jocul, iar in aceasta functie trebuie sa setam toti temporizatorii din script la valorile necesare.

Aceasta clasa nu este suficientea pentru a definii toate tipurile de inamici, deoarece face dificila diferențierea intre inamici pentru anumite sisteme si anumiti inamici actioneaza diferit fata de standard. Din acest motiv este necesar sa definim o serie de clase pentru fiecare inamic in parte, care sa mosteneasca aceasta clasa de baza, si care sa isi adauge propria lor logica la standardul definit de EnemyBase.

EnemyAmbusher nu se foloseste de sistemul de navigare. Este inamicul bombardier, care zboara prin aer direct catre baza jucatorului. Cand ajunge in contact cu aceasta, lanseaza bombe care provoaca multe pagube, dupa care isi continua drumul, parasind scena de lupta. Din acest motiv metodele Move() si FindTarget() trebuie sa fie suprascrise.

EnemyFlying si **EnemyRanged** sunt inamici care urmeaza in totalitate logica definita de EnemyBase. Singura exceptie la aceste doua clase este ca anumite turnuri vor face diferențiere intre ele. O parte din turnuri pot sa atace doar inamicii de sol (EnemyRanged si EnemyMelee) iar alte turnuri pot sa atace doar inamicii zburatori (EnemyFlying si EnemyAmbusher).

EnemyMelee este un inamic care sa atace alte turnuri doar cand a ajuns langa turnuri. Din acest motiv metoda lui de atac trebuie sa fie suprascrisa putin.

4.1.7 Ierarhia turnurilor defensive

Ierarhia turnurilor este putin mai complexa si va fi descrisa pe bucati.

TurretBase

Urmand aceeasi logica ca la inamici, am definit o clasa de baza pentru modul in care trebuie sa actioneze un turn defensiv. Aceasta implementeaza doua interfete: IDestroyable si IAttacker~~;EnemyBase;~~. Sablonul de la IAttacker defineste faptul ca un turn defensiv poate sa atace obiecte de tipul

EnemyBase, care este clasa din care mostenesc toti inamicii jocului.

TurretBase are si ea nevoie de o initializare controlata si din acest motiv trebuie sa mosteneasca clasa LogicProcessBase. TurretBase defineste proprietatile turnurilor, cum ar fi viata, puterea de atac, etc. Pe langa asta, majoritatea turnurilor pot fi imbunatatite, proprietatile crescand exponential cand acest lucru se intampla. Din acest motiv a fost nevoie sa creez fisiere configurabile care definesc valori pt proprietatile turnurilor. Aceasta clasa detine o instanta a unui astfel de fisier, si defineste o functie ”SetLevelProp(int level)” care citeste fisierul configurabil si seteaza toate proprietatile in functie de acesta. Pe langa asta, cand turnurile sunt imbunatatite, le schimba modelul 3D pentru a avea o schimbare vizuala.

Clasa defineste si alte metode care au fost abordate in capitolele anterioare, precum: Die(), OnResumeGame(), Init(), etc. Functia DrawRange() afiseaza/ascunde raza de atac a turnului, raza care a fost construita intr-un shader special definit in Shader Graph.

PlayerBase

PlayerBase este o clasa care mosteneste TurretBase, si reprezinta baza de operatii a jucatorului. Aceasta implementeaza o parte din actiunile definite in interfete, precum: Attack(), FindTarget(), etc.

Pe langa asta, cand baza jucatorului ramane fara viata si este distrusa, jocul trebuie sa se termine. Acest lucru poate fi realizat foarte usor prin suprascrierea metodei Die(). Cand baza ramane fara viata, instiintea managerul jocului, iar pe urma acesta isi porneste procesul de inchidere al nivelului. Mai multe detalii vor fi discutate in capitolul 4.1.9

BuildableTurret

BuildableTurret este clasa care mosteneste TurretBase si defineste functionalitatea pentru toate turnurile care pot fi construite pe harta de joc. Aceasta clasa implementeaza IRecoverable, care defineste modul in care turnurile pot sa isi refaca viata. De indata ce turnul a fost lovit de un inamic, se calculeaza un cost necesar pentru a reduce turnul la viata maxima. In caz ca jucatorul are suficienti bani si doreste sa refaca turnul, acesta va incepe

procesul refacere. Pe parcursul catorva secunde, se va reface treptat pentru viata lipsa in momentul inceperii acestui proces. In acest timp turnul nu poate sa atace inamicii, dar poate sa fie lovit in continuare de acestia, deci nu este garantat ca va revenii la viata maxima dupa terminarea procesului.

O alta functionalitate majora a turnurilor este optiunea de a imbunatatii turnurile. Cand utilizatorul selecteaza un turn, se cauta in fisierul configurabil costul necesar pentru a imbunatatii turnul la nivelul urmator, in caz ca nu a ajuns inca la nivelul maxim. In caz ca jucatorul are suficienti bani se apeleaza metoda Upgrade(), care verifica daca s-a ajuns la nivelul maxim si daca nu, dezactiveaza toate starile defavorabile ale turnului (cum ar fi raza turnului), creste nivelul si apeleaza metoda SetLevelProp() pentru a citi din nou fisierul de configuratie pentru nivelul nou.

