INSTITUT ZA MATEMATIKU I INFORMATIKU
PRIRODNO-MATEMATIČKI FAKULTET
UNIVERZITET U KRAGUJEVCU

SEMINARSKI RAD IZ RAČUNARSKE GRAFIKE

TOWER DEFENSE

Mentor
dr Boban Stojanović

Student
Lazar Avramović, 56/14

Septembar 2018.

Sadržaj

Predgovor

Tower Defense igre su zasnovane na preživljavanju nadolazećih talasa neprijatelja sa određenim periodom. Cilj igre je odbraniti “kapiju” (ne dozvoljavajući neprijateljima da prođu), postavljanjem odbrambenih mehanizama (kulica – tower) na dozvoljene površine.

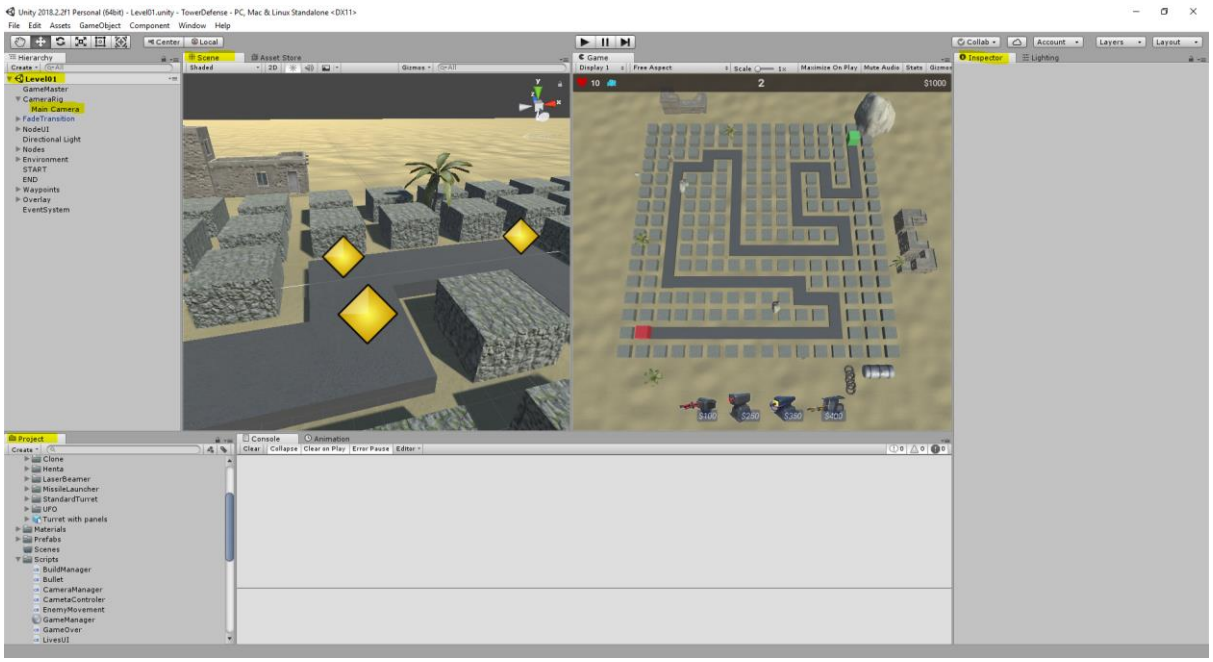
U nastavku ovog dokumenta opisan je postupak razvoja jedne 3D Tower Defense igre za PC (igru je moguće prebaciti i na druge platforme, Android, ali kako je igra prvenstveno namenjena PC računarima, velika je verovatnoća za pad performansi).

Za razvoj je korišćen Unity engine sa C#.

1. Kreiranje okruženja

1.1. Kratko pojašnjenje Unity korisničkog interfejsa

- Scena (Scene) je pozornica, na njoj se nalaze svi objekti i odvija sva “radnja” igre.
- Kamera (Main Camera) predstavlja posmatrača, kroz kameru posmatrač vidi scenu.
- Objektima se manipuliše pomoću grafičkog prikaza scene.
- Svi objekti se sastoje od komponenti i imaju atribute kojima se upravlja preko inspectora.
- Fajlovima projekta se upravlja Project Managerom.



Slika 1: Scena sa objektima i grafički prikaz secene - gore levo, Inspector - desno, Project Manager - dole levo

1.2. Mapa

Mapa se sastoji od četiri vrste objekata: platforme (u nastavku Node), puta (u nastavku Ground), početka (start) i kraja (end).

Svi gore navedeni objekti su kocke (osim puta). Potrebno je kreirati sve navedene objekte i skalirati ih po potrebi. Vrednost Y coordinate vektora skaliranja puta treba da bude manja u odnosu na node (i ostale kocke). Prevlačenjem objekata u neki folder u Project Manager-u kreira se **Prefab** - njegov model (šema). To znači da možemo imati više instanci objekta koji imaju tu šemu. Ovo je korisno kod promene atributa i komponenti objekta. Promena na prefab-u se menifestuje na sve instance objekta na sceni.

Za mapu dimenzija N x M potrebno je kreirati N * M kocki i postaviti ih na scenu u N redova i M kolona. Zatim odabrati put kojim će se neprijatelji kretati i umesto node objekata

postaviti ground objekte (ground objekti ne moraju da imaju svoj prefab, moguće ih je skalirati po potrebi u zavisnosti od oblika puta i ne zahtevaju neku pravilnost). Start i end objekti leže na početku i kraju puta respektivno.

1.3. Materijali

Materijali su sastavni deo svakog objekta, odnosno komponente Mash Renderer. Materijal, kako mu ime kaže, predstavlja ono od čega je taj objekat napravljen. Pomoću materijala manipuliše se izgledom objekta, u nekim slučajevima i ponašanjem, u ovom, samo izgledom.

U folderu projekta Materials napraviti četiri nova materijala (po jedan za svaku od komponenti mape) i postaviti ih u atribut Materials u Mash renderer komponenti šeme (prefab) svakog od objekata. Kako se promene dešavaju na prefab-u, biće vidljive i na svim njegovim instancama na sceni. Na slici 1 može se videti krajnji izgled mape.

1.4. Putokazi (waypoints)

Putokazi imaju ulogu da neprijateljima, koji će se kretati tim putem pokažu gde zapravo treba da idu. Na svakoj promeni pravca puta treba postaviti putokaz, koji je ništa drugo do prazan objekat (Empty Game Object), takođe, treba postaviti i jedan na kraj. Sve postavljene putokaze ubciti pod drugi prazan objekat.

Objektima u toku izvršavanja igre upravlja se skriptama. Skripte se dodaju kao komponente i nisu ništa drugo do C# kodovi.

