Izrada igre obrane tornjevima u programskom alatu Unity

Keretić, Mateo

Undergraduate thesis / Završni rad

2021

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:886754

Rights / Prava: Attribution-NoDerivs 3.0 Unported/Imenovanje-Bez prerada 3.0

Download date / Datum preuzimanja: **2024-03-28**

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Repository / Repozitorij:

Faculty of Organization and Informatics - Digital Repository

zir.nsk.hr

DIGITALNI AKADEMSKI ARHIVI I REPOZITORIJI

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Mateo Keretić

IZRADA IGRE OBRANE TORNJEVIMA U PROGRAMSKOM ALATU UNITY

ZAVRŠNI/DIPLOMSKI RAD

Varaždin, 2020.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Mateo Keretić

JMBAG: 0016131843

Studij: Informacijski sustavi

IZRADA IGRE OBRANE TORNJEVIMA U PROGRAMSKOM ALATU UNITY

ZAVRŠNI/DIPLOMSKI RAD

Mentor/Mentorica:

doc. dr. sc. Mladen Konecki

Varaždin, rujan 2020

Mateo Keretić

Izjava o izvornosti

Izjavljujem da je moj završni/diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

U ovom radu biti će implementirana i opisana računalna igra obrane tornjevima u programskom alatu Unity. Na početku će ukratko biti opisan programski alat Unity na čiji pogon radi čak 50% računalnih igara (Unity, 2020). Zatim će biti opisan žanr igara obrane tornjevima, žanr u kojem neprijatelji šeću po unaprijed određenom putu, a igrač gradi tornjeve oko tog puta s ciljem da ubi neprijatelje prije nego stignu do kraja puta. Postoji više vrsta neprijatelja (brzi, spori, snažni, slabi..) te više vrsta tornjeva od kojih je svaki toranj moguće nadograditi i prodati. Uz to, igrač ima na raspolaganju i jednu čaroliju koja mu pomaže u obrani. Nakon opisa žanra biti će opisane sve metode, algoritmi i funkcionalnosti igre te kako su one ostvarene u programskom alatu Unity. Pisani dio rada služit će kao dokumentacija za računalnu igru „FoiTD“.

Ključne riječi: Unity, tornjevi, neprijatelji, algoritmi, programiranje

Sadržaj

1. Uvod 1
2. Alati 2
   2.1. Unity 2
   2.2. Microsoft Visual Studio 2
   2.3. GIMP 3
   2.4. Unity Asset Store 3
3. Obrana tornjevima „FoiTD“ 4
   3.1. Žanr obrane tornjevima 4
   3.2. Glavni izbornik 5
   3.3. Staza 6
   3.4. Svojstva igrača 8
   3.5. Neprijatelji 8
      3.5.1. Upravljanje neprijateljima 9
      3.5.2. Kretanje neprijatelja 11
      3.5.3. Vrste neprijatelja 13
      3.5.4. Kreiranje valova neprijatelja 13
   3.6. Tornjevi 16
      3.6.1. Ažuriranje mete i rotacija tornjeva 17
      3.6.2. Upravljanje projektilima 21
      3.6.3. Vrste tornjeva 23
      3.6.4. Upravljanje tornjevima 25
         3.6.4.1. Shematski plan (*blueprint*) tornja 25
         3.6.4.2. Upravljanje izgradnjom tornjeva 25
         3.6.4.3. Čvorovi za izgradnju tornjeva 27
         3.6.4.4. Trgovina tornjeva 30
         3.6.4.5. Nadograđivanje i prodavanje tornjeva 32
4. Čarolija 36
5. Upravljanje završetkom igre 38
6. Isporuka igre 40
7. Zaključak 41

Popis literature 42

Popis slika. 43

Prilozi 40

1. Uvod

Danas, više nego ikad, računalna industrija postaje jedna od glavnih, ako ne i glavna, grana zabavne industrije. Način na koji korisnici vrše interakciju sa igrama i kroz igre se konstantno mijenja. Ovo, ne samo da rezultira u većoj općenitoj angažiranosti prema igrama, nego vodi i do potpuno novih segmenta entuzijasta za računalne igre. Samo u 2019. godini generirano je više od 152 bilijuna dolara te je dostignut broj od 2.5 bilijuna igrača (Wijman, 2019).

Na tržištu računalnih igara postoji mnogo različitih žanrova, od akcijskih igara, preko avanturističkih igara pa sve do simulacijskih igara. U ovom radu fokusira se na jedan specifičan žanr, žanr strateških igara. Točnije, fokusira se na jedan podžanr strateških igara koji se naziva žanr obrane tornjevima.

Cilj ovog rada je upoznati čitatelja sa žanrom obrane tornjevima te analizirati i prikazati kako implementirati jednu računalnu igru tog žanra. U sklopu rada, biti će implementirana igra obrane tornjevima u programskom alatu Unity. Rad se sastoji od pisanog i praktičnog dijela. Pisani dio sastoji se od opisa programskog alata Unity i žanra obrane tornjevima. Nakon toga slijedi opis računalne igre koja će se izraditi te opis algoritama koji će se koristiti u kreiranoj računalnoj igri. Praktični dio sastoji se od funkcionalnog prototipa računalne igre implementirane u programskom alatu Unity.

Inspiracija za odabir ove teme bila je moja zainteresiranost za računalne igre još od djetinjstva, a pogotovo za igre obrane tornjevima koje predstavljaju tip igre u koje se svatko može upustiti, od najvještijih igrača pa i do onih manje vještih igrača. Uvijek mi je bio primamljiv tip računalnih igara u kojima igrač mora uložiti vrijeme razmišljajući o optimalnoj strategiji i taktici. Ova tema bila je pravi odabir za mene jer sam želio saznati više o tome kako nastaju takve računalne igre.

2. Alati

Za implementaciju igre obrane tornjevima korišten je programski alat Unity, razvojno okruženje Microsoft Visual Studio te alat za manipulaciju slikama GIMP. Svi grafički elementi i 3D modeli preuzeti su sa Unity Asset Store web trgovine. Sve navedene tehnologije bit će opisane u nastavku poglavlja.

2.1. Unity

U prošlosti, većina tvrtki koje su razvijale videoigre morale su napraviti vlastiti game engine koji bi pokretao komponente njihovog projekta. To je koštalo novaca, ali i uvijek potrebitog programerskog vremena. Ubrzo se javila potreba za nekim univerzalnijim rješenjem, a jedno od najboljih je ponudio Unity Technologies 2005. godine. Zamišljen kao radionica u kojoj se programer bavi igrom, a ne onime što je pokreće. Baziran na C# programskom jeziku, sustav omogućuje izradu impresivnih 2D i 3D igara koje možemo bez ustručavanja uvrstiti unutar najviših standarda u današnjoj industriji (Machina, 2020).

Od 2018. godine, Unity podržava više od 20 platformi te se može koristiti za izradu trodimenzionalnih, dvodimenzionalnih, virtualno realnih i prividno realnih igara kao i za simulacije i razne druge mogućnosti. U zadnje vrijeme, Unity game engine je prisvojen i od industrija izvan industrije računalnih igara kao što su filmska, automobilska, inženjerska i konstrukcijska industrija.(Unity, 2020).

U razvoju 2D igra, Unity omogućuje lagano uvoženje dvodimenzionalnih elemenata i pruža napredni dvodimenzionalni renderer pomoću kojeg je lako postaviti elemente u smislenu cjelinu. Projekti trodimenzionalne prirode mogu računati na mapiranje ispupčenja i refleksija, napredna sjenčanja i slične efekte te mnogobrojne opcije za upravljanje fizikom u igri. Unity software besplatan je za osobno korištenje, uz minimalna ograničenja što rezultira ogromnom rasprostranjenošću i popularnosti alata.

2.2. Microsoft Visual Studio

Programski jezik za pisanje skripti u programskom alatu Unity je C# i piše se u Microsoft Visual Studiu. Microsoft Visual Studio je integrirano razvojno okruženje kreirano od strane Microsofta. Instalacija Unity alata automatski pruža dodatak za kompatibilnost sa Visual Studiom. Svaka Unity skripta koja se piše u Visual Studiu nasljeđuje klasu MonoBehaviour

koja je bazna klasa za sve Unity skripte te omogućuje uporabu raznih metoda koje su potrebne za izradu računalnih igara.

Popis nekih važnih metoda klase MonoBehaviour korištenih u izradi igara (Unity Manual, 2020) :

- Invoke – Pobuđuje neku metodu u određenom vremenu
- InvokeRepeating – Pobuđuje neku metodu u određenom vremenu te ju ponavlja svaki željeni broj sekundi
- StartCoroutine – Pokreće korutinu
- Awake – Metoda koja se poziva kada se skripta pokreće
- OnMouseEnter – Poziva se kada korisnik uđe sa mišom unutar objekta
- OnMouseExit – Poziva se kada korisnik izađe mišom izvan objekta
- Update – Poziva se jednom za svaki okvir
- Start – Poziva se kada je skripta učitana, prije prvog poziva metode Update
- I još mnoge druge…

2.3. GIMP

GIMP je najpopularnija slobodna aplikacija otvorenog koda za stvaranje i obradu rasterske grafike. Koristi se za retuširanje i uređivanje slika, slobodno crtanje, mijenjanje veličine i obrezivanje slika, pretvaranje istih u različite formate i mnoge druge specijalizirane zadatke (GIMP, 2020).

U ovom radu, GIMP je pretežno korišten za ukidanje pozadine raznih slika te za manipuliranje raznim teksturama.

2.4. Unity Asset Store

Unity Asset Store je rastuća trgovina i biblioteka modela, zvukova, dodataka, predložaka, efekata, tekstura, materijala i sl. koja omogućuje svojim članovima kreiranje vlastite imovine i njihovu objavu na stranici trgovine. Na stranici trgovine postoje i besplatna i plaćena imovina koju članovi mogu preuzeti i koristiti direktno u Unity alatu za svoje računalne igre i projekte.

U ovom radu svi modeli, animacije, efekti, materijali i teksture preuzeti su sa Unity Asset trgovine (FullTiltBoogie, 2018, VSQUAD, 2019, PressSTART, 2018, PI, 2017).

3. Obrana tornjevima „FoiTD“

U ovom poglavlju bit će opisan žanr obrane tornjevima i opis igre obrane tornjevima FoiTD te način njezine implementacije u programskom alatu Unity.

3.1. Žanr obrane tornjevima

Računalne igre obrane tornjevima su podžanr strateških igara čiji je primarni fokus na optimalnoj alokaciji resursa i optimalnom pozicioniranju tornjeva.

U najjednostavnijoj formi, igra obrane tornjevima sastoji se od ljudskog igrača koji kupuje i postavlja razne vrste defanzivnih tornjeva koji pucaju na valove neprijatelja različitih vrsta koji se kreću po nekom unaprijed određenom putu. Igrač dobiva određenu svotu novca za svakog neprijatelja kojeg njegov toranj uništi, a svota novca koju igrač dobiva ovisi o snazi poraženog neprijatelja. Ako igrač uspije uništiti sve neprijatelje u nekom valu, igrač pobjeđuje taj val i nastavlja graditi svoju obranu za slijedeći val. Ako igrač prođe sve valove na nekoj razini tada on pobjeđuje tu razinu i može nastaviti na slijedeću. Na početku svake razine igrač ima određen broj životnih bodova, a svaki protivnik koji stigne do kraja puta bez da bude uništen smanjuje broj životnih bodova igrača. Ako životni bodovi igrača padnu na nulu tada je igrač poražen te mora ispočetka pokušati uspješno proći razinu. Svaka razina najčešće ima između 5-15 valova neprijatelja koje igrač mora poraziti te se svakim valom pojačava ukupna težina neprijatelja. Igrač upravlja svojom obranom kupovanjem, postavljanjem, unapređivanjem i prodajom borbenih tornjeva koji nakon postavljanja automatski pucaju prema neprijateljima.

Igre obrane tornjevima pokazale su se kao izazovne igre i zabavan način da prođe vrijeme. Jednostavnost igranja te zanimljive i izazove strategije u kombinaciji sa velikim brojem različitih igara u žanru donijele su veliku populaciju igrača igra obrane tornjevima.