Turnul poate fi vantut de jucator in caz ca este in criza de bani sau vrea sa mute turnul in alta locatie. In acest caz se calculeaza banii pe care ii primeste jucatorul, se dezactiveaza toate starile si functionalitatatile turnului si se distrug referintele specifice. Tote acestea se intampla in metoda SellTurret() implementata in aceasta clasa.

Ultima functionalitate adaugata de aceasta clasa este cea de a permite turnului sa fie controlat de comandantul jocului. Aceasta functionalitate va fi descriptia in detaliu in sectiunea 4.1.8

Turnurile specifice

Cel mai jos nivel din ierarhie presupune definirea claselor specifice pentru fiecare tip de turn in parte, fiecare dintre acestea mostenind clasa Buildable-Turret si adaugand/modificand logica stabilita pana in acest punct in functie de caz.

- **TurretElectricFence.** Este turnul care are viata foarte mare dar raza de atac foarte mica. Poate sa atace doar inamicii de lupta apropiata si actioneaza drept un zid care pazeste toate celelalte turnuri de atacurile inamicilor. Aceast turn modifica functionalitatea prin care se gaseste ce inamic trebuie sa atace si modul in care ataca inamicul.
- **TurretExcavator.** Acest turn este cel mai diferit de norma, in sensul

in care nu poate ataca deloc inamicii. Este un turn care la un anumit interval de timp farmeaza resurse, pe care le converteste la bani de joc. Din acest motiv a fost nevoie sa scape complet de cautarea inamicilor, iar metoda de atac a suprascris-o in functionalitatea de farmare de bani.

- **TurretFlamethrower.** De indata ce un inamic intra in raza acestui turn, lanseaza flacari violente in directia inamicului, flacari care ranesc toti inamicii care stau in ele. Turnul prioritizeaza inamicii de atac de aproape, dar in cazul in care nu exista astfel de inamici, ataca inamicii cu raza de atac. Nu poate sa atace inamicii zburatori sau cei bombardieri.
- **TurretMachineGun.** Este turnul universal care poate sa atace toti inamicii jocului. Acesta lanseaza gloante foarte rapid catre inamici, dar gloantele individual nu provoaca foarte multe daune. Fiecare nivel presupune o viteza de atac mai mare.
- **TurretLaser.** Acest turn ataca inamicii cu un laser foarte puternic care distrugе componzitia moleculara a inamicului in fiecare clipa in care acestia sunt in raza turnului. Poate ataca inimicii de atac apropiat si cei de la distanta, dar ii prioritizeaza pe cei de la distanta. Nu poate ataca inamicii zburatori sau pe cei bombardieri.
- **TurretVulkan.** Este turnul special impotriva inamicilor zburatori si a celor bombardieri. Nu poate ataca inamicii de sol, dar in schimb lanseaza rachete catre cei zburatori, rachete care explodeaza cand ajung in contact cu acestia.

4.1.8 Comandantul

Comandantul este un caracter special care poate fi miscat oriunde dorim pe harta si care ataca inamicii din raza lui de atac, cat timp nu se deplaseaza catre o noua locatie.

In acest scop, clasa Commander implementeaza interfetele IMovable si IAttacker|EnemyBase|. Comandantul nu poate sa fie distrus si din acest motiv nu este nevoie implementarii interfetei IDestroyable.

Comandantul are si el un fisier de configurare care specifica statusurile lui, cum ar fi viata, viteza de miscare, puterea de atac, etc. Pentru a se putea deplasa pe harta, are nevoie sa faca referire la sistemul de navigare si la cel al hartii de joc (grid system). Din acest motiv este necesar sa mosteneasca LogicProcessBase, ca sa isi execute initializarea doar dupa celelalte sisteme au reusit sa si-o termine pe a lor.

Majoritatea functiilor sunt similare cu cele ale turnurilor (Attack, DrawRange, FindTarget, etc.) asa ca nu vor fi reluate aici.

Pe langa aceste functionalitati, comandantul poate sa intre in turnuri si sa le controleze, marindu-le astfel productivitatea. Cand acesta intra in turnuri, le maresteste raza, viteza, puterea de atac si in cazul excavatorului de resurse, creste banii primiti la fiecare livrare de resurse. Ca sa poata intra in turnuri, trebuie sa se deplaseze la acel turn la comanda jucatorului, iar cand turnul este distrus/vandut, acestaiese din turn si se misca catre cel mai apropiat nod din graful hartii de joc care nu este deja ocupat. Comandantul poate sa iasa din turn si fara ca acesta sa fie distrus, dar acest lucru se intampla numai la comanda jucatorului.

4.1.9 Controlul turnurilor si starilor de joc

4.2 Selectarea nivelerelor

4.2.1 Structura meniului

4.2.2 Generarea dinamica a meniului

4.3 Salvarea datelor

4.3.1 Salvarea automata

4.3.2 Criptarea si decriptarea datelor

4.4 Magazinul de imbunatatiri permanente

4.4.1 Structura meniului

4.4.2 Generarea dinamica a meniului

4.5 Sistemul co-op

4.5.1 Modul de lucru cu Photon Engine

4.5.2 Inregistrare si camera de asteptare

4.5.3 Sincronizarea datelor

4.5.4 Monstrul de foc

4.6 Metode de imbunatatire a proiectului

4.6.1 Arta imbunatatita

4.6.2 O poveste pentru joc

4.6.3 Monstrii multipli pentru modul co-op

5 Concluzii

References

- [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 retele. 2006