Izrada RTS igre u Unity okruženju

Kekić, Kristijan

Sveučilište Jurja Dobrile u Puli
Fakultet Informatike u Puli

Kristijan Kekić

Izrada RTS igre u Unity okruženju

Diplomski rad

Pula, rujan, 2024.

Sveučilište Jurja Dobrile u Puli

Fakultet Informatike u Puli

Kristijan Kekić

Izrada RTS igre u Unity okruženju

Diplomski rad

JMBAG: 0303082334, redoviti student

Studijski smjer: Informatika

Kolegij: Dizajn i programiranje računalnih igara

Znanstveno područje: Društvene znanosti

Znanstveno polje: Informacijske i komunikacijske znanosti

Znanstvena grana: Informacijski sustavi i informatologija

Mentor: izv. prof. dr. sc. Tihomir Orehovački

Pula, rujan, 2024.

Sadržaj

1. Uvod

Igre te općenito industrija igara su u današnje vrijeme široko rasprostranjeni s obzirom na raširenost elektroničkih uređaja (računalo, mobitel, konzole) te možemo reći da je sigurno svatko od nas imao doticaj s njima.

Ovaj diplomski rad sadržava proces izrada 3D računalne igre strategije u stvarnom vremenu u platformi Unity. Unity je platforma koja podržava stvaranje 2D i 3D računalnih, mobilnih te igara za konzole, različitih žanrova [1]. Unity je također veoma prikladan za početnike u razvoju videoigara. Neke od danas veoma popularnih igara koje su napravljene u Unity okruženju su Genshin Impact, Among Us, Fall Guys i Pokemon Go. Verzija Unity okruženja korištena u ovom radu je 2022.3.4f1.

Motivacija za pisanje ovog rada je bila želja za upoznavanjem procesa razvoja igara u Unity okruženju koje je danas veoma raznoliko te se i dalje razvija i proširuje. Iako igre strategije u stvarnom vremenu nisu toliko popularne poput FPS (First person shooter) ili RPG (Role playing game) igara, one i dalje predstavljaju izazov svakome tko se upusti u njih jer je potrebno vremena i pokušaja da se dođe do strategije kojom će se poraziti protivnik.

Cilj rada je prikazati i objasniti elemente tijekom razvoja igre Peasants Evolve poput upravljanja resursima, stvaranje jedinica, gradnjom zgrada i jednim od ključnih elemenata ovakvih igara, magla rata (Fog of war).

Svaki od ovih nabrojanih elemenata ima svoju ulogu u strategiji koju će igrač razviti, te se ne bi trebalo davati poseban naglasak bilo kojem dijelu, nego bi se trebalo gledati na to kao na cjelinu.

Nastavak rada sadržava slične igre koje su služile kao inspiracija za izradu ove igre kao Age of Empires, Cossacks te će biti objašnjeni elementi igara strategije u stvarnom vremenu koji su bili korišteni u ovom radu.

2. Unity

Unity postoji još od 2005. godine, proizveden od Unity Technologies (prije toga su se zvali Over the Edge Entertainment). Tijekom godina ovo okruženje se širilo, postajući danas jednim od najpopularnijih okruženja za izradu igara. 17 platformi podržava aktivnosti izrađene u ovom okruženju te glavni fokus imaju mobilne igre. U nastavku će biti opisane neke od komponenta Unity okruženja:

2D i 3D grafika

Unity okruženje podržava izradu igara u 2D i u 3D-u, gdje se u 2D izradi koriste sličice ili sprite-ovi, dok 3D modeli mogu koristiti prilagođene materijale i teksture. 3D grafika također može imati generirani teren koji se može oblikovati po želji, te se može izmijeniti osvjetljenje, dinamički efekti i post-processing efekti (poput sjaja, zamućivanja, kontrasta, dodatnih sjena) [5].

Fizika

Pogon igre u Unity-u omogućuje da se objekti kreću ispravno. Sudari, gravitacija i ostale sile prirode su upravljani od strane pogona igre. Postoje 2 pogona, objektno-orijentirani i tehnološki. Objektno-orijentirani je postavljen da se koristi kao početni te se vrti na jednoj jezgri i dretvi, dok tehnološki pogon koji je usmjeren podatke koristi arhitekturu koja je brža, lakša i optimiziranija za višedretveni rad.

Skripte

Programski jezik koji se koristi u Unity-u je C#. Skripte su dio Unity-a te upravljaju hijerarhijom objekata te organiziraju događaje tako da se odvijaju u pravo vrijeme. Skripte kontroliraju fizičko ponašanje objekata i stvaraju grafičke efekte, te upravljaju AI sustavom [5].

Zvuk i videozapis

Unity pruža alate za miješanje i usavršavanje zvuka s pretpostojećim efektima. Videozapis komponenta omogućuje programerima da objedine videozapise u svoj projekt. Umjetnici i programeri mogu ubacivati vlastito napravljene videozapise i audiozapise u svoj Unity projekt.

Animacija

Animiranje objekata i postavljanje značajki objekata ima jednostavan tijek rada, te ga zbog toga neki ljudi zovu Mecanim. Postoji HAnim (Humanoid Animation), odobrena ISO i IEC

standardna specifikacija za modeliranje i animaciju humanoida. Koristi se za definiranje međusobno zamjenjivih ljudskih figura. Jednom definirane, ove figure mogu se koristiti u različitim igrama.

Umjetnici i animatori koriste preview mode (način pregleda) kako bi pregledali svoj rad prije nego što ga predaju programerima za dodavanje kodova za gameplay. Preview mode omogućava umjetnicima određenu neovisnost od programera jer mogu testirati i fino podešavati isječke, prijelaze i interakcije unaprijed.

Animiranje lika uključuje primjenu kodirane logike na dijelove tijela. Može se primijeniti jednu animacijsku skriptu na različite dijelove tijela. Nakon toga, tu ista skripta se može primijeniti na mnoge likove. Ovo je praktično jer omogućuje ponovnu upotrebu animacijskog koda, umjesto da se troši vrijeme na jedinstvene kodove za svaki dio tijela. Isječci razvijeni unutar i izvan Unity-a mogu se spojiti i integrirati u jednu Unity igru.

Korisničko sučelje

Postoje 3 opcije za razvoj korisničkog sučelja:

1. UIElements: Koristi se web tehnologija poput stilskih listova (stylesheets), dinamičko i kontekstualno upravljanje događajima te očuvanje podataka.
2. Unity UI: Jednostavan alat za raspoređivanje, pozicioniranje i stiliziranje korisničkog sučelja. To je objektno orijentirano sučelje koje koristi komponente i pregled u Game View-u.
3. IMGUI: Grafičko korisničko sučelje koje se koristi za izradu debugging prikaza unutar igre. Nije najbolja opcija za razvoj korisničkog sučelja.

Navigacija i pronalaženje puta

Navigacijska komponenta omogućuje programerima da kreiraju objekte koji se mogu kretati po svijetu igre. Postoje navigacijske mreže koje se automatski generiraju na temelju geometrije u sceni. Off-mesh poveznice omogućuje radnje poput otvaranja vrata ili skakanja s litice dok dinamičke prepreke pokreću prilagodbu putanje objekata tijekom izvođenja igre [2].

3. Igre istog žanra

Neke od najpopularnijih igara ovoga žanra su Age of Empires, Cossacks 3, Starcraft, Warcraft 3. U nastavku će se opisati Age of Empires i Cossacks 3.

3.1. Age of Empires

Age of Empires je jedna on najpoznatijih i najpopularnijih igara strategije u stvarnom vremenu. Datira još iz 1997., kad je napravljen prvi Age of Empires, te je dosad napravljeno 9 verzija igre Age of Empires. Fokus igara je na povijesnim događajima iz Europe, Afrike i Azije, u razdoblju od kamenog do željeznog doba.

Slika 1. Age of Empires 2

Davajući Age of Empires 2 kao primjer, i nakon 20 godina je jedna on najpopularnijih igara strategije u stvarnom vremenu. Ima dobro balansiranu stratešku igrivost da zadrži iskusne igrače zainteresiranima, a da pritom ne bude preteška početnicima. Koncepti poput raspoređivanja zgrada po mapi, upravljanja ograničenjem populacije, prikupljanja ključnih resursa i napredovanja kroz četiri tehnološka doba dodaju značajnu dubinu bez pretjerivanja s mikromenadžmentom. Sustav kontra-jedinica u stilu "kamen-škare-papir" između različitih vrsta pješadije, konjice, strijelaca i opsadnih oružja vodi do uzbudljivih bitaka gdje kreativna strategija često pobjeđuje brojčanu nadmoć ili jedinice višeg ranga. Age Of Empires 2 koja je prikazana na slici 1 impresionira igrače s više od 30 jedinstvenih civilizacija, od kojih svaka ima svoje posebne bonuse, specijalizirane jedinice i raznolika tehnološka stabla, omogućujući različite strateške pristupe. Iskusni igrači su usavršili precizne "redoslijede gradnje" za

ekonomski napredak, brze napade u ranoj fazi igre, dominaciju u kasnoj fazi igre, te strategije za kopnene i vodene mape [3].