Žanr obrane tornjevima je relativno nov žanr koji je dobio na popularnosti zbog mnogih online flash inačica koje je bilo moguće igrati kroz web preglednik. Jedna od prvih komercijalnih igra obrane tornjevima bila je izdana 1990. godine na Atari platformi i zvala se Rampant. Zbog svoje popularnosti, Rampant je bila razvijana i na drugim platformama. Prateći Rampant trend, velik broj popularnih strateških igara je počeo uključivati svoje inačice obrane tornjevima u svoje igre. Najpoznatiji primjer takvog trenda su bile prilagođene igre obrane tornjevima kreirane u igri Warcraft III: The Frozen Throne. Koristeći uređivač mapa dostupan u Warcraft III igri, igrači su sami kreirali defanzivne mape koje su bile vrlo slične današnjem poimanju igre obrane tornjevima (Alistar i suradnici, 2014). Žanr obrane tornjevima sve je više rastao na

popularnosti te su kreirane na tisuće različitih inačica do danas. U slijedećim poglavljima opisat će se implementacija jednostavne igre obrane tornjevima od početka do kraja.

3.2. Glavni izbornik

Igra FoiTD sastoji se od dvije scene. Scena u Unity alatu predstavlja jedinstvenu razinu u koju kreatori postavljaju svoju okolinu, objekte, dekoracije, odnosno gdje grade i dizajniraju svoju igru u dijelovima.

Prva scena predstavlja glavni izbornik koji se prikazuje kada se igra pokrene te u kojem igrač može pokrenuti prvu razinu igre ili izaći iz aplikacije.



Slika 1. Glavni izbornik

Skripta koja se nalazi na glavnom izborniku je MainMenu.cs.

```
public class MainMenu : MonoBehaviour
{
    public string levelToLoad = "MainLevel";

    public void StartGame()
    {
        SceneManager.LoadScene(levelToLoad);
    }

    public void QuitGame()
    {
        Application.Quit();
    }
}
```

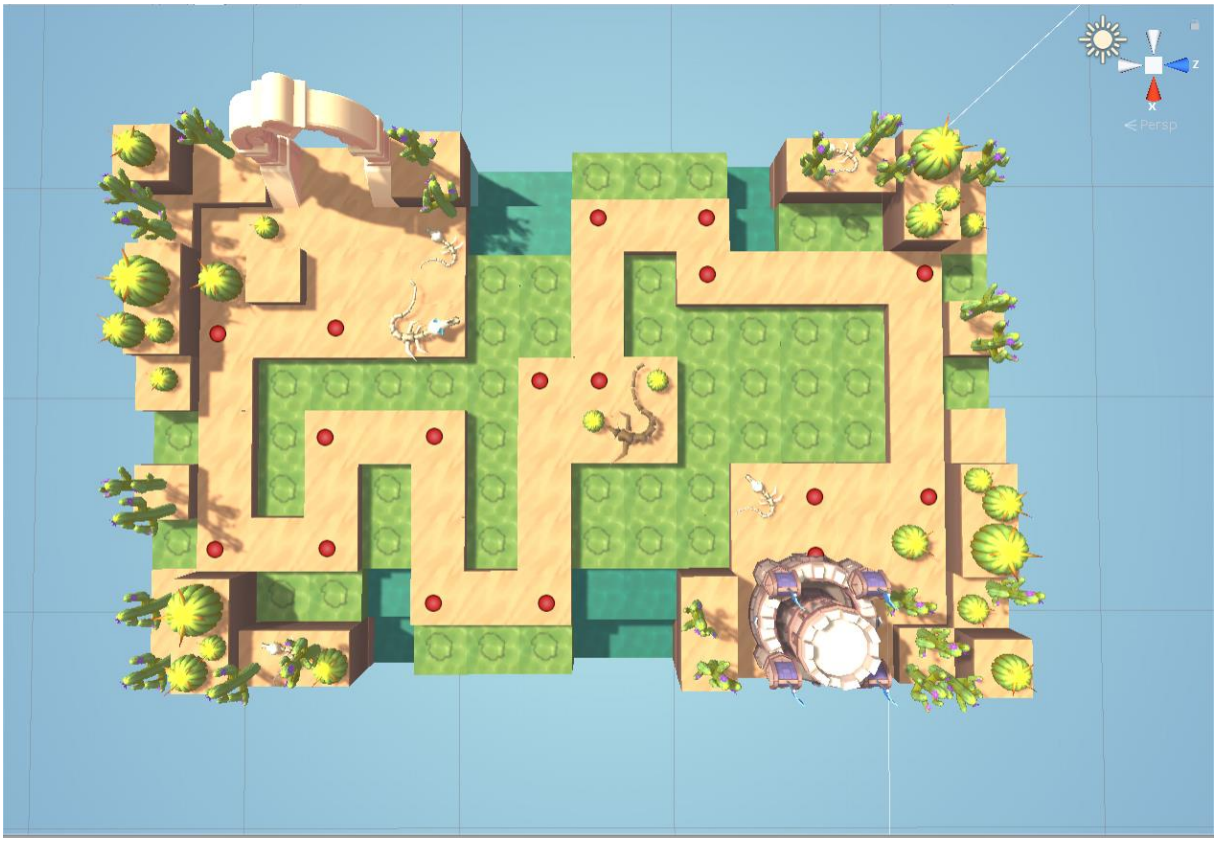
Na početku skripte deklariramo javnu varijablu tipa String koja nam služi za postavljanje imena razine koju želimo učitati kada korisnik pritisne gumb „Započni“ i dajemo joj općenitu vrijednost. U Unity alatu svim javnim varijablama možemo dodijeliti vrijednost putem inspektora koji sadržava sve skripte, zvukove, transformaciju i ostale grafičke elemente objekta.

Kreiramo funkciju StartGame() koja će se pokrenuti kada korisnik pritisne na gumb „Započni“. Ona poziva metodu SceneManager.LoadScene, metoda koja sam služi za manipuliranje scenama. Kada se metoda pozove tada se pokreće scena proslijeđena u argumentu funkcije, u našem slučaju prva razina igre.

U funkciji QuitGame, koja se poziva kada korisnik pritisne gumb „Izađi“ na glavnom izborniku, pozivamo metodu Application.Quit koja nam služi za izlazak iz naše aplikacije.

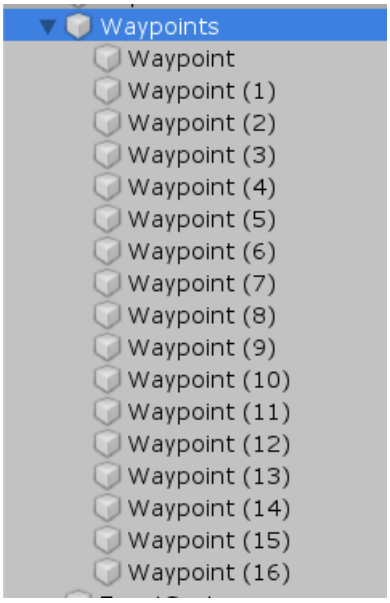
3.3. Staza

U našoj glavnoj sceni nalazi se prva razina igre. Nakon što smo preuzeli i prilagodili teren sa Unity Asset trgovine, potrebno je označiti put kojim će neprijatelji hodati.



Slika 2. Staza

Postavljamo prazne objekte na mapu, označene crvenim točkama na slici, koji nam označavaju željeni put neprijatelja. Nakon što smo postavili željene prazne objekte, kreiramo još jedan novi prazni objekt kojem će svi prethodno kreirani objekti puta biti djeca.



Slika 3. Objekti puta

Sada tom posljednje kreiranom objektu „Waypoints“ dodajemo skriptu u kojoj ćemo sve objekte puta uvrstiti u polje.

```
public class Waypoints : MonoBehaviour
{
    public static Transform[] points;

    void Awake()
    {
        points = new Transform[this.transform.childCount];

        for (int i = 0; i < points.Length; i++)
        {
            points[i] = transform.GetChild(i);
        }
    }
}
```

Skripta Waypoints.cs služi nam za kreiranje tog polja. Kreiramo statičnu varijablu „points“ tipa Tansform polja. Svaki objekt u sceni ima svoj Transform tip koji pohranjuje njegovu poziciju, rotaciju i razmjer. Statična varijabla označava da će sve instance klase Waypoints dijeliti tu varijablu i sve skripte koje pristupaju toj varijabli vidjet će istu vrijednost.

U metodi Awake, koja se pokreće kada se pokreće skripta, tom polju „points“ alociramo memoriju veličine broja djece koju objekt sadržava, tj. veličine ukupnog broja naših objekata puta. Ključnu riječ „this“ koristimo kada želimo pristupiti trenutnoj instanci klase, tj. trenutnom objektu u kojem se skripta nalazi. Prolazimo kroz sve objekte puta te ih po redu dodajemo u

prethodno alocirano polje. Sada imamo naše polje koje nam predstavlja put po kojem će se neprijatelji kretati.

3.4. Svojstva igrača

Prije nego krenemo na implementaciju kretnje neprijatelja po prethodno kreiranom putu kreirat ćemo skriptu PlayerStats.cs kojom ćemo pratiti svojstva igrača.

```
public class PlayerStats : MonoBehaviour
{
    public static int money;
    public int startMoney = 400;

    public static int lives;
    public int startLives = 20;

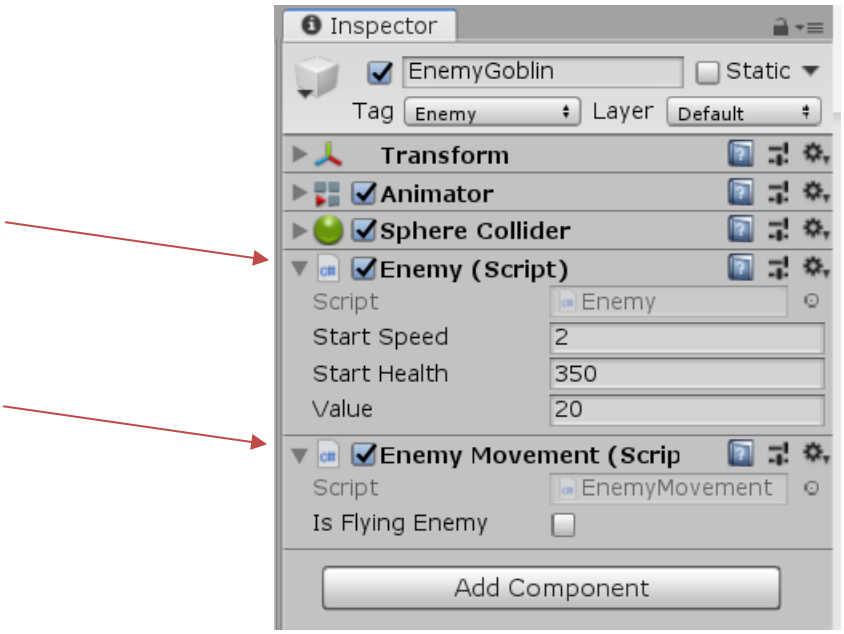
    void Start()
    {
        money = startMoney;
        lives = startLives;
    }
}
```

Deklariramo javne varijable za početni novac i početni broj životnih bodova igrača i dodjeljujemo im općenitu vrijednost te javne statičke varijable kojima ćemo pratiti taj novac i životne bodove igrača kroz igru. Deklariramo ih javnim i statičkim kako bismo kroz druge skripte mogli mijenjati i koristiti vrijednost tih varijabla i sve skripte koje im pristupaju će vidjeti istu vrijednost.

U metodi Start(), na početku skripte, inicijaliziramo vrijednost statičkih varijabli na željenu početnu vrijednost novca i životnih bodova.

3.5. Neprijatelji

Kako bismo mogli manipulirati našim preuzetim modelima neprijatelja te ih prilagoditi našoj igri, na svaki objekt neprijatelja dodat ćemo dvije skripte. Skriptu Enemy.cs koja će nam služiti za manipuliranje brzine, vrijednosti i životnih bodova neprijatelja te skriptu EnemyMovement.cs koja će omogućiti kretanje neprijatelja po stazi od početka do kraja.