Za root objekat (putokazi, u kojem su svi kreirani putokazi) kreirati skriptu Waypoints.cs koja ima statičku promenljivu koja predstavlja niz putokaza i popuniti ga kreiranim putokazima.

```
1   public class Waypoints : MonoBehaviour {
2
3       public static Transform[] points;
4
5       private void Awake()
6       {
7           points = new Transform[transform.childCount];
8           for (int i = 0; i < points.Length; i++)
9           {
10              points[i] = transform.GetChild(i);
11          }
12
13      }
14  }
```

2. Neprijatelji i logika neprijatelja

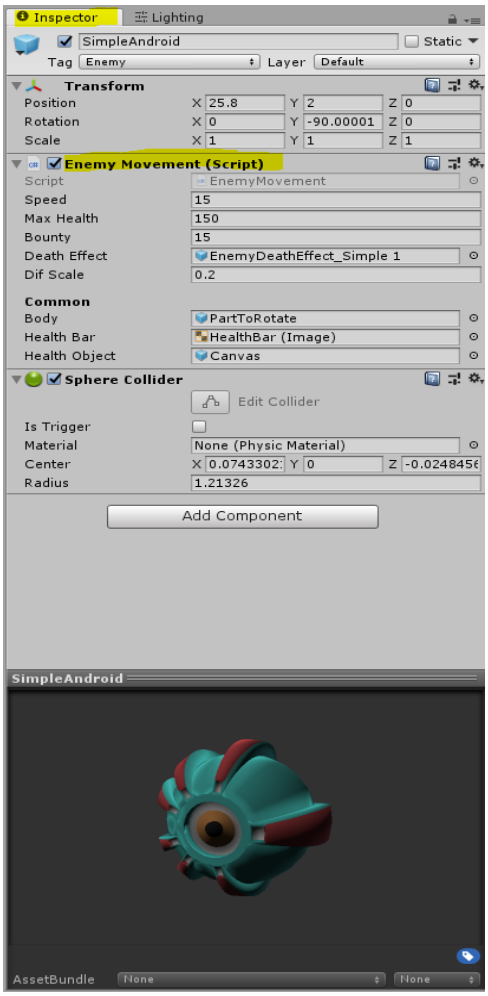
2.1. Import modela i kreiranje neprijatelja

Za početak, naći model po izboru i skinuti ga sa interneta. Naravno, moguće je za neprijatelje koristiti i neki od osnovnih geometrijskih tela, tipa loptu. Import se vrši prevlačenjem .blend ili .fbx fajla u Project Manger.

Postaviti objekat na scenu i odmah napraviti prefab. Svi koraci u nastavku se vrše nad tim prefabom.

Svaki od modela neprijatelja treba da ima **tag** (gornji deo inspector-a) enemy, u nastavku će biti objašnjeno i zašto.

Fizički model se sastoji iz nekoliko delova koji određuju i njegovu funkcionalnost. Neprijatelj ima root prazni objekat koji služi kao kontejner. Na root objektu neprijatelja se kao komponente nalaze Sphere Collider (u ovom slučaju, postoje i drugi collider-i) koji objektu daje "postojanost", detaljnije, objekti sa collider komponentama mogu da interaguju (bukvalno, sudaraju) sa drugim objektima, koji takođe imaju collider komponentu. Druga komponenta je skripta koja upravlja neprijateljem Enemy.cs u njoj se nalazi logika neprijatelja. Unutar root komponente smestiti skinuti model i kanvas (UI element) koji u sebi treba da ima panel, a panel treba da ima sliku (image), koji su takođe UI elementi.



Slika 2: Inspector root komponente standardnog neprijatelja

Isti postupak sa materijalima. Kreirati ih i njima popuniti Mash Renderer komponentu modela.

Nakon svake izmene potrebno je sačuvati promene nad prefabom klikom na apply u inspektoru.

Napraviti više različitih vrsti neprijatelja od kojih će biti sačinjeni talasi u kojima će dolaziti. U ovom slučaju razlikujemo tri vrste: standardni, debeli i brzi. Razlike su u njihovim atributima o kojima će biti reči u nastavku (logika neprijatelja).

2.2. Logika neprijatelja

Za logiku neprijatelja zadužena je skripta Enemy.cs. Razlikujemo dve logičke celine. Jednu koja je zadužena za kretanje i drugu koja interaguje sa ostatkom sveta.

Unity MonoBehavior je klasa koju nasleđuju sve kreirane skripte. MonoBehavior podrzumeva neke predefeinisane metode:

- Start() – kod koji se izvršava pri instanciranju objekata
- Update() – kod koji se izvršava svakog frejma

Takođe, sve deklarisane public promenljive su vidljive u inspectoru u Unity okruženju, što dosta olakšava rad sa skriptama.

2.2.1. Kretanje

Od atributa, svaki neprijatelj mora da ima:

```
15      [Header("Enemy")]
16      public float speed = 10f; //pocetna brzina
17      public float maxHealth = 100;   //zivot
18      public int bounty = 15;     //nagrada po ubijanju neprijatelja
19      public GameObject deathEffect;  //efekat smrti neprijatelja
20      public float difScale = 0.2f;   //faktor skaliranja tezine
21
22      [Header("Common")]
23      public GameObject body;    //model tela neprijatelja
24      public Image healthBar;     //healthBar (slika)
25      public GameObject healthObject = null;  //healthBar (Canvas)
26
27      private Transform target;   //putokaz koji prati
28      private int waypointIndex;  //rbr putokaza
```

Za promenljive brzine i života potrebno je imati i promenljive koje imaju trenutnu vrednost brzine i života neprijatelja.

U Start() metodi vrši se postavljanje inicijalnih vrednost, brzine, života. Postavlja se redni broj potokaza na 0 kao i putokaz koji neprijatelj treba da prati.

U Update() metodi treba naći vektor pravca od trenutne pozicije neprijatelja do putokaza i pomeriti neprijatelja ka putokazu. Koliko će biti pomeren to zavisi od brzine kretanja (speed).

```
29  Vector3 dir = target.position - transform.position;
30  transform.Translate(dir.normalized * speed * Time.deltaTime, Space.World);
```

Transliranjem pomeramo objekat u pravcu vektora izabranom brzinom. Brzinu (sve što zavisi od vremena) potrebno je množiti vremenom koje je prošlo između dva frejma. Time dobijamo istu brzinu na različitim mašinama, na nekim sposobnijim koje imaju veliki broj frejmova, a i na onim koje ne mogu da ispune velike zahteve. Zatim, treba okrenuti healthbar ka posmatraču (aktivnoj kameri, o čemu će biti reči u nastavku) tako što ćemo reći da healthObject (kanvas) gleda u trenutnu kativnu kameru.

```
31  healthObject.transform.LookAt(CameraManager.ActiveCamera.transform.position);
```

Kada je udaljenost trenutne pozicija neprijatelja (transform.position, objekta koji ima skriptu na kojoj radimo kao komponentu) i putokaza koji on prati manja od neke granične vrednosti treba promeniti putokaz i okrenuti telo (body) neprijatelja ka sledećem.

```
32  if (Vector3.Distance(transform.position, target.position) <= 0.2f)
33  {
34        getNextWaypoint();
35        body.transform.LookAt(target.position);
36  }
```

Kod promene putokaza iz statičke promenljive klase Waypoints.cs pristupamo svim putokazima, inkrementiramo redni broj i za trenutni putokaz uzimamo sledeći iz niza putokaza. U slučaju da je putokaz poslednji u nizu treba smanjiti broj života igrača, ubiti neprijatelja i umanjiti broj neprijatelja koji su trenutno živi (više u menadžeru talasa).

```
37      private void getNextWaypoint()
38      {
39          if(waypointIndex >= Waypoints.points.Length - 1)
40          {
41              EndPath();
42              return;
43          }
44
45          waypointIndex++;
46          target = Waypoints.points[waypointIndex];
47      }
48
49      private void EndPath()
50      {
51          PlayerStats.lives--;
52          Destroy(gameObject);
53          WaveSpawner.EnemiesAlive--;
54      }
```

2.2.2. Interakcija sa svetom

Neprijatelji dolaze u talasima, kulice pucaju na njih. Potrebno je napraviti logiku kojom će ovi modeli interagovati jedni sa drugima.

Ono što neprijatelj treba da zna, pored kretanja je da primi štetu nanetu od strane kulice i umre. Potrebno je i vizuelno dati do znanja da je neprijatelj uništen. To postižemo efektom koji neprijatelj treba da pokrene pri umiranju.