3.2. Cossacks 3

Cossacks 3 je isto jedna od popularnih igara strategije u stvarnom vremenu. Izašla je 2016. godine. Nudi 5 povijesnih kampanja koje su bazirane na događajima iz 17. i 18. stoljeća. Postoji element nasumično generirane mape tako da je svaki novi ulazak u igru izazov. Također postoji mogućnost igranja online gdje se do 8 igrača mogu međusobno boriti.

Za razliku od Age of Empiresa gdje je limit nekoliko stotina do tisuću jedinica, Cossacks 3 može na mapi imat do 32000 jedinica. Brojnost jedinica se može vidjeti na slici 2. Može se birati od 12 zemalja koje ukupno imaju 70 različitih vrsta jedinica, 100 različitih istraživanja (koje poboljšavaju jedinice i zgrade) te 140 različitih zgrada, koje omogućavaju borbu na kopnu i na moru [4].

Slika 2. Cossacks 3

4. Implementacija igre

Peasants Evolve je igra strategije u stvarnom vremenu. Na početku igre igrač dobiva nekoliko jedinica (radnike), te određenu količinu resursa. Na drugoj strani mape se nalazi neprijatelj (AI), kojemu je cilj doći do igrača, te ga sa svojim jedinicama poraziti. Neprijatelj šalje vojsku u valovima, dok je na igraču da smisli kako se obraniti od tih valova napada te nakon toga da izvrši protunapad i pobijedi neprijatelja.

Igraču je cilj napraviti zgrade da dobije izvor resursa (jer zgrade daju resurse), i da može u njima trenirati jedinice koje će mu kasnije služiti za borbu. Izgradnja zgrada nije trenutačna te zahtijeva radnike da ju izgrade, što daje dodatni izazov igraču u odlučivanju prioriteta izgradnje zgrada.

Igrač može napraviti istraživanja vještina u igri, da si poboljša proizvodnju ili snagu jedinica, što mu može dati prednost u odnosu na neprijatelja. Postoje standardna poboljšanja koja s vremenom može naučiti te nije ograničen da istraži sve, ali također postoje i poboljšanja koja su usmjerena na napad, obranu i ekonomiju, od kojih svaki ima svoje prednosti. Igrač nije nužno limitiran na ta usmjerena poboljšanja, te može iz svake kategorije naučiti po par.

U ovom poglavlju će se opisati razvoj svih elemenata igre Peasants Evolve. Prvo će se pokazat će i objasniti kako funkcioniraju skripte vezane uz igrača koje uključuju menadžment resursa, gradnju zgrada, treniranje i kretanje jedinica, istraživanje vještine te maglom rata (Fog of war). Nakon toga će se opisati skripte za neprijatelja, kako on gradi zgrade i trenira jedinice, kako funkcionira borba te slanje jedinica. Također će se obraditi zvuk i uvjeti za kraj igre.

4.1. Pogled igrača

Pogled igrača je iz trećeg lica, iz ptičje perspektive. To je veoma bitno za ovakvu igru, jer osim što može vidjeti vlastitu bazu i jedinice na mapi, također ovisno o tome koliko su mu jedinice blizu neprijatelju, može vidjeti dali se neprijatelj približava, što mu daje znak da će neprijatelj ubrzo napasti. Za teksture tla i vode su korišteni paketi iz Unity Asset Storea [8][10].

Igrač može koristiti WASD tipke da se pomiče gore, dolje, lijevo i desno. Zumiranje i odzumiranje se postiže vrtnjom kotačića na mišu.

4.2. Fog of War

Fog of War ili magla rata je jedan od najbitnijih elemenata igara strategije u stvarnom vremenu. Dodaje razinu izazova i iznenađenja, jer se ne zna gdje se protivnik nalazi.

```
public class FogOfWarManager : MonoBehaviour
{
    public GameObject fogQuad;
    public float defaultRevealRadius = 10f;

    private List<GameObject> playerUnits;
    private List<GameObject> playerBuildings;
    private Vector4[] unitPositions;
    private float[] revealRadiuses;
```

Programski kod 1. FogOfWarManager

Za Fog of War je korišten 2D objekt Quad, koji služi da bi se sve sakrilo kao što se može vidjeti u programskom kodu 1. U slučaju da jedinice nemaju svoj određen radijus, postavlja se početni radijus. Vector4 UnitPositions pohranjuje pozicije svih jedinica koje trenutno otkrivaju mapu, dok revealRadiuses pohranjuje vrijednosti radijusa.

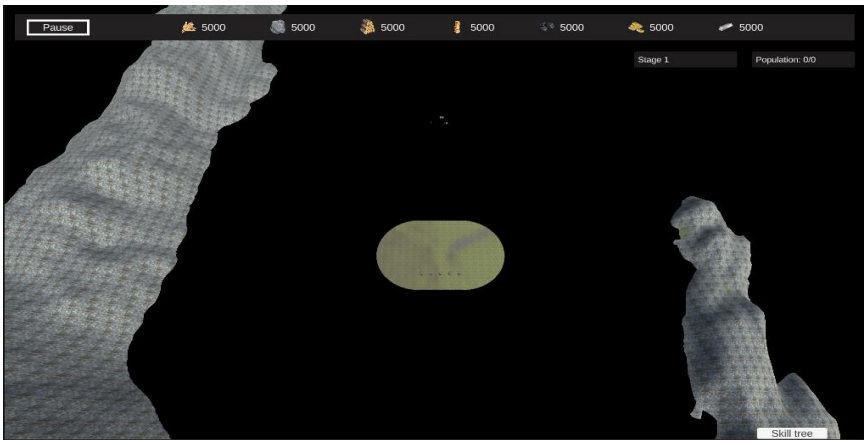
```
void Update()
{
    Material fogMaterial = fogQuad.GetComponent<Renderer>().sharedMaterial;

    playerUnits = SelectionManager.Instance.playerUnits;
    playerBuildings = SelectionManager.Instance.playerBuildings;

    List<GameObject> visionObjects = new List<GameObject>();
    visionObjects.AddRange(playerUnits);
    visionObjects.AddRange(playerBuildings);

    int objectCount = Mathf.Min(visionObjects.Count, unitPositions.Length);

    if (objectCount == 0)
    {
        if (fogQuad.activeSelf)
        {
            fogQuad.SetActive(false);
        }
    }

    for (int i = 0; i < objectCount; i++)
    {
        GameObject obj = visionObjects[i];

        Vector3 pos = obj.transform.position;
        unitPositions[i] = new Vector4(pos.x, pos.y, pos.z, 1);

        float revealRadius = defaultRevealRadius;
        revealRadiuses[i] = revealRadius;
    }

    fogMaterial.SetInt("_UnitCount", objectCount);
    fogMaterial.SetVectorArray("_UnitPositions", unitPositions);
    fogMaterial.SetFloatArray("_RevealRadiuses", revealRadiuses);
}
```

Programski kod 2. FogOfWarManager-2. dio

Fog of war i liste igračevih jedinica i zgrada se konstantno ažuriraju, te se vrijednosti tih listi ubacuju u listu visionObjects. ObjectCount sadržava broj objekata koje će shader obraditi. U petlji se događa obrada pozicija i radijusa, uzima se trenutna pozicija svake jedinice te se sprema u Vector4, dok se za radijus uzima početno postavljena vrijednost radijusa. Nakon toga se ti podaci proslijede shaderu koji ih dalje obrađuje, te on otkriva mapu. Sve ovo je prikazano u kodu 2.