Slika 4. Objekt neprijatelja

Da možemo ponovno koristiti takvu vrstu objekta, moramo ga pretvoriti u tzv. „Prefab“. U alatu Unity Prefab je sistem koji nam omogućava kreiranje, konfiguraciju i pohranjivanje objekata kompletno sa svim svojim komponentama, svojstvima, vrijednostima i objektima djece kao ponovno iskoristivu imovinu. Bilo koju promjenu koju napravimo na Prefabu automatski se primjenjuje na sve instance tog Prefaba koje kreiramo što nam omogućava jednostavnu ponovnu iskoristivost objekta (Unity Manual, 2020). Naš Prefab neprijatelja sastojat će se od preuzetih modela neprijatelja na koje ćemo dodati dvije skripte Enemy.cs i EnemyMovement.cs i moći ćemo ih ponovno koristiti kada nam zatreba.

3.5.1. Upravljanje neprijateljima

Kako bismo mogli upravljati našim neprijateljima biti će nam potrebna skripta Enemy.cs koju ćemo postaviti na svaki objekt neprijatelja kako bismo upravljali brzinom, vrijednošću i životnim bodovima neprijatelja.

```
[HideInInspector]
public float speed;
public float startSpeed = 10f;
private float health;
public float startHealth = 100;
public int value = 20;

void Start()
{
	health = startHealth;
	speed = startSpeed;

}
```

Na početku skripte deklariramo varijable za brzinu, početnu brzinu, životne bodove, početne životne bodove i varijablu za vrijednost neprijatelja. U Start() metodi dodjeljujemo vrijednost varijablama za životne bodove i brzinu. Varijabla za brzinu „speed“ je javna jer će nam biti potrebna u drugim skriptama te poprima svojstvo HideInInspector kojim onemogućujemo mijenjanje te varijable kroz inspektor.

```
public void TakeDamage (float damageAmount)
{
     health -= damageAmount;

     if (health <= 0)
     {
         Die();
     }
}
```

Nadalje u skripti kreiramo javnu metodu kojom ćemo kontrolirati životne bodove neprijatelja. Metoda prima argument koji označava koliko životnih bodova će neprijatelj izgubiti. Smanjujemo životne bodove neprijatelja za taj iznos te provjeravamo da li su životni bodovi neprijatelja pali na nulu ili ispod nule, ako jesu tada pozivamo metodu koja uništava protivnika Die().

```
void Die()
{
     Destroy(gameObject);
     PlayerStats.money += value;
}
```

Metoda Die() uništava objekt neprijatelja te uvećava novac igrača za vrijednost uništenog neprijatelja.

```
public void SlowDownEnemy(float slowPercentage)
{
    StartCoroutine(SlowEnemy(slowPercentage));
}

IEnumerator SlowEnemy(float slowPercentage)
{
    speed = startSpeed * (1f - slowPercentage);
    yield return new WaitForSeconds(0.5f);
    speed = startSpeed;
}
```

U skripti Enemy.cs imamo još jednu metodu SlowDownEnemy koja nam služi za manipuliranje brzine neprijatelja. Kasnije, imat ćemo vrstu tornja koji, kada puca, usporava protivnike i onda ćemo pozivati ovu metodu. Metoda SlowDownEnemy pokreće korutinu SlowEnemy. Korutina je posebna vrsta funkcije koja može pauzirati svoje izvođenje naredbom „yield return“ i pauzira se sve dok ne istekne dano vrijeme te zatim nastavlja svoje izvođenje.

Metoda korutine SlowEnemy smanjuje brzinu neprijatelja ovisno o postotku usporavanja te zatim pauzira svoje djelovanje na pola sekunde sa naredbom yield return new WaitForSeconds(0.5f). Kada istekne pola sekunde brzina neprijatelja se vraća na početnu vrijednost. Ovime smo ostvarili efekt usporavanja neprijatelja na pola sekunde svaki put kada se pozove ova korutina.

3.5.2. Kretanje neprijatelja

Nakon što smo kreirali polje puta po kojem će se naši neprijatelji kretati, sada trebamo napraviti skriptu koja će omogućiti to kretanje po putu. To ćemo ostvariti u skripti EnemyMovement.cs koju ćemo postaviti na svaki objekt neprijatelja.

```
private Transform target;
private int waypointIndex = 0;
public Vector3 positionOffset;
private Enemy enemy;
```

U deklariranju varijabla skripte kreiramo Transform varijablu „target“ koja će nam označavati poziciju u sceni slijedeće mete koju neprijatelj treba pratiti, postavljamo indeks polja na početak, te koristimo jednu bool varijablu koja nam kaže da li je neprijatelj letećeg tipa ili ne. Leteći protivnici ne prate stazu nego lete ravno prema odredištu. Također, treba nam još jedna varijabla tipa Enemy preko koje ćemo dohvatiti brzinu neprijatelja.

```
void Start()
{
    enemy = this.GetComponent<Enemy>();
    if (isFlyingEnemy)
    {
        target = Waypoints.points[Waypoints.points.Length - 1];
    }
    else
    {
        target = Waypoints.points[0];
    }
 }
```

U Start() metodi prvo dodjeljujemo varijabli „enemy“ vrijednost komponente koja se nalazi na objektu neprijatelja, a tipa je Enemy, za to nam služi GetComponent() funkcija. Zatim dodjeljujemo vrijednost varijabli „target“ ovisno o tome da li je neprijatelj letećeg tipa ili ne. Ako je neprijatelj leteći tada „target“ poprima vrijednost posljednjeg elementa u polju Waypoints.points, a inače poprima vrijednost prvog elementa u tom polju. Polje points objašnjeno je u poglavlju 3.3.

```
void Update()
{
    Vector3 direction = target.position - transform.position;
    transform.Translate((direction.normalized) * enemy.speed *
                        Time.deltaTime, Space.World);
```

```
    Quaternion lookRotation = Quaternion.LookRotation(direction);
    float step = 360 * Time.deltaTime;
    transform.rotation = Quaternion.RotateTowards(transform.rotation,
                         lookRotation, step);

    if (Vector3.Distance(transform.position, target.position) <=
        0.2f)
    {
        GetNextWaypoint();
    }
}
```

U Update() metodi, koja se poziva svaki okvir, vršimo kretanje protivnika prema slijedećoj meti. Kreiramo vektor smjera tako da oduzmemo poziciju mete sa trenutnom pozicijom neprijatelja te zatim pozivamo metodu transform.Translate koja objekt neprijatelja translatira određenom brzinom prema toj meti. Kao prvi argument funkcije, postavljamo normaliziranu vrijednost našeg vektora smjera pomnoženu sa brzinom neprijatelja te množimo sa Time.deltaTime. Time.deltaTime je posebna varijabla u Unity-u koja nam prikazuje vrijeme prošlo između trenutnog i prethodnog okvira. Ona nam omogućuje da objekte mijenjamo ili mičemo prema vremenu, a ne prema broju okvira po sekundi koje računalo generira što rezultira jednakim rezultatom neovisno o računalu na kojem se igra pokreće.

Kao drugi argument funkcije postavljamo Space.World, a to nam označava da želimo da se objekt transformira koristeći koordinate cijele scene, a ne koristeći svoje lokalne koordinate. Kako bi kretanje našeg neprijatelja izgledalo prirodno trebamo i njegovo tijelo rotirati prema meti, to radimo pomoću objekata tipa Quaternion. Kvaternion pruža matematičku reprezentaciju orijentacije i rotacije objekta u trodimenzionalnom svijetu. Koristimo metodu Quaternion.RotateTowards koja nam omogućuje uglađeno rotiranje objekta, a kao parametar prima trenutnu rotaciju objekta, rotaciju prema kojoj želimo da se objekt rotira te brzinu rotiranja.

Zatim, svaki okvir provjeravamo da li je daljina između mete i neprijatelja manja od 0.2f pozivajući metodu Vector3.Distance koja vraća daljinu između dva vektora i prosljeđujemo joj pozicije naših objekata mete i neprijatelja. Ovdje moramo koristiti 0.2f kao daljinu jer želimo neprijatelja preusmjeriti kada se dovoljno približi meti. Kada se dovoljno približi meti pozivamo metodu GetNextWaypoint().

```
void GetNextWaypoint()
{
    if (waypointIndex >= Waypoints.points.Length - 1)
    {
        CastleReached();
        return;
    }

    waypointIndex++;
    target = Waypoints.points[waypointIndex];
}
```

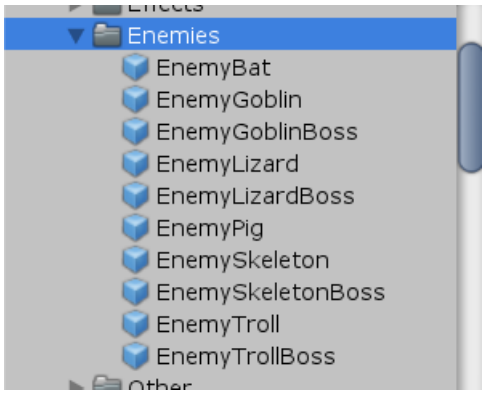
Kada se neprijatelj dovoljno približi meti provjeravamo da li je ta meta posljednja, ako da onda završavamo put metodom CastleReached(), a inače povećavamo indeks polja za jedan te nova meta neprijatelja postaje slijedeći element u polju „Waypoints.points“.

```
void CastleReached()
{
    PlayerStats.lives--;
    Destroy(gameObject);
}
```

Ako neprijatelj stigne do kraja puta bez da je uništen poziva se metoda CastleReached() koja smanjuje životne bodove igrača te uništava objekt neprijatelja koji je stigao do cilja.

3.5.3. Vrste neprijatelja

U našoj igri kreirat ćemo deset različitih vrsta neprijatelja, od kojih će jedna vrsta biti leteći neprijatelj.



Slika 5. Vrste neprijatelja

Svaku od deset vrsta neprijatelja pretvorit ćemo u Prefab koji se sastoji od modela neprijatelja te skripti Enemy.cs i EnemyMovement.cs kako bismo ostvarili ponovnu iskoristivost objekta. Svaka vrsta neprijatelja razlikuje se po brzini, broju životnih bodova te vrijednosti. Sada svaki put kada stvorimo objekt neprijatelja on će se automatski kretati po stazi te ćemo moći upravljati njegovim svojstvima.

3.5.4. Kreiranje valova neprijatelja

Sada, kada se naši neprijatelji znaju kretati stazom te možemo upravljati njihovim statusom, trebamo kreirati skriptu koja će stvarati neprijatelje raznih vrsta u valovima. Za to nam prvo treba jedna pomoćna klasa Wave.cs.

```
[System.Serializable]
public class Wave
{
```

```
    [System.Serializable]
    public class WaveGroup
    {
        public GameObject enemyPrefab;
        public int count;
        public float timeBetweenEnemies;
    }

    public WaveGroup[] waveGroup;
}
```

Nad ovom klasom postavili smo svojstvo System.Seriazable koje nam omogućava da inicijaliziramo tu klasu u inspektoru. Klasa Wave sastoji se od još jedne pomoćne klase WaveGroup sa svojstvom System.Serializable koja sadržava željeni Prefab neprijatelja, broj neprijatelja te vrste koje želimo stvoriti u valu te vrijeme između stvaranja dva takva neprijatelja.

Nakon klase WaveGroup kreiramo još jednu varijablu koja je tipa polja klase WaveGroup zato što želimo u našem valu stvarati više vrsta protivnika pa nam je potrebno polje. Svaki element polja sastoji se od neprijatelja kojeg želimo stvoriti, broj njihovih instanci koliko ih želimo stvoriti te vremenski razmak između stvaranja dva takva neprijatelja.