Prilikom primanja štete treba proveriti da li će neprijatelj umreti, u tom slučaju uništiti neprijatelja u suprotnom umanjiti život i procentualno umanjiti popunjenost healthbar-a (healthbar je popunjen sitim kvadratima, public Image healthBar).

```
55      public void TakeDamage(float dmg)
56      {
57          if (health - dmg <= 0f)
58          {
59              Die();
60              return;
61          }
62
63          health -= dmg;
64
65          healthBar.fillAmount = health / maxHealth;
66      }
```

```
67      private  void Die()
68      {
69          gameObject.SetActive(false);
70          Destroy(gameObject);
71          PlayerStats.money += bounty;
72          GameObject effect = (GameObject)Instantiate(deathEffect,
    transform.position, Quaternion.identity);
73          Destroy(effect, 5f);
74          WaveSpawner.EnemiesAlive--;
75      }
```

Kada neprijatelj umre treba uništiti objekat i uvećati novac koji igrač ima (onoliko koliko ubijeni neprijatelj vredi), instancirati efekat i umanjiti broj trenutno živih neprijatelja.

PlayerStats.cs skripta će biti u GameManager-u (u nastavku) i držaće informacije o igraču (novac, životi, trenutni talas) u statičkim promenljiva kojima se može pristupati iz svih delova programa.

Kako će se težina povećavati tako što će se skalirati maksimalni životni poeni neprijatelja biće reči o inicijalizaciji trenutne vrednosti (health) u metodi Start() u poglavlju o menadžeru talasa neprijatelja (u nastavku Wavespawner).

3. Upravljanje talasima (WaveSpawner)

Napravićemo novi prazni objekat na sceni koji će biti GameManager. On će biti zadužen za upravljanje kompletnom igrom, odnosno imaće komponente (skripte) koje će upravljati mehanikom igre. Npr. Postavljanjem kula, statistikom igrača, vremenskim tokom igre, kamerama i talasima neprijatelja.

Ideja je da neprijatelji dolaze u talasima. Sledeći talas može da krene tek kada tekući bude prazan, tačnije kada svi neprijatelji iz jednog talasa budu ubijeni ili budu izašli sa mape. Nakon što prođu sve vrste neprijatelje (tri u ovom slučaju) skalira se težina neprijatelja i nastavlja se istim redosledom. Round robin princip.

Wavespawner komponenta od atributa ima niz tipa Wave (talas). Wave je ništa drugo do jedna klasa koja sadrži podatke o talasu. Prefab neprijatelja, broj neprijatelja u talasu i brinu kojom se stvaraju. Mesto sa kog kreće talas (SpawnPoint – START obejkat sa mape), vreme između talasa i dve statičke promenljive. Jedna koja predstavlja broj trenutno živih neprijatelja i jedna koja predstavlja broj punih krugova (promene svih neprijatelja). Takođe, Wavespawner će imati i User Interface elemente o kojima će biti reči. Pomoću njih će biti prikazani na ekranu trenutni talas, vreme do dolaska sledećeg, itd.

U Start() metodi elementa niza ćemo prebaciti u red radi lakše manipulacije. Elemente niza je moguće postavljati iz inspectora.

```
76  private void Start()
77      {
78          currentWave.text = waveIndex.ToString();
79          fullWaves = 0;
80          for (int i = 0; i < waves.Length; i++)
81          {
82              red.Enqueue(waves[i]);
83          }
84      }
```

Pokretanje talasa će se obavljati kao korutina, nezavisno od Update() metode i vremena. Zbog vremenske zavisnosti talasa, frekvencije stvranja neprijatelja.

Update() metoda se izvršava samo u slučaju da je broj trenutno živih neprijatelja (statička promenljiva koji smo umanjivali kada neprijatelj umre u Enemz.cs) manji ili jedan nuli.

Tada, kada vreme između talasa istakne pokreće se korutina za stvaranje neprijatelja.

```
85      private IEnumerator SpawnWave()
86      {
87          PlayerStats.rounds++;
88          currentWave.text = PlayerStats.rounds.ToString();
89          Wave wave = red.Dequeue();
90          EnemiesAlive = wave.count;
91          red.Enqueue(wave);
92
93          for (int i = 0; i < wave.count; i++)
94          {
95              SpawnEnemy(wave.enemy);
96              yield return new WaitForSeconds(1f / wave.rate);
97          }
```

```
98
99          waveIndex++;
100
101         if(waveIndex % waves.Length == 0)
102         {
103             fullWaves++;
104         }
105     }
```

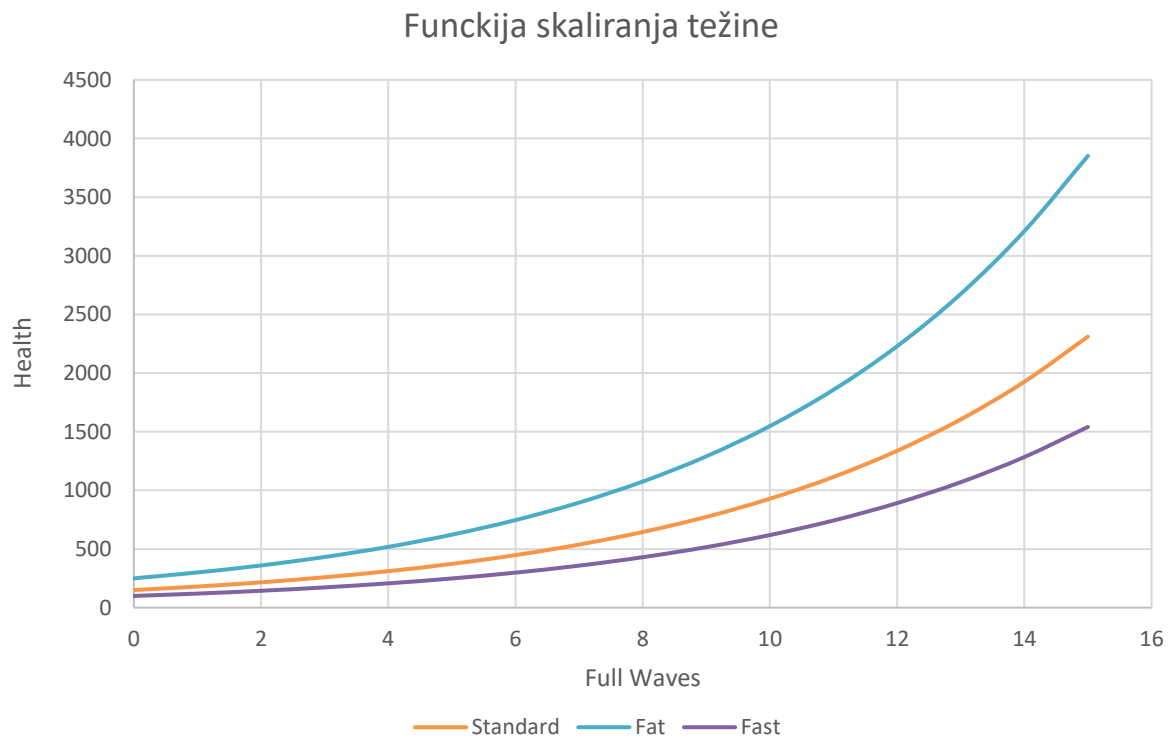
Iz reda se uzima prvi talas i odmah stavlja na kraj reda. Za trenutni talas stvaraju se neprijatelji u zadatim intervalima. Kada su svi neprijatelji stvoreni inkrementira se brojač talasa i u slučaju da su svi tipovi neprijatelja prošli povećava se i broj krugova.