4.3. Početno sučelje

Prva stvar koju igrač vidi nakon što uđe u razinu su resursi, faza u kojoj se nalazi, populacija i mapa sakrivena maglom rata. Dizajn igračevog početnog je dizajniran ovako:

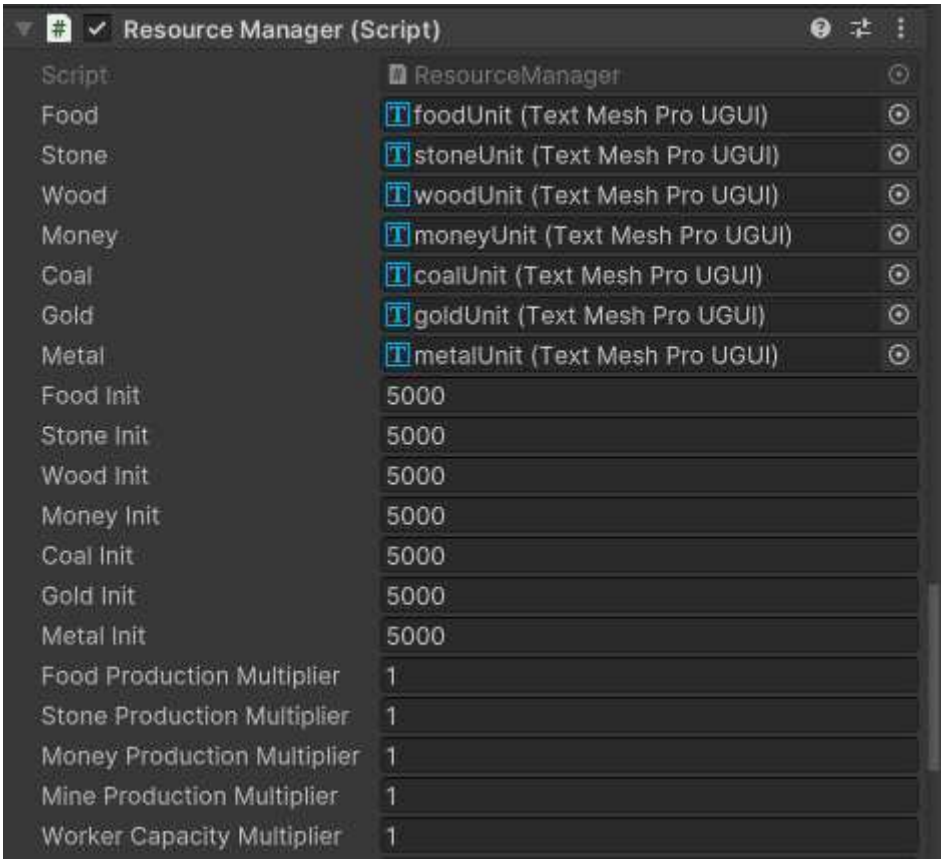


Slika 3. Prikaz sučelja pri pokretanju levela

U gornjem djelu se nalaze resursi koje će igrač koristiti za građenje zgrada, treniranje jedinica i istraživanje vještina. Ispod resursa u desnom dijelu se nalazi pratitelj faze i populacije. Faza je igraču bitna, jer ovisno u kojoj je fazi, dobiva mogućnost gradnje novih zgrada. Populacija predstavlja maksimalan broj jedinica koje igrač može imati na raspolaganju. Sve ovo se može vidjeti na slici 3. U nastavku će se više pojasniti kako se prati stanje svih tih elemenata [8].

4.4. Resursi

Skripta ResourceManager je skripta s kojom se upravlja resursima, njihovom potrošnjom i dobivanjem. ResourceManager skriptu koristi igrač, dok neprijatelj ima EnemyResourceManager s istim funkcionalnostima. Postoji 7 vrsta resursa koje igrač može koristiti: drvo, kamen, hrana, novac, zlato, ugljen i metal. Tih 7 resursa je prikazano na slici 4.



Slika 4. Prikaz resursa i njihovih početnih stanja

Svaki od tih resursa ima svoj način dobivanja, kao i određeni multiplikator, koji povećava proizvodnju resursa. Za proizvodnju drveta je potrebno poslati radnike na njega, koji će kad dođu do drveta krenuti s poslom, te dovoditi drvo u bazu. Ostali resursi se dobivaju preko zgrada, hrana se dobiva iz farme, metal, zlato i ugljen se dobiva iz rudnika, kamen se dobiva iz kamenoloma dok se novac dobiva iz upravne zgrade.

```
public void AddFood(int amount, bool production)
{
    if (production)
        foodCurrent += Mathf.RoundToInt(amount * foodProductionMultiplier);
    else
        foodCurrent += amount;

}
```

Programski kod 3. ResourceManager

Isječak programskog koda prikazuje primjer dodavanja resursa. Metoda AddFood() ima 2 parametra, količinu i boolean production, koji predstavlja dali resurs koji se dodaje dolazi iz zgrade za proizvodnju (jer se resurs može dobiti na još jedan način koji će biti objašnjen kasnije).

4.4.1. Proizvodnja resursa preko zgrade

U nastavku će se prikazati proizvodnja resursa unutar svake zgrade koja ima namjenu proizvoditi resurse:

```
private void Start()
{
    if (isAIControlled)
    {
        enemyResourceManager = EnemyResourceManager.Instance;
    }
    else
    {
        resourceManager = FindObjectOfType<ResourceManager>();
    }

    buildingProgress = GetComponent<BuildingProgress>();

    if (buildingProgress == null || buildingProgress.IsCompleted())
    {
        StartProducing();
    }
    else
    {
        buildingProgress.OnBuildingComplete += StartProducing;
    }
}

2 references
public void StartProducing()
{
    if (!isProducing && (buildingProgress == null || buildingProgress.IsCompleted()))
    {
        isProducing = true;
        StartCoroutine(ProduceResources());
    }
}
```

Programski kod 4. ResourceProducer-1. dio

Isječak gornjeg koda prikazuje početak proizvodnje resursa skripte ResourceProducer koja je univerzalna i za igrača i neprijatelja. Na početku se gleda dali je zgrada na koju je ova skripta vezana pripada igraču ili neprijatelju. Slijedi dohvat skripte buildingProgress koja prati napredak izgradnje zgrade te se nakon toga gleda status zgrade (boolean IsCompleted koji predstavlja dali je zgrada završila s izgradnjom ili ne). Ako je zgrada završena poziva se metoda StartProducing() koja stavlja vrijednost booleana isProducing na true i pokreće korutinu ProduceResources().

```
private IEnumerator ProduceResources()
{
    while (true)
    {
        yield return new WaitForSeconds(productionInterval);

        if (isAIControlled && enemyResourceManager != null)
        {
            ProduceForEnemy();
        }
        else if (resourceManager != null)
        {
            ProduceForPlayer();
        }
    }
}
```

Programski kod 5. ResourceProducer-2. dio

Programski kod 5 opisuje korutinu ProduceResources() koja svakih nekoliko sekundi koje su predstavljene parametrom productionInterval proizvodi resurs vezan uz tu zgradu. Ovisno o tome dali je neprijatelj ili igrač resursi se proizvode za igrača ili neprijatelja.

```
public enum ResourceType { Food, Stone, Gold, Metal, Coal, Money }

public ResourceType resourceProduced;
```

Programski kod 6. ResourceProducer-3. dio

Ovisno o enumeraciji postavljenoj za zgradu, proizvodit će se taj tip resursa, što je prikazano i kodom ispod.

```
private void ProduceForPlayer()
{
    switch (resourceProduced)
    {
        case ResourceType.Food:
            resourceManager.AddFood(productionAmount, true);
            break;
        case ResourceType.Stone:
            resourceManager.AddStone(productionAmount, true);
            break;
        case ResourceType.Gold:
            resourceManager.AddGold(productionAmount, true);
            break;
        case ResourceType.Metal:
            resourceManager.AddMetal(productionAmount, true);
            break;
        case ResourceType.Coal:
            resourceManager.AddCoal(productionAmount, true);
            break;
        case ResourceType.Money:
            resourceManager.AddMoney(productionAmount, true);
            break;
    }
}
```

Programski kod 7. ResourceProduction-4.dio

4.4.2. Dobivanje resursa preko radnika

Osim proizvodnje resursa preko zgrada, kao što je prije navedeno resursi se mogu dobivati i preko radnika.

Na početku igre, postoji nasumično stvaranje stabala na mapi, koje će služiti za dobivanje drveta. Za to je zaslužna skripta TreeGenerator, koja će proizvesti određeni broj stabala na mapi pod određenim uvjetima koji će biti u nastavku objašnjeni.

```
void GenerateTrees()
{
    for (int i = 0; i < numberOfTrees; i++)
    {
        Vector3 position = FindSuitablePosition();
        if (position != Vector3.zero)
        {
            GameObject tree = Instantiate(treePrefabs[Random.Range(0, treePrefabs.Length)], position, Quaternion.identity);
            tree.layer = LayerMask.NameToLayer("Tree");

            Collider treeCollider = tree.GetComponent<Collider>();

            NavMeshObstacle navObstacle = tree.GetComponent<NavMeshObstacle>();
            if (navObstacle == null)
            {
                navObstacle = tree.AddComponent<NavMeshObstacle>();
                navObstacle.carving = true;

                if (treeCollider is CapsuleCollider capsuleCollider)
                {
                    navObstacle.shape = NavMeshObstacleShape.Capsule;
                    navObstacle.size = new Vector3(capsuleCollider.radius * 2, capsuleCollider.height, capsuleCollider.radius * 2);
                    navObstacle.center = capsuleCollider.center;
                }
            }
        }
    }
}


1 reference
Vector3 FindSuitablePosition()
{
    for (int attempts = 0; attempts < 100; attempts++)
    {
        Vector3 position = new Vector3(
            Random.Range(0, terrainSize.x),
            0,
            Random.Range(0, terrainSize.z)
        );
        position.y = terrain.SampleHeight(position) + terrain.GetPosition().y;

        if (IsSuitableForTree(position))
        {
            return position;
        }
    }
    return Vector3.zero;
}
```