Sada, kada možemo kreirati polje koje predstavlja jedan val naših protivnika, trebamo kreirati skriptu koja će omogućiti kreiranje više takvih valova te ostvariti instanciranje protivnika na željenoj poziciji. To ćemo ostvariti u skripti WaveSpawner.cs.

```
public Transform spawnPoint;
public Wave[] waves;
private int waveIndex;
public static int enemiesAlive;
public static bool levelComplete;
```

Na početku skripte deklariramo varijablu tipa Transform koja će nam označavati poziciju i rotaciju na kojoj želimo stvoriti neprijatelje. Zatim kreiramo polje tipa Wave kojem je svaki element polja polje tipa WaveGroup prethodno objašnjeno. Nadalje, deklariramo varijablu tipa int koja predstavlja trenutnu poziciju u polju tipa Wave. Nadalje, deklariramo dvije javne statične varijable koje predstavljaju trenutni broj živih neprijatelja te da li je razina završena.

```
void Start()
{
    enemiesAlive = 0;
    waveIndex = 0;
    levelComplete = false;
}
```

U metodi Start() inicijaliziramo broj trenutno živih protivnika na nulu, indeks polja postavljamo na početak te postavljamo da razina nije završena.

```
int GetNumberOfEnemiesThisWave(Wave wave)
{
    int totalNumOfEnemiesThisWave = 0;
```

```
    foreach (Wave.WaveGroup waveGroup in wave.waveGroups)
    {
        totalNumOfEnemiesThisWave += waveGroup.count;
    }
    return totalNumOfEnemiesThisWave;
}
```

U skripti se također nalazi funkcija GetNumberOfEnemiesThisWave koja kao argument prima val neprijatelja tipa Wave. Na početku funkcije deklariramo broj protivnika ovog vala na nulu. Tada foreach petljom prolazimo kroz svaku valnu grupu u polju valnih grupa waveGroups iz klase Wave te nadodajemo varijabli „totalNumOfEnemiesThisWave“ broj protivnika trenutne valne grupe. Na kraju vraćamo taj dobiveni broj koji nam predstavlja ukupan broj protivnika koji će se stvoriti u cijelom proslijeđenom valu.

```
IEnumerator SpawnWave()
{
    Wave wave = waves[waveIndex];

    enemiesAlive = GetNumberOfEnemiesThisWave(wave);

    foreach (Wave.WaveGroup waveGroup in wave.waveGroups)
    {
       for (int i = 0; i < waveGroup.count; i++)
       {
         Instantiate(waveGroup.enemyPrefab, spawnPoint.position,
                     spawnPoint.rotation);

         yield return new
         WaitForSeconds(waveGroup.timeBetweenEnemies);
       }
    }
    waveIndex++;
}
```

Funkcija SpawnWave() služit će nam za stvaranje našeg vala protivnika. Na početku dohvaćamo val koji želimo stvoriti u varijablu „wave“, tj. prosljeđujemo element našeg polja valova „waves“ na poziciji na kojoj se trenutno nalazimo „waveIndex“. Tada u varijablu „enemiesAlive“ dohvaćamo ukupan broj protivnika koje ćemo stvoriti ovaj val sa funkcijom GetNumberOfEnemiesThisWave kojoj prosljeđujemo trenutni val „wave“.

Sada, prolazimo kroz svaku valnu grupu u polju valnih grupa naše varijable „wave“ i za svaku valnu grupu ulazimo u for petlju koja traje sve dok ne stvorimo željeni broj protivnika „waveGroup.count“ koji se nalazi u toj grupi. Sve dok nismo dostigli željeni broj neprijatelja, pozivamo funkciju Instantiate koja služi za instanciranje objekta u sceni, a kao argumente prima objekt koji želimo instancirati (u našem slučaju to je Prefab odabranog neprijatelja), poziciju na kojoj želimo instancirati objekt te rotaciju na kojoj želimo instancirati naš objekt (to smo postavili varijablom tipa Transform „spawnPoint“ koja se inicijalizira u inspektoru). Nakon što je neprijatelj stvoren pozivamo „yield return new WaitForSeconds(waveGroup.timeBetweenEnemies)“ koja pauzira našu funkciju na

proslijeđeni broj sekundi, a prosljeđujemo vrijeme koje smo postavili da mora proći između stvaranja dva neprijatelja „waveGroup.timeBetweenEnemies“.

Nakon što završi for petlja, foreach petlja prelazi na slijedeću valnu grupu i tako sve dok se ne instancira cijeli val neprijatelja. Kada se instancira cijeli val neprijatelja, izlazimo iz foreach petlje te povećavamo indeks vala za jedan kako bi mogli instancirati slijedeći val.

```
void Update()
{
    if (enemiesAlive > 0)
    {
        return;
    }

    if (waveIndex >= waves.Length)
    {
       if (PlayerStats.lives > 0)
       {
           levelComplete = true;
       }
        this.enabled = false;
        return;
     }

     if (Input.GetKeyDown(KeyCode.Space))
     {
     StartCoroutine(SpawnWave());
     }
 }
```

U Update() funkciji upravljamo kada želimo da se stvori slijedeći val. Na početku provjeravamo da li još postoje živi neprijatelji te ako postoje ne želimo dopustiti stvaranje slijedećeg vala pa sa naredbom return izlazimo iz funkcije.

Ako nema više živih neprijatelja provjeravamo da li smo prošli kroz sve valove neprijatelja. Ako smo prošli sve valove i igrač ima preostalih životnih bodova tada postavljamo varijablu „levelComplete“ na true što nam označava da je igrač uspješno prošao razinu. Nakon toga onemogućujemo daljnje djelovanje ove skripte te izlazimo iz funkcije.

Ako nema živih neprijatelja i igrač još nije prešao sve valove, tada igrač može pritisnuti razmaknicu na tipkovnici kako bi stvorio slijedeći val. Input.GetKeyDown funkcija aktivira se kada korisnik pritisne proslijeđenu tipku na tipkovnici, u našem slučaju razmaknicu. Pritiskom na razmaknicu pokreće se korutina SpawnWave prethodno objašnjena koja stvara novi val neprijatelja.

3.6. Tornjevi

Nakon što stvorimo valove naših neprijatelja potrebno je omogućiti igraču kreiranje tornjeva koji će pucati po tim neprijateljima. Svaki toranj mora imati mogućnost pronalaska

najbližeg neprijatelja koji je u njegovom dosegu te pucati po neprijatelju sve dok nije uništen ili je izašao van dosega tornja. To ćemo ostvariti u dvije skripte, prva skripta Turret.cs omogućit će nam postavljanje svojstva tornja, pronalaženje najbliže mete u dosegu te rotiranje tornja u smjeru najbližeg neprijatelja. Druga skipta, Bullet.cs omogućit će nam upravljanje projektilom koji će pratiti neprijatelja kojeg je toranj naciljao te umanjivati životne bodove neprijatelja kada stigne do njega.

3.6.1. Ažuriranje mete i rotacija tornjeva

Na početku skripte Turret.cs deklariramo svojstva tornja.

```
public float range;
public float fireRate;
public bool slow;
public float slowPercentage;
private float shootRecharge = 0.1f;
```

Dekariramo javne varijable za opseg tornja, brzinu paljbe te varijable kojima označavamo da li toranj ima sposobnost usporavanja protivnika te koliko će iznositi postotak usporavanja. Deklariramo još jednu privatnu varijablu „shootRecharge“ koja će biti pomoćna i poslužit će nam za izračun vremena između paljbe svakog metka jednog tornja. Na početku je postavljamo na 0.1 jer želimo da toranj puca gotovo odmah kada se instancira.

```
private Transform target;
private Enemy targetEnemy = null;
public string enemyTag = "Enemy";
```

Nakon deklaracije svojstva tornja, deklariramo razne tehničke varijable. Varijabla „target“ predstavljat će nam poziciju neprijatelja u sceni, varijabla „targetEnemy“ predstavljat će nam komponentu tipa Enemy.cs tog neprijatelja, a „enemyTag“ nam predstavlja oznaku svih naših neprijatelja, u našem slučaju to je „Enemy“.

```
public Transform partToRotate;
public float turnSpeed;
public Transform firePoint;
public GameObject bulletPrefab;
```

Potrebno je deklarirati još pojedine varijable, a to su „partToRotate“ koja nam označava dio tijela tornja koji želimo rotirati u smjeru protivnika, „turnSpeed“ koja označava brzinu te rotacije. Zatim, „firePoint“ koja nam označava poziciju na objektu tornja iz koje ćemo instancirati projektil te „bulletPrefab“ koja nam označava naš Prefab objekt projektila kojeg ćemo instancirati kada će toranj zapucati.

```
void UpdateTarget()
{
    if (target != null && Vector3.Distance(this.transform.position,
                                           target.position) <= range)
    {
```

```
            return;
        }

        GameObject[] enemies = GameObject.FindGameObjectsWithTag(enemyTag);
        float shortestDistance = Mathf.Infinity;
        GameObject nearestEnemy = null;

        foreach(GameObject enemy in enemies)
        {
            float distanceToEnemy = Vector3.Distance(this.transform.position,
                                                     enemy.transform.position);
            if (distanceToEnemy < shortestDistance)
            {
                shortestDistance = distanceToEnemy;
                nearestEnemy = enemy;
            }
        }

        if (nearestEnemy != null && shortestDistance <= range)
        {
            target = nearestEnemy.transform;
            targetEnemy = nearestEnemy.GetComponent<Enemy>();
        }
        else
        {
            target = null;
        }
    }
```

U funkciji UpdateTarget() ažuriramo metu tornja. Na početku provjeravamo da li već postoji meta, te da li je unutar opsega tornja, ako da, onda se vraćamo iz funkcije naredbom return jer ne želimo da toranj mijenja metu ako mu je trenutna meta unutar opsega.

Ako toranj nema metu ili je trenutna meta izašla izvan opsega kreiramo polje objekata neprijatelja „enemies“ u kojem dohvaćamo neprijatelje koji se trenutno nalaze u sceni sa funkcijom GameObject.FindGameObjectsWithTag koja vraća polje svih objekata u sceni sa proslijeđenom oznakom, a mi prosljeđujemo našu oznaku neprijatelja.

Kreiramo varijablu koja označava daljinu do najbližeg neprijatelja koju inicijaliziramo pomoću Math.Infinity koja označava beskonačnu daljinu. Kreiramo objekt koji će nam predstavljati trenutno najbližeg neprijatelja „nearestEnemy“ te ga inicijaliziramo na null vrijednost.

Sada iteriramo foreach petljom kroz svaki objekt neprijatelja u polju „enemies“. Za svaki objekt neprijatelja računamo daljinu između tornja i neprijatelja te ako je daljina između tornja i neprijatelja manja od trenutne najmanje daljine između tornja i neprijatelja postavljamo da je najmanja daljina jednaka toj novoj daljini te postavljamo objekt najbližeg neprijatelja na trenutnu iteraciju objekta neprijatelja.

Nakon što smo prošli kroz polje svih neprijatelja te pronašli najbližeg neprijatelja, provjeravamo da li je uopće pronađen najbliži neprijatelj te da li je unutar opsega tornja. Ako

najbliži neprijatelj postoji i unutar je opsega tornja tada postavljamo našu varijablu mete tornja „target“ na poziciju tog najbližeg neprijatelja te varijablu „targetEnemy“ na komponentu tipa Enemy tog neprijatelja. Ako nije pronađen najbliži neprijatelj ili je izvan opsega postavljamo metu tornja na null vrijednost jer nema neprijatelja unutar njegovog opsega.

```
void Start()
{
    InvokeRepeating("UpdateTarget", 0f, 0.5f);
}
```

U metodi Start() koja se pokreće prilikom pokretanja skripte pozivamo metodu InvokeRepeating koja proslijeđenu metodu poziva na određeni vremenski trenutak te zatim ponovno svaki određeni broj vremena. U našem slučaju pozivamo metodu UpdateTarget, prvi poziv je odmah na početku te ponovno pozivamo UpdateTarget svakih pola sekunde. UpdateTarget metodu pozivamo na ovaj način, a ne u Update metodi jer ne želimo da naši tornjevi svaki okvir traže novu metu što bi usporavalo performans igre.

```
void LockOnTarget()
{
    Vector3 direction = target.position - this.transform.position;
    Quaternion lookRotation = Quaternion.LookRotation(direction);
    Vector3 rotation = Quaternion.Lerp(partToRotate.rotation, lookRotation,
                       Time.deltaTime * turnSpeed).eulerAngles;
    partToRotate.rotation = Quaternion.Euler(0f, rotation.y, 0f);
}
```

Funkcija LockOnTarget služi nam za rotiranje tijela tornja prema meti. Prvo dohvaćamo vektor smjera oduzimanjem pozicije mete sa trenutnom pozicijom tornja. Dohvaćamo kvaternion koji nam označava smjer rotacije u varijablu „lookRotation“ pozivanjem funkcije Quaternion.LookRotation koja vraća pogled rotacije u smjeru vektora smjera.

Zatim dohvaćamo vektor rotacije pozivanjem Quaternion.Lerp koja nam služi za ugodno rotiranje objekta, a kao argumente prima trenutnu rotaciju objekta koji želimo rotirati (u našem slučaju trenutnu rotaciju tijela tornja), kvaternion rotacije kamo želimo zarotirati naš objekt (u našem slučaju izračunati kvaternion „lookRotation“) te brzinu kojom želimo izvršiti tu rotaciju (kod nas to je Time.deltaTime pomnožen sa definiranom float varijablom za brzinu rotacije „turnSpeed“). To sve pretvaramo u eulerove kutove i spremamo u vektor rotacije jer nam je intuitivnije raditi sa eulerovim kutovima nego sa kvaternionima.