3.1. Skaliranje

Progresija kroz igru ogleda se u skaliranju težine neprijatelja i preživljavanju što većeg broja talasa. Skaliranje težine vrši se povećanjem početne veličine životnih poena neprijatelja. Najočigledniji način, uzimanje Enemy komponente talasa i promene maksimalnog života ne daje dobre rezultate. Promena atributa neprijateljskog prefab objekta je permantentna, što znači da i nakon restarta igre objekti zadržavaju promenjene atribute. Tačno rešenje ovog problema je funkcija inicijalizacije životnih poena. Pri stvaranju svakog neprijatelja (Enemy.cs) u Start() metodi poziva se funkcija koja će postaviti vrednost životnih poena neprijatelja:

```
106     public void SetHealth()
107     {
108         health = maxHealth * Mathf.Pow(1f + difScale, WaveSpawner.fullWaves);
109     }
```

Procentualno uvećava životne poene (health) sa samog početka igre na osnovu broja punih krugova talasa (full waves) i faktora skaliranja težine.



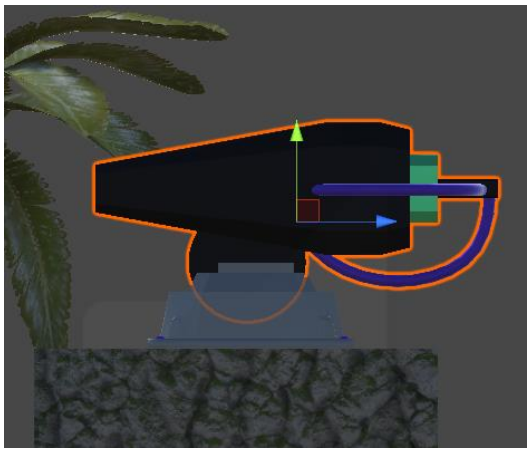
4. Kule i logika kula

4.1. Modeli i kreiranje kula

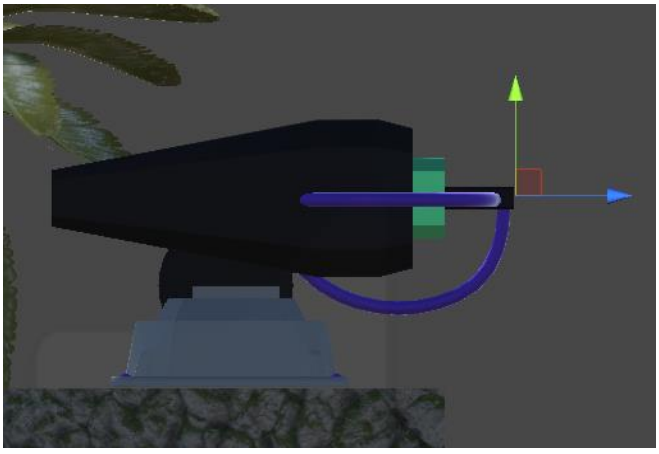
Import modela kula je identičan importu modela neprijatelja. Svaki importovani model je potrebno pretvoriti u prefab i sve dalje promene se vrše isključivo nad prefabom. U ovom slučaju postoji četiri vrste kula:

- Standardna
- Raketa
- Laser
- Mitraljez

Svaka kulica će fizički biti sastavljena od više delova. U root objektu kule nalaze se dva objekata, jedan koji je baza i jedan sa nazivom PartToRotate, koji se nalazi na bazi i predstavlja objekat koji će se rotirati u pravcu neprijatelja u kog cilja. Ovaj objekat je roditelj delovima kule koji su fizički vidljivi, top kule ili bilo koji objekat koji je potrebno okrenuti ka neprijatelju. Kako je deo koji se rotira roditelj sva deca objekti će pratiti njegove promene. Unutar dela koji se rotira biće i kamera koja će omogućiti igraču pogled iz perspektive kule (detaljnije u poglavlju o kameri).



Slika 3: PartToRotate objekat



Slika 4: FirePoint objekat - dete PartToRotate

FirePoint objekat je još jedan prazni objekat koji se nalazi unutar dela koji se rotira i predstavlja početnu poziciju ispaljivanja metkova. Pored importa modela potrebno je importovati i modele metkova za kule ili ih napraviti. Prilikom pucanja, model metka će biti instanciran sa početnom pozicijom u tački FirePoint. Metak će biti nosilac informacija (štete, efekata…) između kule i neprijatelja. Kule pored štete mogu delovati na neprijatelje na bilo koji način. Npr. laser koji pored štete koju nanosi tokom vremena svojim delovanjem i usporava neprijatelja. Svaka kula će imati i opciju unapređenja što će joj pojačati atribute. Kule se mogu skalirati slično kao i neprijatelji s tim što bi u ovom slučaju bilo dobro da ta promena bude i vizuelna. Tako da treba napraviti i po jedan prefab za unapređenu verziju svake od kula.

Nakon kreiranja modela svih tipova kula u obliku prefab-a treba razmisliti o logici.

4.2. Logika kula

Svakom kulom će upravljati Turret.cs skripta. Zajednički atributi kula su kamera iz perspektive kule, brzina rotacije i domet. Takođe, kula mora da ima i reference na objekte dela za rotaciju i FirePoint tačke kao i tag objekta koji treba da gađa (Enemy tag).

```
110     public Camera secondaryCamera;
111
112     [Header("General")]
113     public float range = 15f;
114     public float turnSpeed = 10;
115     public string type;
116     public AudioSource audioEffect = null;
117     [Header("Setup")]
118     public Transform partToRotate;
119     public string enemyTag = "Enemy";
120     public Transform firePoint;
```

Kule možemo podeliti na ona sa projektilima i lasere. U zavisnosti od tipa Turret.cs skripta će imati i pozivati različite atribute i metode. Tip kule se određuje jednom bool promenljivom, Use Laser, koja označava da li kula koristi laser ili ne.

4.2.1. Projektili

Bool promeljiva nije čekirana i kula koristi projektile. Sledi da kula kao atribut mora da ima reference na objekat projektila (prefab) i brzinu ispaljivanja projektila.

```
121     [Header("Use Bullets (default)")]
122     public float fireRate = 1f;
123     public GameObject bulletPrefab;
```

Projektili nose informacije od kule do neprijatelja. To implicira da projektil mora da ima svoj kontroler. Skripta Bullet.cs je komponenta projektila koja interaguje sa neprijateljima. Prenosi poruke kule neprijatelju. Projektil od atributa mora da ima štetu koju nanosi, brzinu kojom putuje i efekat pri udaru. U slučaju rakete, projektil mora da ima i radijus eksplozije u kojem nanosi štetu neprijateljima.

```
124     public float speed = 70f;
125     public int dmg = 55;
126     public GameObject impactEffect;
127     public float explosionRadius = 0f;
```

Takođe, mora imati i referencu na metu koju prati.

Dakle, kula pronalazi svoju metu tako što iz niza neprijatelja, koji dobija tako što kupi sve objekte sa scene koji imaju tag neprijatelja nalazi neprijatelja na najmanjem rastojanju od nje. Pri postavljanju funckije u Start() metodi pozivanje ove funkcije je postavljeno na dva puta u sekundi, radi smanjenja nepotrebnih iteracija kroz niz. Meta se setuje na neprijatelja koji je ispunio ove uslove.

Slecećи korak je okretanje kule ka meti koje radi LockOnTarget() metod. PartToRotate objekat se pozicionira u pravcu vektora od kule do mete.

Brzina kojom kula može da ispaljuje projektila je data fireRate-om. Potrebno je voditi računa o vremenu proteklom od poslednjeg ispaljenog metka. Uvođenjem posebne promenljive za to u svakoj iteraciji Update() metode merimo proteklo vreme i samo u slučaju da je vrednost promenljive manja ili jednaka nuli moguće je ispaliti projektil. Pozivom

metode Shoot(), instancira se novi objekat čija je referenca u bulletPrefab atributu i poziva se Seek() metod projektila koji je deo Bullet.cs komponente projektila.