Programski kod 8. TreeGenerator-1.dio

Kroz for petlju iteriramo svako stablo te pronalazimo poziciju na terenu za postavljanje stabla. Metoda FindSuitablePosition traži ispravnu poziciju za postavljanje stabla. Ispravna pozicija se traži 100 puta, te ako se ispune uvjeti koji se nalaze u metodi IsSuitablePosition(), metoda će vratiti poziciju. Ako je pozicija dobra, odabire se jedan od nekoliko predložaka stabala koji će se koristiti, te ako nema će mu se postaviti sloj „Tree“, koji će služiti za prepoznavanje da se radi o stablu. Nakon toga se još dodatno postavi NavMeshObstacle komponenta da jedinice ne prolaze kroz drvo te se napravi carving da se stvori rupa u NavMeshu, i na kraju se postavi NavMeshObstacle kao kapsula (pod uvjetom da stablo ima CapsuleCollider).

```
bool IsSuitableForTree(Vector3 position)
{
    float height = terrain.SampleHeight(position) + terrain.GetPosition().y;
    float slope = terrain.terrainData.GetSteepness(position.x / terrainSize.x, position.z / terrainSize.z);

    if (slope > maxSlope || height < waterLevel + 0.2f || height > maxLevel)
        return false;

    return true;
}
```

Programski kod 9. TreeGenerator-2.dio

Uvjeti za ispravnu poziciju su visina i nagib terena. Postavljeno je da se stabla smiju nalaziti samo na određenoj visini između razine vode (waterLevel) i najviše dopuštene visine terena (maxLevel). Drugi uvjet je da teren nije previše strm (oko 30 stupnjeva je maksimalno dopuštenje), jer ako je teren previše strm boolean IsSuitableForTree će vratiti false, dajući znanja da ta pozicija ne terenu nije ispravna.

```
public class TreeResource : MonoBehaviour
{
    public int woodAmount = 100;
    public int woodPerChop = 1;

    1 reference
    public void Chop()
    {
        woodAmount -= woodPerChop;
        if (woodAmount <= 0)
        {
            Destroy(gameObject);
        }
    }
}
```

Programski kod 10. TreeResource

Kao što se vidi i u programskom kodu 10, svakom stablu se također dodjeljuje skripta TreeResource, u kojoj se postavlja koliko drveta ima svako stablo, te koliko ga gubi pri „zamahu sjekire“ od radnika. Metoda Chop() je zaslužna za oduzimanje jedinica drva od stabla.

```
public void AssignCommandPost(Transform newCommandPost)
{
    if (commandPost == null)
    {
        commandPost = newCommandPost;
    }
}
```

Programski kod 11. ResourceGatherer-1. dio

Sada će se opisati proces dobivanja drveta: prije nego što radnik krene po resurse, mora mu se dodijeliti upravljačka zgrada u koju će se vraćati da dostavi resurse, što je prikazano u programskom kodu 11. Zgrada mu se dodjeljuje nakon obavljenog treniranja. Postoje 4 radnje koje se odvijaju za dohvat drveta:

```
public void GoToTree(TreeResource tree)
{
    if (navMeshAgent == null)
        return;

    targetTree = tree;
    currentState = State.MovingToTree;
    navMeshAgent.SetDestination(targetTree.transform.position);
}

1 reference
private void MoveToTree()
{
    if (targetTree == null)
    {
        currentState = State.Idle;
        return;
    }

    float distanceToTree = Vector3.Distance(transform.position, targetTree.transform.position);
    if (distanceToTree <= chopRange)
    {
        currentState = State.Harvesting;
    }
}
```

Programski kod 12. ResourceGatherer-2. dio

U prvoj metodi GoToTree() radnik odlazi do stabla, ovisno o odlučenom stablu od igrača, te dobiva stanje MovingToTree.

Kada dobije stanje MovingToTree, izvršava se metoda MoveToTree, u kojoj se izračunava udaljenost između radnika i stabla, te se radnika šalje do stabla. Svaki radnik ima prikačenu varijablu chopRange, koji predstavlja maksimalnu udaljenost od koje radnik može krenuti sa „sječom stabla“. Kada je došao u chopRange, prelazi u stanje Harvesting, što se može vidjeti u kodu 12.

```
private void HarvestTree()
{
    if (isEnemy)
        unitStats.capacity = unitStats.baseCapacity;
    else
        unitStats.capacity = Mathf.RoundToInt(unitStats.baseCapacity * resourceManager.workerCapacityMultiplier);

    workTimer += Time.deltaTime;
    if (workTimer >= chopTime)
    {
        workTimer = 0.0f;
        targetTree.Chop();
        currentWood += targetTree.woodPerChop;

        if (currentWood >= unitStats.capacity)
        {
            navMeshAgent.SetDestination(commandPost.position);
            currentState = State.Returning;
        }

        if (targetTree.woodAmount <= 0)
        {
            targetTree = null;
            currentState = State.Idle;
        }
    }
}
```

Programski kod 13. ResourceGatherer-3. dio

Kada pređe u stanje Harvesting, radniku se postavi kapacitet (koliko jedinica drveta može prenijeti) ovisno o početnom kapacitetu i mogućim istraživanjima. Kapacitet za neprijatelja je

postavljen na početni. Varijabla workTimer se povećava s vremenom te kada se ujednači s parametrom chopTime (koji predstavlja interval između izvođenja „sječenja drveta“), radnik dodaje u kapacitet količinu drveta, određena parametrom woodPerChop. „Sječa“ stabla se može vidjeti na slici 5. Ako je stablo ostalo bez drveta za uzeti, radnik se prebacuje u stanje Idle, te mu se briše referenca. Ako je radnik popunio svoj kapacitet, prelazi u stanje Returning.

```
private void ReturnToCommandPost()
{
    if (commandPost == null)
    {
        currentState = State.Idle;
        return;
    }

    float distanceToCommandPost = Vector3.Distance(transform.position, commandPost.position);
    if (distanceToCommandPost <= commandPostRange)
    {
        if (isEnemy)
        {
            enemyResourceManager?.AddWood(currentWood, true);
        }
        else
        {
            resourceManager?.AddWood(currentWood, false);
        }

        currentWood = 0;
        currentState = State.MovingToTree;

        if (targetTree != null)
        {
            navMeshAgent.SetDestination(targetTree.transform.position);
        }
    }
}
```

Programski kod 14. ResourceGatherer-4. dio

Stanje Returning predstavlja vraćanje u bazu tj. postavljenoj upravljačkoj zgradi. Ispražnjuje se njegov kapacitet te ovisno dali je neprijatelj ili igrač, dodat će resurs odgovarajućem menadžmentu resursa. Kapacitet radnika se postavlja na 0 te se stanje radnika postavlja na MovingToTree, čime se započinje proces od početka, kao što je prikazano u kodu 14.

Slika 5. Prikaz sječe stabla

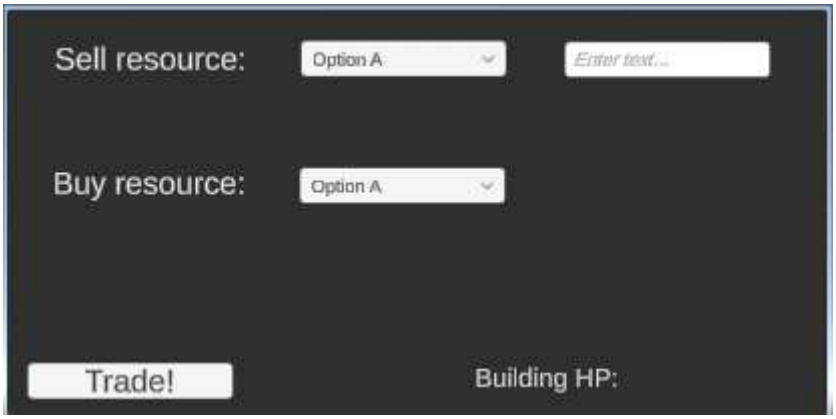
4.4.3. Dobivanje resursa prodajom

Osim skupljanja resursa, određeni resursi mogu se dobiti i prodajom. To se može napraviti u zgradi Market, gdje igrač može za određenu količinu resursa kupiti drugi resurs, po prije definiranim omjerima zamjene.

```
private void InitializeExchangeRates()
{
    exchangeRates = new Dictionary<string, Dictionary<string, float>>()
{
    { "Wood", new Dictionary<string, float>
        {
            { "Stone", 0.5f },  // 1 Wood -> 0.5 Stone
            { "Gold", 0.2f },   // 1 Wood -> 0.2 Gold
            { "Coal", 0.4f },   // 1 Wood -> 0.4 Coal
            { "Metal", 0.4f },  // 1 Wood -> 0.4 Metal
            { "Money", 0.5f },  // 1 Wood -> 0.5 Money
            { "Food", 0.8f }    // 1 Wood -> 0.8 Food
        }
```

Programski kod 15. MarketUI-1. dio

Metoda InitializeExchangeRates() sadrži omjer zamjene resursa za svaki resurs. Programski kod gore prikazuje omjere zamjene drveta za sve ostale resurse.