Na kraju nam preostaje postaviti rotaciju tijela našeg tornja na izračunati eulerov kut.

```
void Shoot()
{
    GameObject bulletGO = (GameObject)Instantiate(bulletPrefab,
                          firePoint.position,firePoint.rotation);
    Bullet bullet = bulletGO.GetComponent<Bullet>();

    if (bullet != null)
    {
```

```
            bullet.Seek(target);
        }
    }
```

Slijedeća funkcija u skripti Turret.cs je Shoot() koja nam služi za instanciranje projektila. Na početku instanciramo objekt projektila na poziciji i rotaciji postavljene pozicije za ispaljivanje projektila. Dohvaćamo komponentu tipa Bullet tog projektila. Ako postoji komponenta tipa bullet tada pozivamo njezinu metodu Seek koja služi za translatiranje projektila prema meti i prosljeđujemo joj našu metu. Komponenta tipa Bullet.cs i njezina metoda Seek biti će objašnjene u slijedećem poglavlju (3.6.2.).

```
void Update()
{
    if (target == null)
    {
        return;
    }

    LockOnTarget();

    if (shootRecharge <= 0f)
    {
        Shoot();

        if (slow)
        {
            targetEnemy.SlowDownEnemy(slowPercentage);
        }

        shootRecharge = 1f / fireRate;
    }

    shootRecharge -= Time.deltaTime;
}
```

U Update() metodi, svaki okvir provjeravamo da li postoji meta. Ako meta ne postoji tada se vraćamo iz funkcije, a ako meta postoji pozivamo metodu LockOnTarget() koja rotira tijelo tornja prema meti.

Nadalje, provjeravamo da li je toranj spreman zapucati te pozivamo metodu Shoot() koja instancira metak ako je spreman. Ako je toranj zapucao i toranj ima sposobnost da usporava neprijatelje tada pozivamo metodu neprijatelja iz komponente Enemy.cs SlowDownEnemy koja poziva korutinu koja usporava protivnika na pola sekunde (objašnjeno u poglavlju 3.5.1.). Nakon što je toranj zapucao postavljamo vrijeme punjenja tornja na jednu sekundu podijeljeno sa brzinom paljbe tornja.

Svaki okvir smanjujemo vrijeme punjenja tornja za Time.deltaTime i kada dostigne nulu tada toranj ponovno može zapucati.

3.6.2. Upravljanje projektilima

Nakon što je naš toranj bio spreman na pucanje i pozvao je metodu Shoot() instancirali smo objekt projektila i dohvatili njegovu komponentu Bullet.cs:

```
public int damage;
public float speed;
public float explosionRadius = 0f;
private Transform target;
```

Na početku skripte deklariramo javna svojstva projektila koja inicijaliziramo u inspektoru, a to su njegova šteta, tj. koliko životnih bodova oduzima neprijatelju kada stigne do njega, brzina kojom projektil putuje te radijus eksplozije čija je općenita vrijednost nula. Ako želimo da projektil radi štetu više protivnika odjednom tada trebamo povećati radijus eksplozije.

Nadalje, deklariramo jednu privatnu Transform varijablu „target“ koja će nam predstavljati poziciju našeg neprijatelja.

```
public void Seek (Transform target)
{
    this.target = target;
}
```

Sada možemo vidjeti implementaciju metode Seek koju smo pozvali u skripti iz prethodnog poglavlja Turret.cs. Ona, kada je toranj spreman zapucati i instancirao se projektil, postavlja metu koju će projektil pratiti.

```
void Damage(Transform enemy)
{
    Enemy e = enemy.GetComponent<Enemy>();

    if (e != null)
    {
        e.TakeDamage(damage);
    }
}
```

Metoda Damage služi nam za smanjivanje životnih bodova neprijatelja, a kao argument poprima Transform neprijatelja. Dohvaćamo komponentu tipa Enemy tog neprijatelja te, ako ona nije null vrijednost, pozivamo metodu TakeDamage (objašnjena u poglavlju 3.5.1.) i prosljeđujemo joj štetu našeg projektila. TakeDamage metoda smanjuje životne bodove neprijatelja te ga uništava ako su oni manji od nula.

```
void Explode()
{
    Collider[] colliders = Physics.OverlapSphere(this.transform.position,
                           explosionRadius);
    foreach (Collider collider in colliders)
    {
        if (collider.tag == "Enemy")
        {
            Damage(collider.transform);
```

```
        }
    }
}
```

Sljedeća metoda u skripti Bullet.cs je Explode() i ona nam služi za oštećivanje više neprijatelja odjednom. Kada se metoda pozove, dohvaćamo u polje „colliders“ sve fizičke objekte koji se nalaze u radijusu sfere sa ishodištem u trenutnoj poziciji projektila. To nam omogućuje funkcija Physics.OverlapSphere koja dohvaća sve fizičke objekte u nekom radijusu sfere. Kao prvi argument poprima ishodište sfere (u našem slučaju trenutna pozicija projektila), a kao drugi argument prima radijus sfere (u našem slučaju naš radijus eksplozije).

Sada iteriramo sa foreach petljom kroz sve objekte koje smo pohranili u naše polje „colliders“. Za svaki objekt iteracije koji ima oznaku „Enemy“, tj. ako je objekt naš neprijatelj, tada pozivamo metodu Damage kojoj kao argument šaljemo Transform tog objekta neprijatelja i smanjujemo mu životne bodove. Na taj način ostvarili smo funkcionalnost oštećivanja više protivnika odjednom.

```
void HitTarget()
{
    if (explosionRadius > 0f)
    {
        Explode();
    }
    else
    {
        Damage(this.target);
    }

    Destroy(gameObject);
}
```

Sljedeća metoda u ovoj skripti je HitTarget koju želimo pozivati kada se projektil sasvim približi meti. U metodi provjeravamo da li je radijus eksplozije veći od nule. Ako je radijus eksplozije veći od nule pozivamo metodu Explode jer želimo oštetiti sve neprijatelje u radijusu eksplozije, a inače pozivamo samo Damage gdje je argument naša meta i oštećujemo samo tog jednog neprijatelja.

```
void Update()
{
    if (target == null)
    {
        Destroy(gameObject);
        return;
    }

    Vector3 direction = target.position - this.transform.position;
    float distanceThisFrame = speed * Time.deltaTime;

    if (direction.magnitude <= distanceThisFrame)
    {
        HitTarget();
        return;
```

```
    }

    this.transform.Translate(direction.normalized * distanceThisFrame,
                             Space.World);
    this.transform.LookAt(target);
}
```

Sada, kada smo objasnili sve funkcije koje ćemo koristiti u ovo skripti prelazimo na Update() metodu koja se poziva svakog okvira.

Prvo provjeravamo da li je meta null vrijednost. Ako je meta null vrijednost tada uništavamo projektil i vraćamo se iz funkcije naredbom return.

Ako imamo metu, tada trebamo odrediti vektor smjera prema toj meti, a on je jednak razlici vektora mete i vektora gdje se projektil trenutno nalazi. Zatim određujemo daljinu putanje projektila za svaki okvir i ona iznosi Time.deltaTime pomnoženo sa našom varijablom brzine projektila „speed“ jer želimo konstantno micanje kroz svaki okvir.

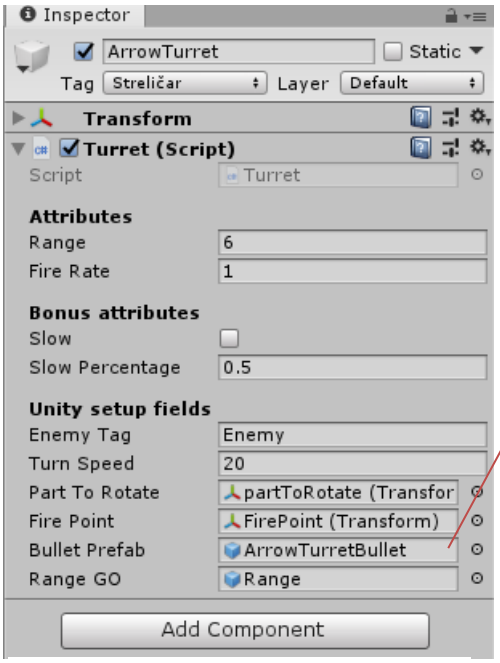
Nadalje, provjeravamo da li je magnituda vektora smjera prema meti manja od daljine koju projektil prolazi svaki okvir. Ako je manja tada se projektil sasvim približio neprijatelju i pozivamo funkciju HitTarget() koja će smanjiti životne bodove neprijatelja i naredbom return se vraćamo iz Update metode.

Ako je magnituda vektora smjera prema meti veća od daljine koju projektil prođe u jednom okviru onda trebamo približiti projektil meti pa ga translatiramo prema meti. Projektil translatiramo u normaliziranom smjeru prema meti pomnoženo sa daljinom koju projektil treba proći svaki okvir, kao drugi argument translatacije prosljeđujemo Space.World jer želimo da se objekt transformira koristeći koordinate cijele scene.

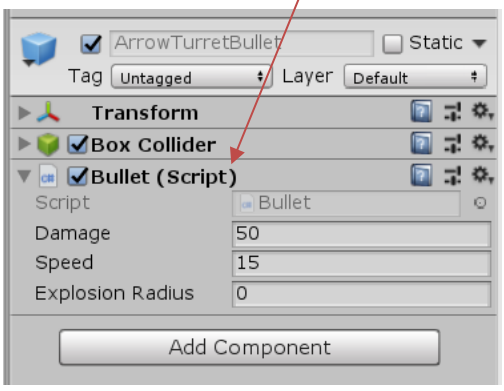
Na kraju pozivamo LookAt funkciju i prosljeđujemo joj našu metu. LookAt funkcija rotira naš projektil da gleda prema meti.

3.6.3. Vrste tornjeva

Kreirali smo skriptu Turret.cs koja upravlja traženjem mete, ažuriranjem mete i rotacijom tornja te skriptu Bullet.cs koja upravlja projektilom tornja te radi štetu neprijateljima. Sada našim preuzetim modelima sa Unity Asset trgovine trebamo na pravi način postaviti te skripte. Na svaki model projektila dodat ćemo skriptu Bullet.cs i pretvoriti u Prefab. Na svaki model tornja dodat ćemo skriptu Turret.cs i pretvoriti u Prefab.



Slika 6. Prefab tornja



Slika 7. Prefab projektila

Svakom Prefabu tornja dodajemo skriptu Turret.cs i u inspektoru u polje Bullet Prefab dodajemo Prefab našeg projektila (crvena strelica na slici) na kojem se nalazi skripta Bullet.cs. Na ovaj način omogućavamo jednostavnu manipulaciju i kreiranje novih tornjeva i projektila te mijenjanje snage, opsega, brzine paljbe i ostalih atributa. Sve što je potrebno za kreiranje nove vrste tornja je modelu tornja i projektila dodati naše kreirane skripte te priključiti projektil na skriptu tornja. Ako želimo da naš toranj ima sposobnost usporavanja neprijatelja samo trebamo označiti u inspektoru atribut Slow. Ako želimo da toranj radi štetu više neprijatelja odjednom dovoljno je da mu povećamo atribut Explosion Radius na željeni radijus štete.

Različite vrste tornjeva u našoj igri:

- Streličar – jeftin, prosječan u svim segmentima, nema sposobnosti
- Streličar X – nadogradnja streličara
- Bacač raketa – dugo vrijeme punjenja, sposobnost oštećivanja više neprijatelja odjednom
- Bacač raketa X – nadogradnja bacača raketa
- Zamrzivač – mala snaga, brza paljba, sposobnost usporavanja neprijatelja
- Zamrzivač X – nadogradnja zamrzivača
- Snajper – velika snaga, veliki opseg, dugo vrijeme punjenja, skup, nema sposobnosti
- Snajper X – nadogradnja snajpera

3.6.4. Upravljanje tornjevima

Sada, kada naši tornjevi znaju ažurirati mete i ispaljivati projektila prema njoj te projektili znaju pratiti metu i oštetiti metu, trebamo implementirati način na koji ćemo kupovati, postavljati, nadograđivati i prodavati naše tornjeve. Za to će nam biti potrebno više skripata koje ćemo opisati u ovom poglavlju.