Nakon ispaljenog projektila, kula je do sledećeg trenutka kada može da puca završila sa radom. Sada projektil nosi informacije (štetu i efekat) ka neprijatelju.

Metod Seek() Bullet.cs komponente postavlja nosi refrencu na metu koju projektil juri. U svakom frejmu projektil se pomera ka meti na isti način na koji se neprijatelj kreće ka putokazu. U slučaju da je je vektor od projektila do mete manji od razdaljine koju projektil prelazi u jednom frejmu projektil pogađa metu. U slučaju pogotka treba proveriti radijus eksplozije. Ako je on nula štetu prima samo pogođeni neprijatelj. U suprotnom, štetu primaju svi neprijatelji u zadatom radijusu. Pozivom funkcija Damage() i Explode() respektivno, nanosi se šteta meti ili u slučaju Explode() metode uzimaju se sve collider komponente u radijusu i ako je njihov tag tag neprijatelja poziva se metod Damage() za taj objekat.

```
128     private void Explode()
129     {
130         Collider[] colliders = Physics.OverlapSphere(transform.position,
    explosionRadius);
131         foreach (var colider in colliders)
132         {
133             if(colider.tag == "Enemy")
134             {
135                 Damage(colider.transform);
136             }
137         }
138     }
139 
140     void Damage(Transform enemy)
141     {
142         EnemyMovement e = enemy.GetComponent<EnemyMovement>();
143         if(e != null)
144             e.TakeDamage(dmg);
145     }
```

Metod Damage() poziva TakeDamge() metod Enemy.cs skripte opisane u delu o neprijateljima.

Nakon udara, projektil biva uništen.

4.2.2. Laser

Jedina razlika lasera i kula sa projektilom je upravo projektil. Laser umesto toga ima komponentu LineRenderer. Ova komponenta služi za iscrtavanje linija od tačke do tačke. U slučaju da je bool promenljiva UseLaser aktivna kula će imati dodatne atribute.

```
146     [Header("Use Laser")]
147     public bool useLaser = false;
148     public int dmgOverTime = 30;
149     public string effectType = "none";
150     public float effectPower = 0.45f;
151     public LineRenderer lineRenderer;
152     public ParticleSystem impactEffect;
153     public Light impactLight;
154     public AudioSource sound;
```

U metodi Update() u slučaju da postoji meta i da je korišćenje lasera aktivno poziva se LineRenderer komponenta čija referenca se zadaje u inspector objekta i aktivira. Umesto

metoda Shoot() poziva se metod Lase(). Za metu se poziva funckija TakeDamege() koja u ovom slučaju nanosi štetu u zavisnosti od vremena. Dok je laser zaključan na meti ona će primate štetu u jedinici vremena (dmgOverTime atribut). Takođe, na metu će biti primenjen i efekat (npr. usporeno kretanje) ako laser ima neki specijalni efekat. Ovo se postiže pozivanjem metode ApplyEffect() Enemy.cs komponente.

```
155     private void Laser()
156     {
157         targetEnemy.TakeDamage(dmgOverTime * Time.deltaTime);
158         targetEnemy.ApplyEffect(effectType, effectPower);
159 
160         if (!lineRenderer.enabled)
161         {
162             lineRenderer.enabled = true;
163             impactEffect.Play();
164             impactLight.enabled = true;
165         }
166         lineRenderer.SetPosition(0, firePoint.position);
167         lineRenderer.SetPosition(1, target.position);
168 
169         sound.Play();
170 
171 
172         Vector3 dir = firePoint.position - target.position;
173 
174         impactEffect.transform.position = target.position + dir.normalized *
    transform.localScale.x / 2;
175         impactEffect.transform.rotation = Quaternion.LookRotation(dir);
176     }

177 
178 public void ApplyEffect(string type, float amount)
179     {
180         switch (type)
181         {
182             case "slow":
183                 {
184                     speed = startSpeed * (1f - amount);
185                     break;
186                 }
187             default:
188                 break;
189         }
190     }
```

Laser() iscrtava liniju od FirePoint tačke na kuli do površine neprijatelja i instancira efekat pri udaru (opciono). Izgledom linije manipuliše se iz LineRenderer komponente kule sa laserom.

5. Postavljanje kula, unapređenje i prodaja (BuildManager)

Šta je ideja kod postavljanja kula? Igrač mora na neki način da izabere kulu koju želi da izgradi i postavi je na željeno mesto. Ova funkcionalnost igraču omogućava korisnički interfejs koji treba napraviti. Pored postavljanja kula igrač mora da ima mogućnost da tu kulu unapredi i proda. Igrač će putem korisničkog interfejsa komunicirati sa logikom igre.

Node objekte ćemo dopuniti Node.cs skriptom koja će biti zadužena za upravljanje pojedinačnim node objektima. Potrebno je voditi računa da li taj node ima sagrađenu kulu na sebi i da li je ona unapređena. U zavisnosti od toga, biće pozivane metode za izgradnju, unapređenje i prodaju kula. Takođe, Node.cs skripta upravlja i vizuelnom reprezentacijom selektovanja i prelaženja kursorom preko node obejkata, a klikom na neki node njegova referenca biva prosleđena BuildManager-u i smeštena u SelectedNode promenljivu.

5.1. Postavljanje kula

Unutar nekog kanvasa napraviti panel i u njemu po dugme za svaku vrstu kule. Panel (u nastavku Shop) će biti spona između igrača i igre. Shop će imati kao komponentu skriptu Shop.cs koja će biti zadužena za izbor kule. Za svaku vrstu kule, radi lakše implementacije, napravićemo klasu TurretBlueprint koja predstavlja šemu kule.

```
191 [System.Serializable]
192 public class TurretBlueprint {
193
194     // Use this for initialization
195     public GameObject prefab;
196     public int cost;
197     public GameObject upgradedPrefab;
198     public int upgradeCost;
199
200     public int getSellValue()
201     {
202         return cost / 2;
203     }
204 }
```

Skripta Shop.cs će od atributa imati reference na šeme za svaku vrstu kule (TurretBlueprint) i metode za izbor svake od kula.

U GameManager objekat (koji upravlja igrom) dodaćemo jos jednu komponentu. Skriptu BuildManager.cs koja će upravljati izgradnjom. BuildManager je singlton, ima jednu statičku promenljivu koja je referenca na samog sebe.

BuildManager upravlja izgradnjom. Potrebno je da ima informacije o tome šta pravi i gde pravi. Tako da će postojati promenljiva TurretToBuild (kula koju treba da sagradi) koja je tipa TurretBlueprint i trenutno selektovani node, SelectedNode. Po izboru kule iz Shop objekta poziva se metod za izbor selektovane kule.

```
205     public void SelectStandardTurret()
206     {
207         buildManager.SelectTurretToBuild(standardTurret);
208     }
```

Referenca se prosleđuje BuildManager-u koji je postavlja u promenljivu TurretToBuild. Klikom na neki node, poziva se BuildTurret() metod Node.cs skripte u slučaju da je moguće sagratiti kulu na tom mestu i SelectedNode se postavlja na node na kome je sagrađena kula. Naravno, pre izgradnje potrebno je proveriti da li igrač ima dovoljno novca za to. Svakim klikom na neki node menja se selektovani node, novim klikom na isti node on više neće biti selektovan.

Slika 5: Shop

Svako dugme Shop objekata predstavlja jednu vrstu kule. Dugme ima OnClick() događaj na koji će biti pozvana odgovarajuća funckija odabira kule (TurretBlueprint) Shop.cs skripte.