Slika 6. Prikaz sučelja za razmjenu resursa

U sučelju se nalaze dva dropdowna, jedan gdje se bira resurs koji se prodaje a ispod njega resurs koji se želi kupiti. Pokraj resursa za prodaju se nalazi polje u koje se može unijeti broj resursa koji se želi prodati, a nakon odabranog resursa koji se želi kupiti, ispod polja za unos se dobije broj resursa koji će se dobiti prilikom razmjene, kao što se može vidjeti na slici 6.

```
private bool PerformTrade(string sellResource, string buyResource, int sellAmount, int buyAmount)
{
    if (HasEnoughResources(sellResource, sellAmount))
    {
        DeductResource(sellResource, sellAmount);
        AddResource(buyResource, buyAmount);
    }
    return false;
}
```

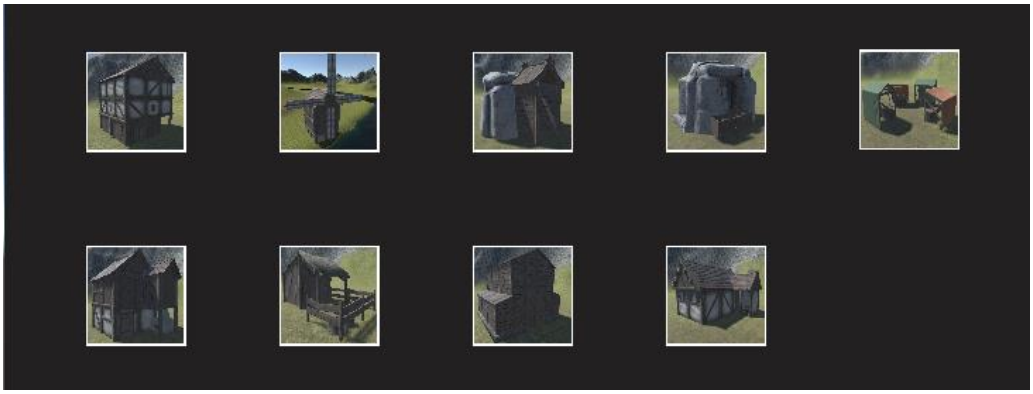
Programski kod 16. MarketUI-2. dio

Prilikom klika gumba Trade, izvršit će se metoda PerformTrade koja će gledati dali igrač ima dovoljno resursa za oduzeti da se izvrši razmjena, ako ima igraču će se oduzeti resursi i dodati oni koji je tražio.

4.5. Gradnja zgrada

Gradnja zgrada je sljedeći element igre koji će biti objašnjen. Objasnit će se proces postavljanja zgrade, odlučivanja dali je mjesto gdje igrač želi postaviti zgradu prikladno, kako se zgrade izgrađuju te što one mogu jednom kad su izgrađene.

Igrač kreće s postavljanjem zgrade tako da pritisne tipku B, te se tima otvara prozor u donjem desnom dijelu gdje igrač ima na raspolaganju 9 zgrada koje može izgraditi: upravljačka zgrada, farma, kamenolom, rudnik, vojarna, konjica, škola, market i artiljerija. Postoji i jedna posebna jedinica koja se prikazuje u ovom sučelju, ali tek kada se otključa istraživanje, most. Sve ove zgrade su prikazane u slici 7. Za zgrade su korišteni besplatni modeli dostupni u Unity Asset Storeu [7][11].



Slika 7. Prikaz sučelja za odabir zgrade za izgradnju

Upravljačka zgrada služi kao odredišna točka radnika gdje će dostavljati drva, te služi za treniranje radnika i proizvodnju novca.

Farma služi za proizvodnju hrane, kamenolom služi za proizvodnju kamena dok rudnik služi za proizvodnju ruda, specifično metala, zlata i ugljena.

Vojarna, konjica i artiljerija služe za proizvodnju jedinica za borbu.

Market služi razmjeni resursa, škola služi za istraživanje vještina.

4.5.1. Priprema predloška zgrade

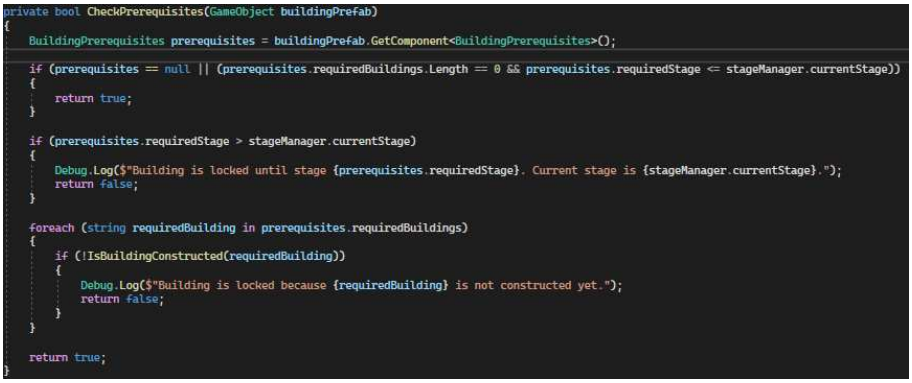
Prva stvar koja se mora napraviti prilikom izgrade zgrade je pripremiti njen predložak koji će se na kraju instancirati kao nova zgrada.

```
public void StartPlacingBuilding(GameObject buildingPrefab, Quaternion rotation, float raiseAmount)
{
    buildingPrefabToPlace = buildingPrefab;
    buildingRotation = rotation;
    this.raiseAmount = raiseAmount;

    if (buildingPrefabToPlace.name.Contains("Bridge"))
    {
        isBridge = true;
        bridgeLength = 2f;

        buildingRotation = Quaternion.Euler(0, 0, 0);
    }
    else
    {
        isBridge = false;
    }

    if (CheckPrerequisites(buildingPrefab))
    {
        IsPlacingBuilding = true;

        if (currentBuildingPreview != null)
        {
            Destroy(currentBuildingPreview);
        }

        currentBuildingPreview = Instantiate(buildingPrefabToPlace);
        currentBuildingPreview.GetComponent<Collider>().enabled = false;
        SetBuildingPreviewOpacity(0.5f);
    }
    else
    {
        DisplayMessage("Prerequisites not met for this building.");
    }
}
```

Programski kod 17. BuildingPlacement-1.dio

Prilikom klika na bilo koju od ikona u sučelju za izgradnju zgrada, pokreće se metoda StartPlacingBuilding(), prikazana na programskom kodu 17. Ona za parametre prima predložak zgrade koja se planira izgraditi, njezina rotacija i određena visina za podizanje od terena (da zgrada ne izgleda da je u terenu).

Dodijeljuje vrijednosti primljenih parametra lokalnim parametrima buildingPrefabToPlace, buildingRotation i this.raiseAmount. Ako se radi o mostu, boolean isBridge će biti true, te će se postaviti određena dužina mosta. Sljedeće što se radi je da se gleda dali su svi uvjeti za izgradnju zgrade zadovoljeni, za to je zaslužna metoda CheckPrerequisites():