3.6.4.1. Shematski plan (*blueprint*) tornja

Prvo što nam je potrebno implementirat ćemo u skripti TurretBlueprint.cs. To će nam biti pomoćna klasa koja će sadržavati sve potrebne informacije o pojedinom tornju i njegovoj nadogradnji.

```
[System.Serializable]
public class TurretBlueprint
{
    public GameObject prefab;
    public int cost;
    public GameObject upgradedPrefab;
    public int upgradeCost;

    public int GetSellAmount()
    {
        return cost / 2;
    }

    public int GetUpgradedSellAmount()
    {
        return (cost + upgradeCost) / 2;
    }
}
```

Našoj pomoćnoj klasi dodat ćemo svojstvo System.Serializable kako bi ju mogli inicijalizirati u inspektoru Unity alata. Na početku skripte deklarirat ćemo dva javna objekta „prefab“ i „upgradedPrefab“ koji će sadržavati Prefab tornja i Prefab unaprijeđene verzije tog tornja. Npr., u našim vrstama tornjeva definirali smo streličara i streličara X. Prefab streličara postavit ćemo u varijablu „prefab“, a streličara X postavit ćemo u varijablu „upgradedPrefab“. Isto tako, i za toranj i njegovu unaprijeđenu verziju definiramo koliko će koštati za izgradnju.

Nadalje, imamo još dvije funkcije GetSellAmount i GetUpgradedSellAmount koje nam vraćaju iznos koliko igrač dobiva kada proda toranj. Zbrajamo ukupnu cijenu tornjeva te je dijelimo sa dva. Igrač će uvijek prilikom prodaje dobiti natrag pola cijene koliko je platio toranj.

3.6.4.2. Upravljanje izgradnjom tornjeva

Skripta BuildManager.cs služit će nam za upravljanje izgradnjom tornjeva i biti će realizirana po singleton uzorku. Singleton uzorak je način implementacije klase na koji osiguravamo da klasa ima samo jednu globalno dostupnu instancu kojoj ostale klase mogu pristupiti u bilo kojem vremenu (Unity Manual, 2020). Našu klasu BuildManager.cs

implementirat ćemo kao singleton jer ne želimo da u jednom vremenu postoji više instanca klase BuildManager koje će upravljati izgradnjom naših tornjeva.

```
public static BuildManager instance;

void Awake()
 {
    if (instance != null)
    {
        return;
    }
    instance = this;
}
```

Singleton uzorak imlementiramo tako da na početku deklariramo javnu statičku varijablu tipa iste klase koju implementiramo (u našem slučaju BuildManager). Zatim, u funkciji Awake, koja se pokreće čim se skripta pokreće, provjeravamo da li je već instancirana statička varijabla te, ako je već instancirana, vraćamo se iz funkcije. Ako varijabla još nije instancirana tada ju postavljamo na trenutnu instancu klase. Ovime osiguravamo da se klasa instancira samo kada joj neka klasa prvi puta pristupa te nikad više.

```
private TurretBlueprint turretToBuild;
private TurretNode selectedTurretNode;
```

Nadalje, u skripti BuildManager.cs, deklariramo dvije privatne varijable koje će nam biti potrebne za upravljanje izgradnjom tornjeva. Prva je tipa TurretBlueprint (objašnjeno u poglavlju 3.6.4.1.), a druga je tipa TurretNode (objašnjeno u poglavlju 3.6.4.3.) i označavati će čvor na terenu na kojem želimo upravljati tornjom.

```
public TurretBlueprint GetTurretToBuild ()
{
     return turretToBuild;
}
```

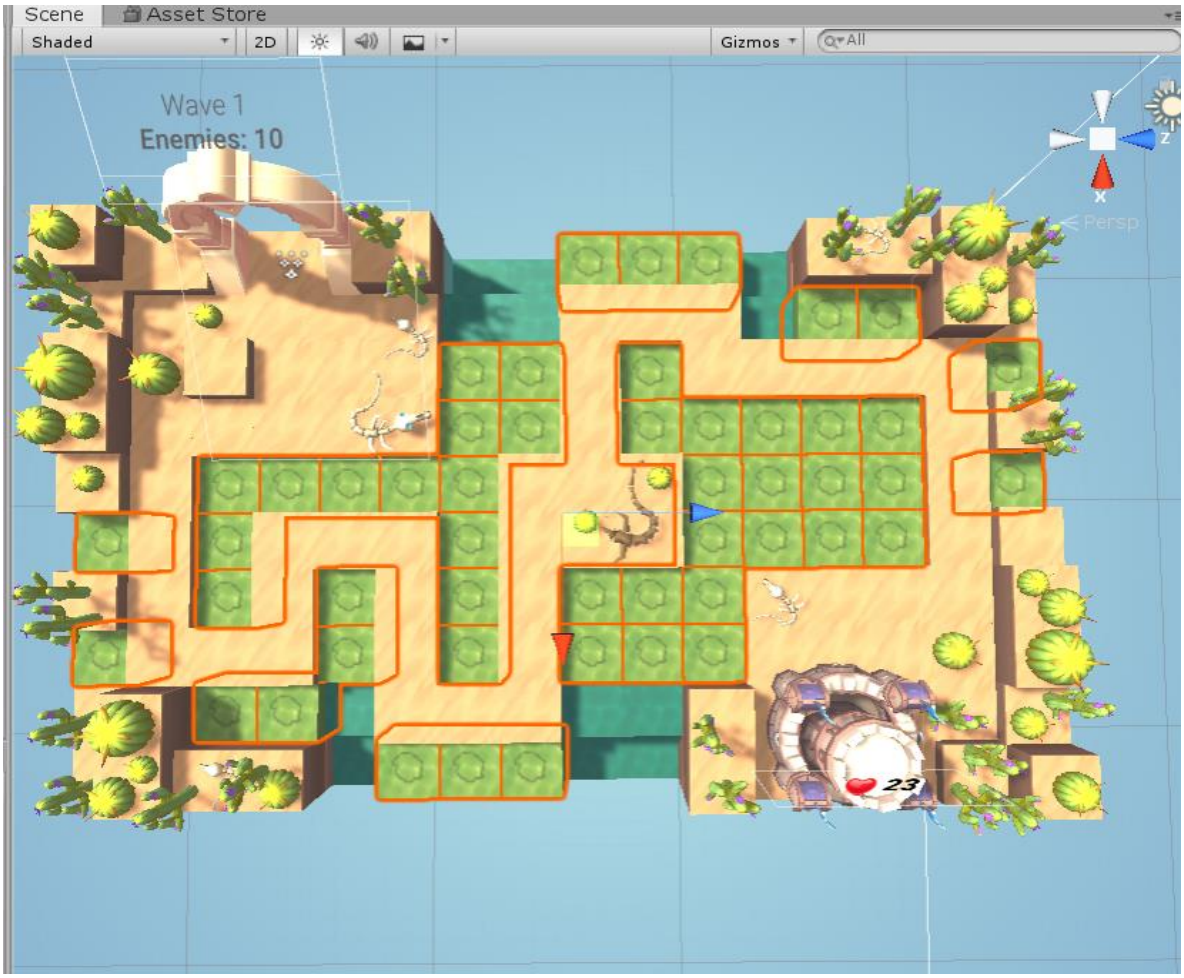
Funkcija GetTurretToBuild nam vraća varijablu turretToBuild koja će biti null ako nije postavljena ili tipa TurretBlueprint ako je postavljena. Služit će nam u drugim funkcija kao provjera mogućnosti građenja te dohvat pravilnog tornja za građenje.

```
public void SelectTurretToBuild (TurretBlueprint turret)
{
    turretToBuild = turret;
}
```

Funkcija SelectTurretToBuild postavlja proslijeđeni TurretBlueprint u našu varijablu tipa TurretBlueprint „turretToBuild“. Pozivat ćemo kada igrač odabere toranj iz trgovine (objašnjeno u poglavlju 3.6.4.4.)

3.6.4.3. Čvorovi za izgradnju tornjeva

Kako bismo mogli postaviti tornjeve na naš teren, trebamo u našoj sceni označiti čvorove koji će pružati mogućnost da na njih postavimo toranj.



Slika 8. Čvorovi

Označili smo čvorove (crveno označeni objekti na slici) za koje želimo da imaju mogućnost zadržavanja tornjeva. Sada ćemo na njih dodati skriptu TurretNode.cs u kojoj ćemo implementirati tu mogućnost.

```
[HideInInspector]
public GameObject turret;
[HideInInspector]
public TurretBlueprint turretBlueprint;
[HideInInspector]
public bool isUpgraded = false;
```

Svaki čvor sadržavati će informacije o trenutno sagrađenom tornju (varijabla „turret“), o trenutnom shematskom planu tornja (varijabla „turretBlueprint“) te bool varijablu u kojoj ćemo označavati da li je toranj unaprijeđen ili ne (varijabla „isUpgraded“) koju na početku inicijaliziramo na false jer na početku toranj nikad nije unaprijeđen. Svakoj varijabli dodali smo

svojstvo HideInInspector jer ne želimo dopustiti njezinu inicijalizaciju u inspektoru, ali želimo omogućiti drugim skriptama pristup do nje.

```
public Vector3 positionOffset;
public Vector3 rotationOffset;
```

Dalje deklariramo dva vektora pomoću kojih ćemo toranj smjestiti na pravilnu poziciju i rotaciju na čvoru.

```
BuildManager buildManager;

void Start()
{
    buildManager = BuildManager.instance;
}
```

Zatim kreiramo varijablu tipa BuildManager i u Start metodi dohvaćamo jedinu instancu te klase pošto je realizirana kao singleton (3.6.4.2.).

```
void BuildTurret (TurretBlueprint blueprint)
{
    if (PlayerStats.money < blueprint.cost)
    {
        return;
    }

    PlayerStats.money -= blueprint.cost;

    GameObject turret = (GameObject)Instantiate(blueprint.prefab,
                        transform.position + positionOffset,
                        transform.rotation *
                        Quaternion.Euler(rotationOffset));
    this.turret = turret;

    turretBlueprint = blueprint;
}
```

Funkcija BuildTurret instancira objekt našeg tornja, a kao argument prima shematski plan tornja koji sadržava informacije o tornju kojeg želimo sagraditi te njegovu cijenu.

Na početku funkcije provjeravamo da li igrač ima dovoljno novaca za izgradnju tornja, ako nema vraćamo se iz funkcije.

Ako igrač ima dovoljno novaca tada mu smanjujemo novac za cijenu tornja kojeg želi sagraditi.

Zatim instanciramo objekt proslijeđenog tornja sa funkcijom Instantiate koja prima Prefab tornja koji želimo instancirati, poziciju na kojoj želimo instancirati toranj (u našem slučaju to je trenutna pozicija čvora na koju nadodajemo naš izmak pozicije) te rotaciju na kojoj želimo instancirati toranj (trenutna rotacija čvora kojoj nadodajemo naš izmak rotacije). U nastavku, postavljamo privatnu varijablu „turret“, deklariranu na početku skripte na instancirani toranj kako bi čvor pratio koji toranj je trenutno izgrađen na njemu.

Na kraju funkcije, postavljamo varijablu „turretBlueprint“, deklariranu na početku skripte, na proslijeđenu varijablu funkciji tipa TurretBlueprint kako bi čvor znao pratiti shematski plan tornja prilikom unaprjeđenja ili prodaje tornja.

```
public void UpgradeTurret()
{
    if (PlayerStats.money < turretBlueprint.upgradeCost)
    {
        return;
    }

    PlayerStats.money -= turretBlueprint.upgradeCost;

    Destroy(this.turret);

    GameObject turret =
   (GameObject)Instantiate(turretBlueprint.upgradedPrefab,
                    transform.position + positionOffset,
                    transform.rotation *
                    Quaternion.Euler(rotationOffset));
    this.turret = turret;

    isUpgraded = true;
}
```

Funkcija UpgradeTurret služi za unaprjeđivanje trenutno postavljenog tornja na čvoru. Na početku provjeravamo da li igrač ima dovoljno novaca za napredak tornja te ako nema izlazimo iz funkcije. Ako ima dovoljno novaca tada mu oduzimamo novac.