5.2. Unapređenje i prodaja kula

Ideja za unapređenje kula je identična onoj za izgradnju. Klikom na neko dugme pozvaće se funckija koja će je unaprediti.

Treba napravit (analogno Shop objektu) neki objekat koji će predstavljati korisnički interfejs vezan za Node. Njime će biti omogućena manipulacija node objektima, odnosno, onim što se na njemu nalazi. Ta komponenta će se zvati NodeUI i u sebi će imati panel sa dva dugmeta: Upgrade (unapred) i Sell (prodaj). NodeUI objektom će upravljati NodeUI.cs skripta a sam obejkat će na početku biti isključen (disable). Kada neki node bude selektovan (a node je moguće selektovati kada nešto postoji na njemu) biće prikazan i korisnički interfejs vezan za taj node, odnosno, opcija unapređenja i prodaje kule.

Prilikom selektovanja node objekta BuildManager koji ima referencu na NodeUI skriptu prosleđuje joj selektovani node. To znači da će NodeUI.cs skripta imati refrencu na node za koji treba da prikaže kontrole. Kada dobije node postavlja ga u promenljivu (Target) i preko nje pristupa metodama Node.cs kao što su UpgradeTurret() i SellTurret(). Kada postoji selektovani node, korisnički interfejs vezan za njega je aktivan. Kada nema selektovanog noda korisnički interfejs je deaktiviran do sledećeg selektovanja.

NodeUI.cs skripta ima dve metode: Upgrade() i Sell(). Respektivno su vezane za Upgrade i Sell dugme korisničkog interfejsa na već opisani način u 5.1. Klikom na jedno dugme poziva se UpgradeTurret() ili SellTurret() metoda Node.cs skripte vezane za selektovani node prosleđen iz BuildManager-a. U slučaju prodaje, igrač dobija 50% uloženog novca.

```
209 public void Upgrade()
210 {
211     target.UpgradeTurret();
    BuildManager.instance.DeselectNode
    ();
212 }
```

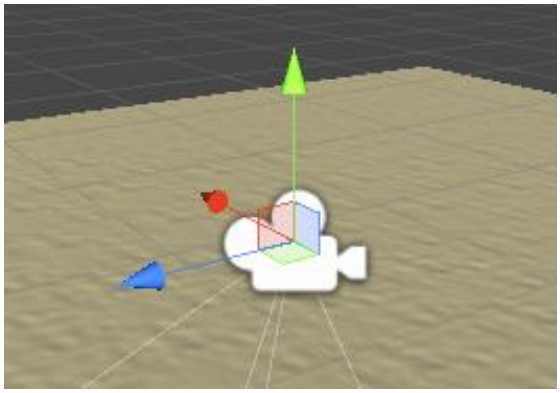
```
    public void Sell()
213 {
214     target.SellTurret();
215     target.isUpgraded = false;
    BuildManager.instance.DeselectNode
    ();
216
217 }
```

6. Kamera

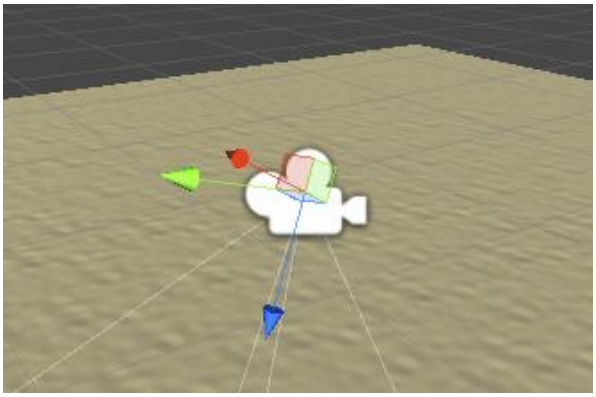
Igrač ima dve mogućnosti, može da gleda igru kroz glavnu kameru ili iz prepsektive željene kule. Glavnom kamerom moguće je upravljati i tako posmatrati scenu iz svih uglova, dok su sporedne kamere zakačene za kule.

6.1. Glavna kamera

Glavna kamera se sastoji od dva objekta, root objekta koji je nosač kamere (CameraRig) i dete objekta same kamere. Nosač kamere je prazni objekat, pozicioniran je tako da bude normalan na ravan XZ globalnog koordinatnog sistema. Kamera samim tim što se nalazi unutar nosača ima istu poziciju ali može da ima drugi ugao. Tako da će kamera gledati na mapu.



Slika 7: Nosač kamere (CameraRig)



Slika 6: Glavna kamera (MainCamera)

Kamerom će upravljati skripta CameraController.cs koja je komponenta CameraRig objekta. Skripta ima sledeće atribute:

```
218     public Camera Kamera;
219     public float panSpeed = 30f; //pomeraj po X i Z
220     public float panBordedThickness = 10f;
221     public float scrollSpeed = 5f;
222     public float rotationSpeed = 7f;
223     public float minY, maxY;
224
225     private bool doMovement = true;
226     private Vector3 defaultPosition;
227     private Quaternion defaultRotation;
228     private Quaternion defaultCameraRotation;
```

U Start() metodi treba zapamtiti sve početne vrednosti pozicija i rotacija.

Pomeranje kamere vrši se translacijom nosača kamera. Nosač (zajedno sa kamerom) se translira kroz prostor, ugao pod kojim kamera gleda ostaje isti samo se promenila pozicija sa koje gleda. Pomeranje je implementirano na dva načina. Moguće je kretati se pomoću tastature, a moguće je i kretanje dovođenjem kursora miša do ivice ekrana, pravac kretanja zavisi od ivice.

U Update() metodi, u svakom frejmu proverava se da li igrač drži neko dugme ili je pozicija miša u blizini neke od ivica (panBorderThickness).

```
229 if (Input.GetKey("w") || Input.mousePosition.y >= Screen.height -
    panBordedThickness)
230         {
231             transform.Translate(Vector3.forward * panSpeed * Time.deltaTime,
    Space.Self);
232         }
233         if (Input.GetKey("s") || Input.mousePosition.y <= panBordedThickness)
234         {
235             transform.Translate(Vector3.back * panSpeed * Time.deltaTime,
    Space.Self);
236         }
237         if (Input.GetKey("d") || Input.mousePosition.x >= Screen.width -
    panBordedThickness)
238         {
239             transform.Translate(Vector3.right * panSpeed * Time.deltaTime,
    Space.Self);
240         }
241         if (Input.GetKey("a") || Input.mousePosition.x <= panBordedThickness)
242         {
243             transform.Translate(Vector3.left * panSpeed * Time.deltaTime,
    Space.Self);
244         }
```

Kamera se kreće u odnosnu na svoj lokalni koordinatni sistem.

Zumiranje je isto što i translacija nosača kamera po Y osi. Funkcija koja uzima input sa kružića miša vraća realan broj iz intervala [-1,1]. Broj predstavlja brzinu i smer okretanja kružića. Tako da se zumiranje vrši redukovanjem vrednosti Y komponente vektora pozicije nosača kamere.

```
245         float scroll = Input.GetAxis("Mouse ScrollWheel");
246         Vector3 pos = transform.position;
247 
248 
249         pos.y -= scroll * 1000 * scrollSpeed * Time.deltaTime;
250         pos.y = Mathf.Clamp(pos.y, minY, maxY);
251 
252         transform.position = pos;
```

Clam() funckijom postavljamo granične vrednosti kako ne bi bilo moguće zumiranje do samog dna mape kao i suprotno, zumiranje u beskonačnost od mape.