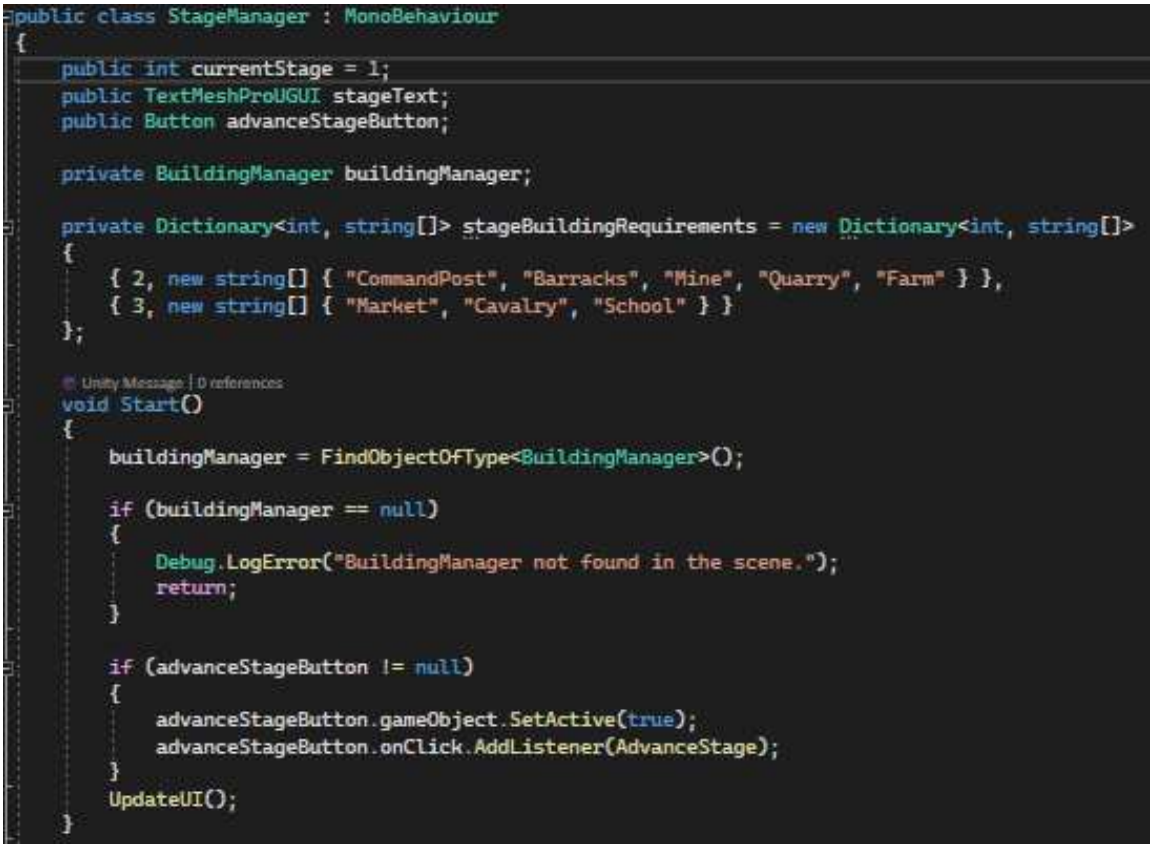


```
private bool CheckPrerequisites(GameObject buildingPrefab)
{
    BuildingPrerequisites prerequisites = buildingPrefab.GetComponent<BuildingPrerequisites>();

    if (prerequisites == null || (prerequisites.requiredBuildings.Length == 0 && prerequisites.requiredStage <= stageManager.currentStage))
    {
        return true;
    }

    if (prerequisites.requiredStage > stageManager.currentStage)
    {
        Debug.Log($"Building is locked until stage {prerequisites.requiredStage}. Current stage is {stageManager.currentStage}.");
        return false;
    }

    foreach (string requiredBuilding in prerequisites.requiredBuildings)
    {
        if (!IsBuildingConstructed(requiredBuilding))
        {
            Debug.Log($"Building is locked because {requiredBuilding} is not constructed yet.");
            return false;
        }
    }

    return true;
}
```

Programski kod 18. BuildingPlacement-2.dio

Metoda gleda dali su svi uvjeti ispunjeni tako da uzme skriptu BuildingPrerequisites koja je univerzalna i povezana sa svakom zgradom, a sadrži popis svih zgrada koje moraju biti izgrađene prije nego što trenutna zgrada koju igrač planira može sagraditi. Tu pomaže metoda i metoda IsBuildingConstructed koja se vrti unutar petlje, te gleda dali se instanca zgrade koja se nalazi u nizu requiredBuildings nalazi na mapi. Provjera također uključuje provjeru faze u kojoj igrač mora biti prije nego što može sagraditi tu zgradu.

Ako je sve u redu postavlja se vrijednost booleana isPlacingBuilding na true, te se instancira predložak zgrade koja se planira izgraditi s povećanom prozirnošću.



```
public class StageManager : MonoBehaviour
{
    public int currentStage = 1;
    public TextMeshProUGUI stageText;
    public Button advanceStageButton;

    private BuildingManager buildingManager;

    private Dictionary<int, string[]> stageBuildingRequirements = new Dictionary<int, string[]>
    {
        { 2, new string[] { "CommandPost", "Barracks", "Mine", "Quarry", "Farm" } },
        { 3, new string[] { "Market", "Cavalry", "School" } }
    };

    Unity Message | 0 references
    void Start()
    {
        buildingManager = FindObjectOfType<BuildingManager>();

        if (buildingManager == null)
        {
            Debug.LogError("BuildingManager not found in the scene.");
            return;
        }

        if (advanceStageButton != null)
        {
            advanceStageButton.gameObject.SetActive(true);
            advanceStageButton.onClick.AddListener(AdvanceStage);
        }
        UpdateUI();
    }
```

Programski kod 19.StageManager

Kako je već spomenuta faza, skripta StageManager prikazana u programskom kodu 19 je zaslužna za praćenje faze u kojoj je igrač trenutno. Skripta ima varijable currentStage koja predstavlja trenutnu fazu, varijabla stageText koja će se prikazati na ekranu te gumb s kojim igrač može prijeći u novu fazu. Svaka faza je zaključana iza par zgrada koje igrač prvo mora napraviti da bi mogao prijeći u iduću fazu.

```
public bool CanAdvanceToNextStage()
{
    if (currentStage >= 3)
    {
        return false;
    }

    if (stageBuildingRequirements.ContainsKey(currentStage + 1))
    {
        string[] requiredBuildings = stageBuildingRequirements[currentStage + 1];

        foreach (string buildingName in requiredBuildings)
        {
            if (buildingManager.GetBuildingCount(buildingName) == 0)
            {
                return false;
            }
        }
    }

    return true;
}
```

Programski kod 20. StageManager-2. dio

Provjeru dali igrač može prijeći u novu fazu ili ne obavlja metoda CanAdvanceToNextStage(). Kao što se može vidjeti u programskom kodu 20, kada igrač u listi zgrada unutar BuildingManagera ima sve zgrade koje su potrebne za prelazak u novu fazu, omogućuje mu se gumb s kojim prelazi u iduću fazu, te otključava nove zgrade. Potom se gumb opet sakrije te se moraju nove zgrade sagradit za prelazak na iduću fazu (osim ako igrač ne dođe do faze 3, to je zadnja faza). Pozicija gdje se nalazi stage je prikazan na slici 8.

Slika 8. Prikaz stage polja

```
private void Update()
{
    if (IsPlacingBuilding)
    {
        FollowCursor();

        if (Input.GetKey(KeyCode.R))
        {
            RotateBuilding(Time.deltaTime * 100f);
        }

        if (Input.GetKeyDown(KeyCode.R))
        {
            RotateBuilding(1f);
        }

        if (Input.GetMouseButtonDown(0) && currentBuildingPreview != null && currentBuildingPreview.activeSelf)
        {
            PlaceBuilding();
        }

        if (Input.GetKeyDown(KeyCode.Escape))
        {
            CancelBuildingPlacement();
        }
```

Programski kod 21. BuildingPlacement-3. dio

Ranije spomenuti boolean je bitan jer omogućuje sljedeći dio izgradnje, gdje se postavlja da predložak prati kursor igrača, da se zgrada može rotirati po želji igrača prije postavljanja te samog postavljanja zgrade na teren.

4.5.2. Provjera terena

Nakon što su zadovoljeni svi uvjeti te igrač ima mogućnost gradnje te zgrade, sljedeći korak u procesu izgradnje je provjera uvjeta terena, da igrač može izgraditi zgradu na određenoj poziciji.

```
private void FollowCursor()
{
    if (!IsPlacingBuilding || currentBuildingPreview == null) return;

    Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
    if (Physics.Raycast(ray, out RaycastHit hit, Mathf.Infinity, terrainLayer))
    {
        Vector3 position = SnapToGrid(hit.point);
        position.y += raiseAmount;

        if (isBridge)
        {
            if (position.y < 53f)
            {
                currentBuildingPreview.SetActive(false);
                return;
            }

            Vector3 bridgeStart = hit.point;
            currentBuildingPreview.transform.position = bridgeStart;

            currentBuildingPreview.SetActive(true);
        }
        else
        {
            if (IsValidPlacement(position))
            {
                currentBuildingPreview.transform.position = position;
                currentBuildingPreview.transform.rotation = buildingRotation;
                currentBuildingPreview.SetActive(true);
            }
            else
            {
                currentBuildingPreview.SetActive(false);
            }
        }
    }
}
```

Programski kod 22. BuildingPlacement-4. dio

Ako se u nekom trenutku odustane od izgradnje zgrade, FollowCursor() prestaje s radom. Koristi se Raycast da se dobije pozicija kursora te je namješten da specifično udara u layer terena što se vidi i u kodu 22 priloženom gore [6]. Dobivena pozicija se nakon toga podešava metodom SnapToGrid(), da se omogući postavljanje na mrežaste pozicije.

```
private bool IsValidPlacement(Vector3 position)
{
    if (!IsTerrainEven(position) || IsBuildingAlreadyPresent(position))
        return false;

    if (IsTreePresent(position))
        return false;

    return CheckResourceCost(buildingPrefabToPlace);
}
```