Kako bismo stvorili unaprijeđeni toranj na istoj poziciji trebamo prvo uništiti trenutno postavljeni toranj na čvoru. Nakon što smo ga uništili, instanciramo unaprjeđenu verziju tornja kao i u funkciji BuildTurret samo što sada funkciji Instantiate prosljeđujemo unaprjeđeni Prefab tornja do kojeg dolazimo preko shematskog plana „turretBlueprint“.

Postavljamo da je trenutno sagrađeni toranj čvora jednak novoinstanciranom tornju te postavljamo varijablu „isUpgraded“ na true što nam označava da je toranj unaprijeđen.

```
public void SellTurret()
{
    if (!isUpgraded)
    {
        PlayerStats.money += turretBlueprint.GetSellAmount();
    }
    else
    {
        PlayerStats.money += turretBlueprint.GetUpgradedSellAmount();
        isUpgraded = false;
    }

    Destroy(turret);
    turretBlueprint = null;
}
```

Kako imamo funkcije za izgradnju i unaprjeđenje tornjeva potrebna nam je i funkcija za prodaju tornjeva. Funkcija SellTurret nam omogućuje tu prodaju.

Na početku funkcije provjeravamo da li je toranj unaprjeđen ili ne. Ako nije unaprijeđen tada povećavamo novac igrača samo za cijenu prodaje obične verzije tornja, a ako je unaprijeđen povećavamo mu novac za prodajnu cijenu unaprijeđene verzije tornja te postavljamo varijablu „isUpgraded“ na false kako bi mogli novi običan toranj sagraditi na istom čvoru.

Nakon toga uništavamo objekt tornja te postavljamo shematski plan čvora na null jer nam više nije potreban pošto smo uništili sam toranj.

```
void OnMouseDown()
{
    if (EventSystem.current.IsPointerOverGameObject())
    {
        return;
    }

    if (buildManager.GetTurretToBuild() == null)
    {
        return;
    }

    BuildTurret(buildManager.GetTurretToBuild());
    buildManager.SelectTurretToBuild(null);
}
```

OnMouseDown funkcija poziva se svaki put kada igrač pritisne lijevi klik miša. Kada klikne provjeravamo da li je igrač kliknuo na čvor koji već sadržava neki objekt. Ako sadržava, to znači da je na čvoru već sagrađen toranj pa se vraćamo iz funkcije.

Zatim provjeravamo da li je u buildManageru selektiran toranj koji želimo izgraditi te se vraćamo ako je on null.

Ako selektiran toranj postoji, tada pozivamo BuildTurret metodu kojoj prosljeđujemo taj selektirani toranj. Nakon što smo sagradili toranj postavljamo selektirani toranj u „buildManageru“ natrag na null vrijednost.

3.6.4.4. Trgovina tornjeva

Skripta Shop.cs omogućit će igraču da vidi dostupne tornjeve koje može kupiti te će omogućiti selektiranje željenog tornja. Sučelje trgovine prikazuje slike tornjeva te njihovu cijenu.



Slika 9. Sučelje trgovine

```
public TurretBlueprint arrowTurret;
public TurretBlueprint missileLauncher;
public TurretBlueprint slowingTurret;
public TurretBlueprint ballistaTurret;
```

Na početku skripte Shop.cs deklrariramo shematske planove TurretBlueprint za svaki toranj koji će igrač moći kupiti iz trgovine te ga inicijaliziramo u inspektoru.

```
BuildManager buildManager;

void Start()
{
    buildManager = BuildManager.instance;
}
```

Zatim deklariramo varijablu tipa BuildManager te ju dohvaćamo u Start metodi kao i u prethodnom poglavlju.

```
public void SelectArrowTurret()
{
    buildManager.SelectTurretToBuild(arrowTurret);
}

public void SelectMissileLauncher()
{
    buildManager.SelectTurretToBuild(missileLauncher);
}

public void SelectSlowingTurret()
{
    buildManager.SelectTurretToBuild(slowingTurret);
}

public void SelectBallistaTurret()
{
    buildManager.SelectTurretToBuild(ballistaTurret);
}
```

Deklrariramo funkcije za svaku vrstu tornja koja, kada igrač klikne na toranj u trgovini, poziva metodu BuildManagera buildManager.SelectTurretToBuild i prosljeđuje mu selektirani

TurretBlueprint sa trgovine. BuildManager tada sadržava selektirani toranj te ga TurretNode.cs skripta može izgraditi ukoliko igrač ima dovoljno novaca.

3.6.4.5. Nadograđivanje i prodavanje tornjeva

Kako bismo mogli jednostavno nadograđivati i prodavati tornjeve kreiramo jednostavno sučelje sa gumbima za nadogradnju i prodaju.



Slika 10. Sučelje nadogradnje i prodaje

Skripta TurretSelectUI.cs omogućit će nam upravljanje našim sučeljem. Naše kreirano sučelje postavit ćemo u scenu na početku igre te ga onemogućiti. Kada igrač pritisne na toranj tada ćemo ga omogućiti i smjestiti iznad odabranog tornja sa pripadajućim cijenama.

```
public GameObject ui;
public Text upgradeCost;
public Text sellCost;
public Button upgradeButton;
private TurretNode target;
```

Na početku skripte deklariramo objekt „ui“ koji nam predstavlja sučelje sa gornje slike. Deklariramo objekte tipa Text „upgradeCost“ i „sellCost“ kojima ćemo ažurirati tekst cijena ovisno o tome koja vrsta tornja je odabrana. Zatim deklariramo varijablu tipa Button koja će nam predstavlja gumb za unaprjeđenje tornja, njega želimo onemogućiti kada je toranj unaprjeđen. Posljednje deklariramo privatnu varijablu „target“ tipa TurretNode koja nam predstavlja čvor na kojem se toranj nalazi i informacije o njemu te iznad kojeg treba prikazati naše sučelje.

```
public void SetTarget( TurretNode target)
{
    this.target = target;
    this.transform.position = target.transform.position +
                              target.positionOffset;

    if (!target.isUpgraded)
    {
        upgradeCost.text = target.turretBlueprint.upgradeCost.ToString();
        upgradeButton.interactable = true;
        sellCost.text = target.turretBlueprint.GetSellAmount().ToString();
    }
    else
```

```
    {
        upgradeCost.text = "MAX";
        upgradeButton.interactable = false;
        sellCost.text =
        target.turretBlueprint.GetUpgradedSellAmount().ToString();
    }
    ui.SetActive(true);
}
```

Glavna funkcija ove skripte je SetTarget koja kao argument prima čvor sa tornjom kojeg je igrač odabrao. Na početku funkcije postavljamo privatnu varijablu „target“ na proslijeđeni čvor te postavljamo poziciju sučelja na poziciju čvora sa postavljenim izmakom čvora. Zatim provjeravamo da li je toranj unaprjeđen. Ako nije unaprjeđen, postavljamo tekst cijene gumba za unaprjeđenje na cijenu tornja koji je na odabranom čvoru, omogućujemo klik na gumb unaprjeđenja te postavljamo tekst gumba za prodaju na cijenu tornja koji se nalazi na čvoru.

Ako je toranj već unaprjeđen, tada postavljamo tekst cijene unaprjeđenja na „MAX“, onemogućujemo klik na gumb unaprjeđenja jer je toranj već unaprjeđen te postavljamo tekst cijene prodaje tornja na prodajnu cijenu odabranog unaprjeđenog tornja.

Nakon što smo postavili tekst cijena i omogućenost gumba tada aktiviramo sučelje sa metodom ui.SetActive(true).

```
public void Hide()
{
    ui.SetActive(false);
}
```

Metodom Hide sakrivamo sučelje za unaprjeđenje / prodaju.

```
public void Upgrade()
{
    target.UpgradeTurret();
}
```

Metoda Upgrade poziva se klikom igrača na gumb „Nadogradi“ te poziva metodu čvora tipa TurretNode.cs koja se naziva UpgradeTurret (objašnjena u poglavlju 3.6.4.3.) koja unaprjeđuje toranj u njegovu poboljšanu verziju ako igrač ima dovoljno novaca.

```
public void Sell()
{
    target.SellTurret();
}
```

Metoda Sell poziva se klikom igrača na gumb “Prodaj” te se poziva metoda trenutnog čvora na kojeg je igrač pritisnuo tipa TurretNode naziva SellTurret (objašnjena u poglavlju 3.6.4.3.) koja prodaje trenutni toranj koji se nalazi na čvoru i povećava novac igrača.

Kako bismo pravilno upravljali sučeljem za nadogradnju / prodaju tornja trebamo dodati još dvije funkcije u skriptu BuildManager.cs

```
public TurretSelectUI turretSelectUI;

public void DeselectTurretNode()
{
    selectedTurretNode = null;
    turretSelectUI.Hide();
}

public void SelectTurretNode(TurretNode turretNode)
{
    if (selectedTurretNode == turretNode)
    {
        DeselectTurretNode();
        return;
    }
    turretSelectUI.SetTarget(turretNode);
}
```

Na početku skripte dodajemo novu varijablu tipa naše novokreirane skripte TurretSelectUI kojom ćemo upravljati sučeljem za nadogradnju / prodaju tornjeva.

Funkcija DeselectTurretNode postavlja varijablu skripte BuildManager.cs „selectedTurretNode“ na null vrijednost te sakriva sučelje za nadogradnju / prodaju tornjeva.

Funkcija SelectTurretNode kao argument funkcije poprima čvor tipa TurretNode. Prvo provjeravamo da li je već označen taj čvor. Ako je već označen tada pozivamo metodu DeselectTurretNode koja ga odznačuje i sakriva sučelje za nadogradnju / prodaju. Time smo postigli da se igraču otvara sučelje kada prvi puta pritisne na čvor sa tornjem, a zatvara kada drugi puta pritisne na čvor.

Nakon toga pozivamo funkciju SetTarget prethodno objašnjenu te joj prosljeđujemo čvor nad kojim želimo postaviti sučelje za nadogradnu / prodaju.

Sada, kako bismo upotpunili funkcionalnost trebamo negdje pozvati tu metodu SelectTurretNode. Dodati ćemo tu funkcionalnost na klik miša u skripti TurretNode.cs (objašnjeno u poglavlju 3.6.4.3).

```
void OnMouseDown()
{
    if (EventSystem.current.IsPointerOverGameObject())
    {
        return;
    }

    if (turret != null)
    {
        buildManager.SelectTurretNode(this);
        return;
    }

    if (!buildManager.CanBuild())
    {
        return;
    }
```

```
    BuildTurret(buildManager.GetTurretToBuild());
    buildManager.SelectTurretToBuild(null);
}
```

Nakon provjere da li već postoji objekt na odabranom čvoru dodali smo provjeru da li na odabranom čvoru već postoji toranj. Ako postoji tada pozivamo prethodno objašnjenu metodu SelectTurretNode i prosljeđujemo mu trenutno selektirani čvor. Na taj način će naše sučelje za nadogradnju / prodaju tornja poprimiti poziciju iznad selektiranog tornja sa pravilnim svojstvima.