Rotacijom je kretanje kamere upotpunjeno i moguće je posmatranje scene iz bilo kog ugla. Rotacija oko Y ose (levo-desno) vrši se transformacijom nosača kamere. Međutim, ako menjamo ugao pod kojim kamera gleda, moramo vršiti transformacijom glavne kamere unutar nosača (MainCamera).

```
253 if (Input.GetKey("c"))
254   {
255       Quaternion eu = new Quaternion(rotationSpeed * Time.deltaTime, 0, 0, 1);
256       Kamera.transform.Rotate(eu.eulerAngles, Space.Self);
257   }
258   if (Input.GetKey("z"))
259   {
260      Quaternion eu = new Quaternion(-rotationSpeed * Time.deltaTime, 0, 0, 1);
261      Kamera.transform.Rotate(eu.eulerAngles, Space.Self);
262   }
263   if (Input.GetKey("e") )
264   {
265       Quaternion eu = new Quaternion(0, rotationSpeed * Time.deltaTime, 0, 1);
266       transform.Rotate(eu.eulerAngles, Space.World);
```

```
267    }
268    if (Input.GetKey("q"))
269    {
270        Quaternion eu = new Quaternion(0, -rotationSpeed * Time.deltaTime, 0, 1);
271        transform.Rotate(eu.eulerAngles, Space.World);
272    }
```

Nosač kamere se rotira oko Y ose, a kamera se rotira oko X ose. S tim što kameru treba rotirati oko X ose lokalnog koordinatnog sistema.

Kao dodatak, resetovanje kamere na početne vrednosti (poziciju i rotaciju) pritiskom na dugme (u ovom slučaju “r”). Kao i zaključavanje i otključavanje kamere pomoću promenljive doMovement. U slučaju da je promenljiva false, izaći (return) iz Update() metode. Promena stanja zaključavanja se takođe vrši pritiskom na dugme (u ovom slučaju “space”).

6.2. Promena kamere

Sekundarna kamera se nalazi u svakoj kuli i one imaju reference na svoje kamere, tako da je potrebno naći način za promenu trenutne aktivne kamere.

Proširivanjem GameManager objekta koji kontroliše igru dodavanjem skripte CameraManager.cs koja će kontrolisati kamere dobijamo željene funkcionalnosti.

CameraManager.cs treba da ima samo jedan atribut, referencu na glavnu kameru (MainCamera) koju će dobiti iz inspectora objekta i referencu na BuildManager. U slučaju da je potrebna informacija o aktivnoj kameri treba uvesti i jednu statičku promenljivu koja će to pamtiti.

Igrač menja kameru pritiskom na dugme (“v”). Iz BuildManager-a možemo da dodjemo do selektovanog node objekta. Ako postoji kula no tom objektu kamera te kule postaje aktivna. U slučaju da nijedan node nije selektovan ili da nema kule aktivna kamera je glavna kamera. Glavna kamera je uvek uključena dok su sve kamere na kulama isključene. Dubina (depth – postavlja se iz inspectora) glavne kamere je postavljena na -1 dok je kod ostalih kamera 0. To znači da će se kamera sa većom dubinom (kamera na kuli) iscrtati iznad glavne kamere, tako da kada su obe kamere uključene igrač će gledati kroz onu sa većom dubinom (kameru na kuli). Ponovnim pritiskom na taster za promenu, selektovanjem nove kule, izgradnjom, unapređenjem ili prodajom sekundarna kamera se isključuje i glavna kamera postaje ponovo aktivna.

```
273    public void ChangeCamera()
274    {
275        Node n = buildManager.GetSelectedNode();
276        if (n != null)
277        {
278            Turret t = n.turret.GetComponent<Turret>();
279            t.secondaryCamera.enabled = !t.secondaryCamera.enabled;
280            ActiveCamera = t.secondaryCamera;
281        }
282        else
283        {
284            mainCamera.enabled = true;
285            ActiveCamera = mainCamera;
286        }
287    }
```

7. Korisnički interfejs

Prodavnica (shop) je već opisana u poglavlju o kulama. Drugi deo korisničkog interfejsa pokazuje statusne informacije o igraču.

U istom kanvasu u kom je i prodavnica treba napraviti novi panel koji će biti pozicioniran u gornjem delu ekrana. Unutar panela treba napraviti sledeče UI objekte:

- Text objekat koji će prikazivati vreme do novog talasa (Time)
- Text objekat koji će prikazivati stanje novca igrača (Money)
- Image objekat sa tekstom koji će predstavljati preostale živote igrača (Lives)
- Image objekat sa tekstom koji će predstavljati trenutni talas (Waves)

Napravljene objekte treba dati kao reference skriptama koje će ažurirati informacije koje oni prikazuju. To može biti bilo koja skripta ili bilo koja Update() metoda pošto su sve promenljive sa informacijama statičke. U ovom slučaju, ažuriranje podataka se vrši u Update() metodi Wavespawner.cs skripte. Dat je primer ažuriranja jednog Text objekta, identično je za ostale.

```
288         public Text waveCountDownText;

289         countdown -= Time.deltaTime;
290         countdown = Mathf.Clamp(countdown, 0f, Mathf.Infinity);
291         waveCountDownText.text = string.Format("{0:00.00}", countdown);
```

8. Tok igre

Prirodni nastavak na korisnički interfejs je način na koji igrač utiče na tok igre. Igrač treba nekako da započne igru, treba da ima mogućnost zaustavljanja igre (pauze) i igra se u nekom trenutku treba završiti. Ovo se takođe postiže korisničkim iterfejsom, koji ima određenu dubinu, odnosno, postoji neka logika ispod njega koja upravlja tokom igre s toga, treba biti obrađen kao posebna celina.