Programski kod 23. BuildingPlacement-5. dio

Nakon toga dolazi provjera IsValidPlacement(), koja unutar sebe ima još par metoda u svrhu provjere, od kojih će svaka biti objašnjena:

```
private bool IsTerrainEven(Vector3 position)
{
    float halfGridX = gridSize.x / 2;
    float halfGridZ = gridSize.z / 2;
    float maxHeightDifference = 0.1f;

    float centerHeight = Terrain.activeTerrain.SampleHeight(position);


    for (float xOffset = -halfGridX; xOffset <= halfGridX; xOffset += gridSize.x)
    {
        for (float zOffset = -halfGridZ; zOffset <= halfGridZ; zOffset += gridSize.z)
        {
            Vector3 samplePosition = position + new Vector3(xOffset, 0f, zOffset);
            float sampleHeight = Terrain.activeTerrain.SampleHeight(samplePosition);


            float heightDifference = Mathf.Abs(sampleHeight - centerHeight);
            if (heightDifference > maxHeightDifference)
            {
                return false;
            }
        }
    }

    return true;
}
```

Programski kod 24. BuildingPlacement-6. dio

Metoda IsTerrainEven prikazana u kodu 24 je prva provjera koja gleda dali je teren na određenoj poziciji ravan unutar nekog područja (veličina zgrade). Prima parametar (Vector3 koji predstavlja poziciju kursora). Varijable HalfGridX i halfGridZ predstavljaju polovicu veličine mreže gridSize, koja je jedna od varijabla skripte BuildingPlacement koji se postavlja unutar Inspectora. MaxHeightDifference predstavlja maksimalno dopuštenu razliku visine terena u tom području da bi se on smatrao ravnim. CenterHeight predstavlja visinu terena u središtu područja te se ona uzima kao referenca. Petljom se prolazi kroz sve točke unutar mreže, te se za svaku točku uzima njena visina uz pomoć Terrain.activeTerrain.SampleHeight(samplePosition). Ako se dogodi da bilo koja točka ima veće odstupanje u visini od maxHeightDifference, smatra se da teren nije ravan te nije prikladan za izgradnju zgrade na toj poziciji. Ako je sve u redu na red dolaze sljedeće provjera, IsBuildingAlreadyPresent() i IsTreePresent().

```
private bool IsBuildingAlreadyPresent(Vector3 position)
{
    Collider[] colliders = Physics.OverlapSphere(position, gridSize.x / 2);

    foreach (Collider collider in colliders)
    {
        BuildingInstance existingBuilding = collider.GetComponent<BuildingInstance>();
        if (existingBuilding != null)
        {
            return true;
        }
    }

    return false;
}

1 reference
private bool IsTreePresent(Vector3 position)
{
    float checkRadius = gridSize.x;

    Collider[] colliders = Physics.OverlapSphere(position, checkRadius);

    if (colliders.Length > 0)
    {
        foreach (var collider in colliders)
        {
            TreeResource treeResource = collider.GetComponent<TreeResource>();
            if (treeResource != null)
            {
                return true;
            }
        }
    }
    return false;
}
```

Programski kod 25. BuildingPlacement-7. dio

Metoda IsBuildingPlacement prikazana u kodu 25 služi da se provjeri dali na poziciji (koju prima kao parametar) nalazi već neka zgrada ili ne. Postavlja se varijabla colliders, koja će sadržavati sve kolidere unutar određene sfere oko pozicije. Veličina sfere je određena polovicom veličine mreže.

Kad se dobiju svi kolideri unutar sfere, gleda se dali dobiveni kolider sadrži komponentu BuildingInsance, koju ima svaka zgrada. Ako se dobije, smatra se da se zgrada nalazi na tom području te nije prikladna pozicija za izgradnju nove zgrade.

IsTreePresent funkcionira na sličan način, samo uzima veću veličinu radijusa, cijelu mrežu.

Isto gleda dali se nalazi neki kolider u sferi koji sadrži komponentu TreeResource koju ima svako drvo. Ako se pronađe drvo u toj sferi, područje nije dobro za izgradnju zgrade.

Ako su ova 3 uvjeta zadovoljena, ostaje još jedan uvjet za zadovoljiti, a to je dali igrač ima dovoljno resursa za izgradnju te zgrade.

```
private bool CheckResourceCost(GameObject buildingPrefab)
{
    BuildingCost buildingCost = buildingPrefab.GetComponent<BuildingCost>();
    if (buildingCost != null)
    {
        return resourceManager.HasEnoughResources(
        buildingCost.woodCost,
        buildingCost.stoneCost,
        buildingCost.goldCost,
        buildingCost.foodCost,
        buildingCost.moneyCost,
        buildingCost.coalCost,
        buildingCost.metalCost
    );
    }
    return true;
}
```

Programski kod 26. BuildingPlacement-8. dio

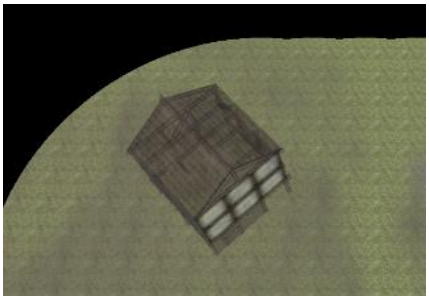
Provjera dali igrač ima dovoljno resursa se odvija na taj način da se prvo uzme skripta vezana uz zgradu, BuildingCost.

```
public class BuildingCost : MonoBehaviour
{
    public int woodCost;
    public int stoneCost;
    public int goldCost;
    public int foodCost;
    public int moneyCost;
    public int coalCost;
    public int metalCost;
    public string description;
    public int buildingLimit;
}
```

Programski kod 27. BuildingCost

BuildingCost sadrži informacije koliko svakog od resursa treba za izgradnju te zgrade, kratak opis čemu zgrada služi te limit koliko se takvih instanca zgrade može napraviti, što se vidi i u programskom kodu 27.

Nakon što se dobije informacija o tome koliko je resursa potrebno, ResourceManager gleda dali igrač ima dovoljno svakog resursa te ako ima, i taj uvjet je zadovoljen, te će CheckResourceCost prikazan u kodu 26 vratiti true.



Slika 9. Predložak zgrade kada su uvjeti zadovoljeni za izgradnju

Ako su svi ovi uvjeti zadovoljeni, igrač će moći vidjeti predložak zgrade na ekranu kao što se može vidjeti na slici 9. Ako u bilo kojem trenutku neki od uvjeta nije zadovoljen, predložak se neće pojaviti. Sljedeće što igrač treba napraviti je kliknuti lijevi klik miša, te postavlja zgradu na teren. Naravno, ako želi može rotirati zgradu po volji, što radi držanjem tipke R. Prikaz takve rotirane zgrade se nalazi na slici 10. Ako stisne Escape dok planira sagraditi zgradu, odustaje od izgradnje zgrade.

Slika 10. Rotirana zgrada

4.5.3. Izgradnja zgrade

Sljedeći korak na redu je sama izgradnje zgrade.

```
private void PlaceBuilding()
{
    Vector3 position = currentBuildingPreview.transform.position;
    Vector3 scale = currentBuildingPreview.transform.localScale;

    BuildingCost buildingCost = buildingPrefabToPlace.GetComponent<BuildingCost>();
    if (buildingCost != null && buildingManager.CanBuild(buildingCost.name, buildingCost.buildingLimit))
    {
        DeductResources(buildingPrefabToPlace);

        GameObject placedBuilding = Instantiate(buildingPrefabToPlace, position, buildingRotation);
        placedBuilding.transform.localScale = scale;

        if (isBridge)
        {
            NavMeshSurface navMeshSurface = placedBuilding.AddComponent<NavMeshSurface>();
            navMeshSurface.BuildNavMesh();
        }

            BuildingProgress buildingProgress = placedBuilding.GetComponent<BuildingProgress>();
        if (buildingProgress != null)
        {
            buildingProgress.StartBuilding();
            buildingProgress.AutoAssignBuilders();
        }

        NavMeshObstacle obstacle = placedBuilding.AddComponent<NavMeshObstacle>();

        Vector3 originalSize = obstacle.size;

        if(buildingPrefabToPlace.name.Contains("Farm"))
            obstacle.size = originalSize * 5;
        obstacle.carving = true;

        buildingManager.Build(buildingCost.name, placedBuilding.transform);
        if(!isBridge)
        SelectionManager.Instance.playerBuildings.Add(placedBuilding);

        Debug.Log($"Built {buildingCost.name}. Total: {buildingManager.GetBuildingCount(buildingCost.name)}");
    }
    else
    {
        Debug.Log($"Cannot build more {buildingCost.name}. Limit reached.");
    }

    Destroy(currentBuildingPreview);
    IsPlacingBuilding = false;
    buildingPrefabToPlace = null;
}
```

Programski kod 28. BuildingPlacement-9. dio

Nakon što je igrač kliknuo lijevi klik miša nakon što mu se pojavio predložak, metoda PlaceBuilding() uzima poziciju i veličinu, te ih sprema u varijable Vector3 position i scale. Također se gleda dali igrač nije prešao maksimalan broj izgrađenih zgrada tog tipa, za što služi BuildingManager.

```
public class BuildingManager : MonoBehaviour
{
    private Dictionary<string, int> buildingCounts = new Dictionary<string, int>();
    public List<Transform> commandPosts = new List<Transform>();
    public float globalBuildingHPMultiplier = 1f;

    private HashSet<string> buildingsConstructed = new HashSet<string>();
    public event Action<string> OnBuildingCompleted;

    1 reference
    public bool CanBuild(string buildingType, int limit)
    {
        if (!buildingCounts.ContainsKey(buildingType))
        {
            buildingCounts[buildingType] = 0;
        }
        return buildingCounts[buildingType] < limit;
    }
```

Programski kod 29. BuildingManager

BuildingManager prikazan u kodu 29 ima rječnik buildingsCounts koji prati broj izgrađenih zgrada za svaku od mogućih zgrada. String predstavlja naziv zgrade a int broj izgrađenih zgrada. Lista commandPostova služi za radnike, da ako nemaju pridružen upravljački centar, da im se pridruži jedan iz te liste. Varijabla globalBuildingHPMultiplier predstavlja multiplikator kojim se može povećati zdravlje zgrada, ovisno o istraživanjima. HashSet buildingsConstructed služi kasnije za stabla vještina, kao i event OnBuildingComplete, ali o njima će se reći kasnije.