4. Čarolija

Uz izbor između različitih vrsta tornjeva, igrač ima i na raspolaganju jednu čaroliju koja radi štetu svim živim neprijateljima u sceni. Implementirana je u skripti Spell.cs.

```
public int spellCost;
public int spellDamage;
```

Na početku skripte deklariramo varijable koje predstavljaju cijenu i snagu čarolije.

```
private void CastSpell()
{
    PlayerStats.money -= spellCost;

    GameObject[] enemies = GameObject.FindGameObjectsWithTag("Enemy");

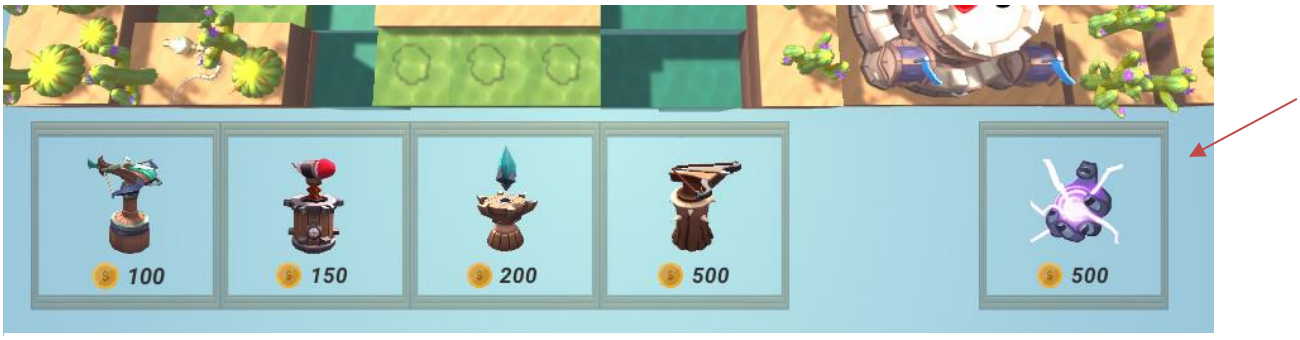
    foreach (GameObject enemy in enemies)
    {
        enemy.GetComponent<Enemy>().TakeDamage(spellDamage);
    }
}
```

Funkcija CastSpell na početku oduzima igraču novac u iznosu koliko košta čarolija. Zatim dohvaćamo u polje sve objekte na sceni sa oznakom „Enemy“, dohvaćamo sve žive neprijatelje. Zatim prolazimo kroz polje neprijatelja te za svakog neprijatelja pozivamo metodu njegove komponente tipa Enemy naziva TakeDamage (objašnjeno u poglavlju 3.5.1.) i prosljeđujemo joj definiranu snagu naše čarolije. Ovime smo ostvarili oštećivanje svakog neprijatelja na sceni.

```
public void ActivateSpell()
{
    if(PlayerStats.money >= spellCost)
    {
        CastSpell();
    }
}
```

Funkcija ActivateSpell služi nam za aktivaciju čarolije. Kada se pozove prvo provjerava da li igrač ima dovoljno novaca za aktiviranje čarolije. Ako igrač ima dovoljno novaca, tada se poziva prethodno definirana metoda CastSpell.

Kao i tornjeve, čaroliju dodajemo na naše sučelje trgovine i aktiviramo klikom miša na čaroliju.



Slika 11. Sučelje trgovine sa čarolijom

5. Upravljanje završetkom igre

Kreirat ćemo skriptu koja će upravljati stanjem igre tj., ako je igrač izgubio sve životne bodove želimo da se igra zaustavi i želimo pokrenuti sučelje koje sugerira igraču da je izgubio. U slučaju da je igrač uništio sve neprijatelje kroz sve valove, tada zaustavljamo igru te prikazujemo pobjedničko sučelje.

```
public GameObject gameOverUI;
public GameObject levelCompleteUI;
public static bool gameOver;

private void Start()
{
    gameOver = false;
}
```

Na početku deklariramo objekte sučelja za kraj igre te za pobjedu igre. Deklariramo još jednu statičku varijablu tipa bool koja će nam označavati da li je igra završila ili ne te je u Start metodi inicijaliziramo na false jer igra još nije završila.

```
void EndGame()
{
    gameOver = true;
    gameOverUI.SetActive(true);
}
```

Funkcija EndGame postavlja varijablu „gameOver“ na true te pokreće sučelje za završetak igre kada je igrač izgubio.

```
void LevelComplete()
{
    gameOver = true;
    levelCompleteUI.SetActive(true);
}
```

Funkcija LevelComplete postavlja varijablu „gameOver“ na true te pokreće pobjedničko sučelje kada igrač uništi sve neprijatelje.

```
void Update()
{
    if (gameOver)
    {
        return;
    }

    if (PlayerStats.lives <= 0)
    {
        EndGame();
    }

    if (WaveSpawner.levelComplete)
    {
        LevelComplete();
```

```
        }
}
```

U Update metodi prvo provjeravamo da li je igra završena, ako je igra završena tada ne želimo više ulaziti u ovu funkciju te se vraćamo iz nje.

Zatim provjeravamo da li je igraču ponestalo životnih bodova, ako mu je ponestalo životnih bodova pozivamo metodu EndGame koja pokreće gubitničko sučelje.



Slika 12. Gubitničko sučelje

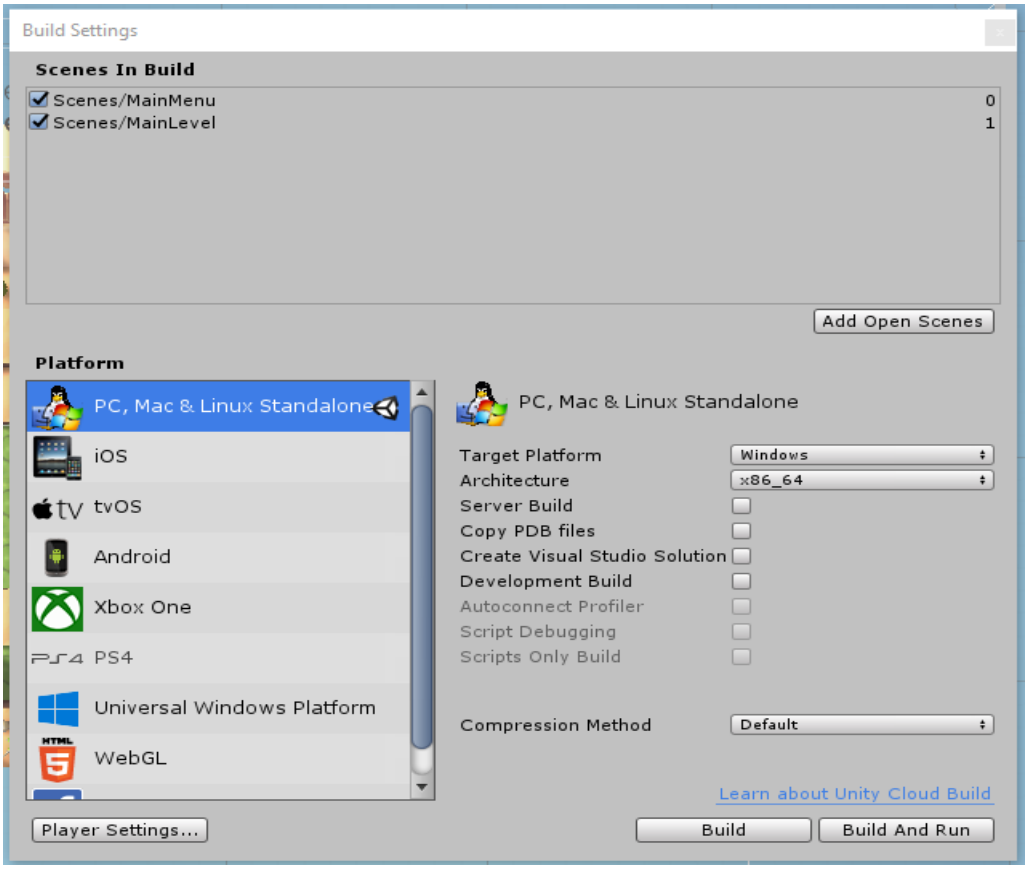
Ako igrač prođe sve i valove i istinita je statična varijabla klase WaveSpawner koja označava kraj razine(objašnjeno u poglavlju 3.5.4.), tada se pokreče pobjedničko sučelje.



Slika 13. Pobjedničko sučelje

6. Isporuka igre

Kada smo završili sa izgradnjom i implementacijom naše igre Unity alat nam nudi mogućnost isporuke igre. U Unity alatu kliknemo na File>BuildSettings i otvara nam se slijedeći prozor:



Slika 14. Isporuka igre

Klikom na Player Settings možemo mijenjati ime, logo, ikonu i slične postavke naše igre. Klikom na gumb Build izgrađujemo igru koju mogu pokrenuti svi korisnici neovisno o tome da li imaju instaliran Unity alat ili ne.

7. Zaključak

Izrada računalne igre nije nimalo lagan posao, ali je vrlo zanimljiv i koristan projekt ako se uloži dovoljno vremena u njegovu izradu. Osim u izradu igre, potrebno je uložiti i vremena u proučavanje Unity alata. Svako iskustvo u objektno orijentiranom programiranju uvelike olakšava implementaciju samih algoritama potrebnih u igri.

Jednostavnost i popularnost Unity alata za izradu igara uvelike je mi je olakšao posao kod izrade igre. Pošto sam upoznat sa principima objektno orijentiranog programiranja sve što mi je trebalo je dobro se upoznati sa alatima potrebnim za izradu igre obrane tornjevima, a ogromna popularnost Unity alata rezultirala je stotinama poučnih videa o njegovom radu koji su mi pomogli da se brzo i efikasno prilagodim njegovoj okolini.

Možda i najbolji sastav Unity alata je njegova trgovina imovinom Unity Asset Store koja omogućuje svim programerima kao meni, koji nisu upoznati sa izradom modela, grafičkih elemenata i animiranjem, da pronađu željene 2D ili 3D modele, efekte, grafičke elemente i još mnogo toga te ih je jednostavno ažurirati i implementirati u vlastitu igru bez potrebnog znanja o načinu njihove izrade.

Smatram da je žanr obrane tornjevima bio odličan način da se uvedem u svijet izrade računalnih igara jer takva igra nema previše kompleksnih algoritama i metoda koje bi bile izvan dosega početnika. Ovaj projekt donio mi je mnogo korisnog programerskog iskustva koje će mi definitivno pomoći u budućnosti.

Popis literature

[1] Alistar, E., Leeuwen, R. P., i Togelius, J. (2014). *Computational Intelligence and Tower Defence Games*. Preuzeto 23.07.2020. s https://ieeexplore.ieee.org/document/5949738

[2] Asset Store, (2020). *Quick guide to the Unity Asset Store*. Preuzeto 23.07.2020. s https://unity3d.com/quick-guide-to-unity-asset-store

[3] Brackeys, (2017). How to make a Tower Defense Game [Video playlist]. Preuzeto 25.07.2020. s https://www.youtube.com/playlist?list=PLPV2KyIb3jR4u5jX8za5iU1cqnQPmbzG0

[4] FullTiltBoogie, (2018). *Starter Particle Pack*. Preuzeto 23.07.2020 s https://assetstore.unity.com/packages/vfx/particles/starter-particle-pack-83179#releases

[5] GIMP, (2020). About GIMP. Preuzeto 23.07.2020. s https://www.gimp.org/about/

[6] Machina, (2020). *Unity 3D game engine: tvornica igara indie studija*. Preuzeto 23.07.2020. s https://www.machina.academy/machina-blog/unity-tvornica-indie-igara

[7] PI, E. L. (2017). *Level 1 Monster Pack*. Preuzeto 23.07.2020. s https://assetstore.unity.com/packages/3d/characters/creatures/level-1-monster-pack-77703#releases

[8] PressSTART, (2018). *PBR_Orc-Pig*. Preuzeto 23.07.2020 s https://assetstore.unity.com/packages/3d/characters/creatures/pbr-orc-pig-109248#releases

[9] Unity, (2020). *Welcome to Unity*. Preuzeto 22.07.2020. s https://unity.com/our-company

[10] Unity Manual, (2020). *Unity User Manual*. Preuzeto 23.07.2020 s https://docs.unity3d.com/Manual/index.html

[11] VSQUAD, (2019). *MEGA Towers Pack*. Preuzeto 23.07.2020. s https://assetstore.unity.com/packages/3d/environments/fantasy/mega-towers-pack-152854#releases

[12] Wijman, T. (2019). *The Global Games Market Will Generate $152.1 Billioin 2019 as the U.S. Overtakes China as the Biggest Market*. Preuzeto 23.07.2020. s https://newzoo.com/insights/articles/the-global-games-market-will-generate-152-1-billion-in-2019-as-the-u-s-overtakes-china-as-the-biggest-market/

Popis slika.

Slika 1: Glavni izbornik 5
Slika 2: Staza 6
Slika 3: Objekti puta 7
Slika 4: Objekti neprijatelja 9
Slika 5: Vrste neprijatelja 13
Slika 6: Prefab tornja 24
Slika 7: Prefab projektila 24
Slika 8: Čvorovi 27
Slika 9: Sučelje trgovine 31
Slika 10: Sučelje nadogradnje i prodaje 32
Slika 11: Sučelje trgovine sa čarolijom 37
Slika 12: Gubitničko sučelje 39
Slika 13: Pobjedničko sučelje 39
Slika 14: Isporuka igre 40

Prilozi

Igra dostupna na https://github.com/mkeretic1/tower-defense-game