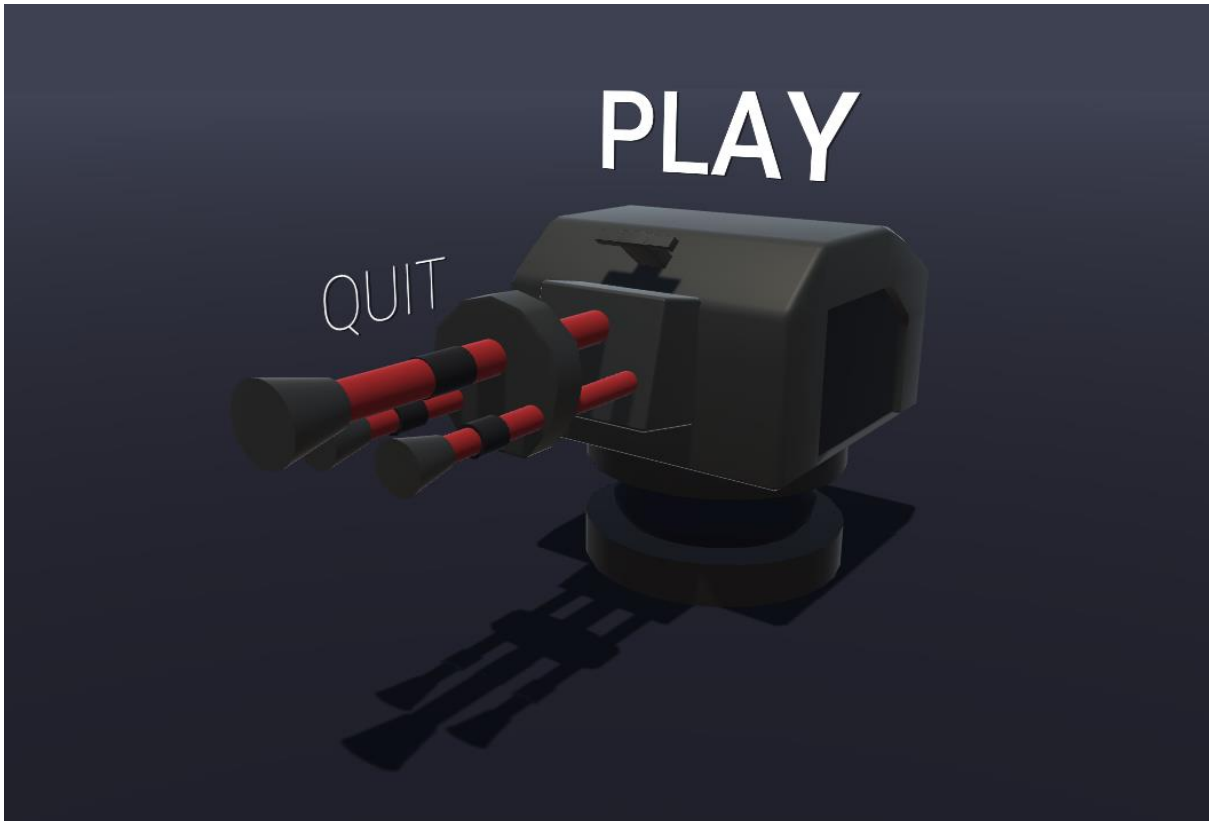
8.1. Početak igre

Igrača pre početka igre treba da sačeka neki meni koji će mu ponuditi opcije. Odabirom opcije Play započinje igru.

Meni koji će igrač prvo videti implemetira se kao nova scena. U Unity engine-u je moguće napraviti više scena i menjati ih po potrebi. Nova scena (MainMenu) će imati samo glavnu kameru i input elemente koje će igrač koristiti.

Kako će meni izgledati nije bitno, pozabavićemo se funkcionalnostima. Izgled menija je krajnje subjektivan.

Glavnim menijem će upravljati skripta koja će biti vezana za neki prazni objekat. Jedino što MainMenu.cs skripta treba da zna je scena koju treba da učita nakon što igrač klikne na dugme Play. MainMenu u ovoj implementaciji poseduje dugme Play i dugme Quit, analogno tome, skripta MainMenu.cs će imati metode Play() i Quit().



Slika 8: Main Menu

Napomena: U slučaju da želite da se dugme nalazi u 3D svetu (ne bude zalepljeno na ekran) RenderMode kanvasa u kom se nalazi tekst mora da bude World Space (dostupno u inspectoru).

```
292 public class MainMenu : MonoBehaviour {
293
294     public string levelToLoad = "Level01";
295     public SceneFader sceneFader;
296
297    public void Play()
298     {
299         sceneFader.FadeTo(levelToLoad);
300     }
301
302     public void Quit()
303     {
304         Application.Quit();
305     }
306 }
```

8.2. Pauza

Igraču treba opcija da pauzira igru. Nameće se GameManager kao upravljač igrom jedino mesto gde možemo implementirati takvu logiku. Proširićemo ga skriptom PauseMenu.cs koja će upravljati pauzom.

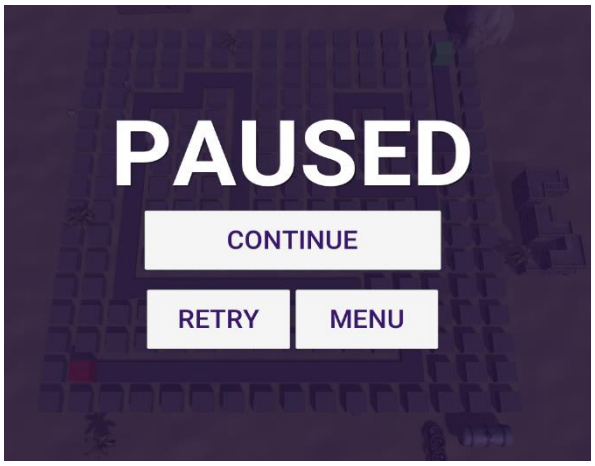
Korisniči interfejs, odnostno meni za pauzu (PauseMenu) treba napraviti kao i glavni meni. Postaviti input objekte i za svakom od njih dodeliti metod iz PauseMenu.cs skripte koju će pozvati na klik. Jednostavn meni za pauzu ima opcije za nastavak, restart i glavni meni. Analogno tome, PauseMenu.cs skripta će imati metode koje odgovaraju funckionalnostima ograničenim inputom.

Meni za pauzu je na početku isključen, pritskom na neko dugme (“esc”) uključuje se. Kada je pauza aktivan treba zaustaviti igru. To se postiže postavljanjem brzine protoka vremena na 0. Data je funkcija koja uključuje i isključuje prikaz menija i zaustavlja ili nastavlja igru.

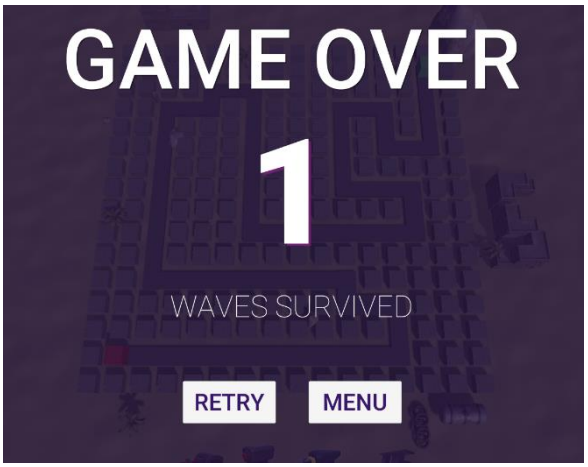
```
307     public void Toggle()
308     {
309         ui.SetActive(!ui.activeSelf);
310
311         if(ui.activeSelf)
312         {
313             Time.timeScale = 0f;
314         }
315         else
316         {
317             Time.timeScale = 1f;
318         }
319     }
```

Klikom na dugme za nastavak igre poziva se funkcija Toogle() koja vraća protok vremena na 1 (podrazumeva se da je pre toga bila pauza, inače igrač ne bi mogao da klikne na to dugme).

Nova igra (Retry) i odlazak na glavni meni su identični učitavanju scene u MainMenu komponenti. S tim što Retry() metod učitava istu scenu, vraća vreme na 1 i postavlja broj živih neprijatelja na -1 (statička promenljiva WaveManger.cs).



Slika 10: Meni za pauzu



Slika 9: Kraj igre

8.3. Kraj igre

Igra se završava kada igrač ostane sa 0 života. Život gubi kada neprijatelj stigne do kraja puta.

Kada igrač ostane bez života treba zaustaviti igru, prikazati mu određene statističke podatke i ponuditi mu opcije da igra ispočetka ili da se vrati na glavni meni.

Ideja je identična ideji menija za pauzu. Treba napraviti UI objekat (GameOver) koji će igraču prikazati do kog talasa je preživeo i ponuditi mu opcije. Takođe, ovaj objekat je na početku isključen. GameManager treba proširiti GameOver.cs skriptom koja će voditi računa o kraju igre. U svojoj update metodi proveravaće da li igrač ima života na raspolaganju i u slučaju da ih više nema aktiviraće GameOver prikaz.

```
320     private void Start()
321     {
322         gameIsOver = false;
323     }
324 
325     void Update () {
326 
327         if (gameIsOver)
328             return;
329 
330         if(PlayerStats.lives <= 0)
331         {
332             EndGame();
333         }
334     }
335 
336     private void EndGame()
337     {
338         Time.timeScale = 0f;
339         gameIsOver = true;
340         gameOverUI.SetActive(true);
341     }
```

Igrač je završio igru i ima opcije za novu igru i glavni meni koje su implementirane kao i kod menija za pauzu.