Metoda CanBuild() kao parametre prima tip zgrade koji se gradi te limit koliko tih zgrada igrač može imati. Ako zgrada ne postoji u rječniku, taj tip se dodaje s početnom vrijednosti 0, što znači da taj tip zgrade još nije izgrađen. Vrijednost metode CanBuild je true ako tip zgrade i njen broj izgrađenih zgrada ne prelaze limit.

Vraćajući se na PlaceBuilding(), u sljedećem koraku se oduzimaju resursi od igrača potrebni za izgradnju zgrade te se instancira zgrada na poziciji predloška zgrade.

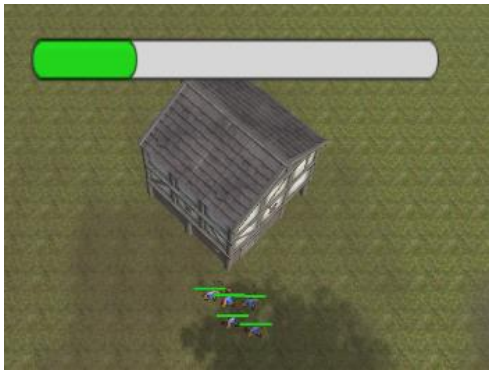
BuildingProgress je skripta zaslužna za napredak izgradnje, te se u njoj pokreće metoda StartBuilding().

```
public class BuildingProgress : MonoBehaviour
{
    public float totalBuildPoints;
    public float currentBuildPoints;
    public float baseBuildPoints;
    public float buildRadius;

    public GameObject progressBarPrefab;
    private ProgressBar progressBar;
    private RectTransform progressBarRectTransform;
    private bool isBuilding = false;
    private bool isCompleted = false;
    private bool isDestroyed = false;

    private BuildingManager buildingManager;

    public float buildSpeedMultiplier = 1f;
    public float buildingHPMultiplier = 1f;

    public bool isAIControlled = false;

    public event Action OnBuildingComplete;
    public event Action OnBuildingDestroyed;
```

Programski kod 30. BuildingProgress-1. dio

Klasa prikazana u kodu 30 sadržava nekoliko varijabli nužne za praćenje napretka izgradnje zgrada. TotalBuildingPoints, currentBuildPoints i baseBuildPoints predstavljaju zdravlje zgrade. CurrentBuildPoints predstavlja trenutno zdravlje zgrade, te se na početku postavi na 1. Kada vrijednost varijable currentBuildPoints postane jednaka vrijednosti totalBuildPoints, tada se smatra da je zgrada gotova s gradnjom. BaseBuildPoints predstavlja početnu vrijednost zdravlja zgrade, koja se može povećavati s multiplikatorima, te se ta vrijednost zapisuje u totalBuildPoints. Buildradius predstavlja radijus unutar kojeg će zgrada tražiti slobodne radnike.

Za bolju vizualnu projekciju stanja gradnje zgrade, koristi se progressBarPrefab, koji predstavlja trenutnu dovršenost izgradnje zgrade, prikazano na slici 11.



Slika 11. ProgressBar kod zgrade

Kako radnici daju zdravlje zgradi, tako zelena boja popunjava progress bar. Nakon što currentBuildPoints i totalBuildPoints postanu isti, progress bar se uklanja.

U skripti se nalaze i varijable koje prate stanje zgrade, isBuilding, isCompleted i isDestroyed kao i multiplikatori za brzinu izgradnje zgrade i zdravlja zgrade.

```
public void StartBuilding()
{
    if (isAIControlled)
    {
        return;
    }

    if (!isCompleted)
    {
        isBuilding = true;
        progressBar.gameObject.SetActive(true);
    }
}
```

Programski kod 31. BuildingProgress-2. dio

U metodi StartBuilding(), prvo se gleda dali je zgrada od igrača ili neprijatelja. Ako je od igrača, također se i gleda dali je završena ili ne te ako nije, progress bar ostaje aktivan.

```
public void AutoAssignBuilders()
{
    ResourceGatherer[] peasants = FindObjectsOfType<ResourceGatherer>();

    foreach (ResourceGatherer peasant in peasants)
    {
        float distanceToBuilding = Vector3.Distance(peasant.transform.position, transform.position);

        if (distanceToBuilding <= buildRadius && peasant.IsIdle())
        {
            peasant.GoToBuilding(this);
        }
    }
}
```

Programski kod 32. BuildingProgress-3. dio

U kodu 32 je prikaznaa metoda AutoAssignBuilders() koja dodjeljuje radnike koji trenutno ne rade da dođu do te zgrade. Prvo se računa udaljenost od zgrade u varijabli distanceToBuilding. Ako je radnik unutar buildRadiusa i ako je Idle (što znači da ne radi) onda će otići do zgrade gdje će krenuti „raditi na zgradi“:

Zadnji korak prilikom izgradnje zgrade je dodavanje NavMeshObstacle, da igra zna da radnici i ostale jedinice ne mogu prolaziti kroz zgradu nego da ju moraju zaobilaziti. Obstacle.carving opet stvara rupu na NavMeshu.

Zgrada je evidentirana u BuildingManageru, tako da se zna da je ona izgrađena te se dodaje u listu igračevih zgrada putem SelectionManager.Instance.playerBuildings.

Na kraju se uništava predložak zgrade, te se postavlja vrijednost booleana IsPlacingBuilding na false, davajući do znanja da igrač više nije u procesu izgradnje zgrade.

```
private void UpdateBuildingHpDisplay()
{
    BuildingProgress buildingProgress = currentBuilding.GetComponent<BuildingProgress>();

    if (buildingProgress != null)
    {
        buildingHpText.text = $"Building HP: {buildingProgress.currentBuildPoints}/{buildingProgress.totalBuildPoints}";
    }
    else
    {
        buildingHpText.text = "Building HP: N/A";
    }
}
```

Programski kod 33. SchoolUI-1. dio

Klikom na bilo koju zgradu, u ovom primjeru klikom na školu, osim prikaza jedinica za treniranje, vještine koje se mogu naučiti (ovisno o zgradi), svaka zgrada ima prikaz njenog zdravlja u donjem desnom dijelu. Napravljeno je da se UpdateBuildingHpDisplay prikazana u kodu 33 poziva u Update() dijelu skripte, te se prikaz zdravlja konstantno ažurira. Sama skripta dohvaća BuildingProgress komponentu u kojoj se nalazi zdravlje (currentBuildingPoints), te se sprema u varijablu buildingHpText koja je na početku inicijalizirana. Zdravlje zgrade se prikazuje u ovom obliku, trenutno zdravlje / ukupno zdravlje.

4.5.4. Izgradnja mosta

Most je posebna vrsta zgrade jer se razliku od ostalih mora otključati istraživanjem. Igrač je limitiran na 1 most u igri, ali mu može dati taktičku prednost jer mu može omogućiti da pokrene napad iz novog neočekivanog smjera.

Proces gradnje mosta funkcionira na isti način kao gradnja ostalih zgrada, ali postoje određene razlika.

```
private bool isBridge = false;
private float bridgeLength = 2f;
public float minBridgeLength = 1f;
public float maxBridgeLength = 4f;
public float bridgeLengthStep = 0.5f;
```

Programski kod 34. BuildingPlacement-10. dio

Skripta BuildingPlacement prikazana u kodu 34 sadrži i ove varijable, koje su specifične za most. Boolean isBridge se koristi da se označi dali je zgrada most ili ne. BridgeLength, maxBridgeLength i bridgeLengthStep označavaju početnu veličinu mosta, minimalnu i maksimalnu te veličina za koju igrač može smanjiti ili povećati duljina mosta.

```
if (isBridge)
{
    if (Input.GetKeyDown(KeyCode.Q))
    {
        AdjustBridgeLength(-bridgeLengthStep);
    }
    else if (Input.GetKeyDown(KeyCode.E))
    {
        AdjustBridgeLength(bridgeLengthStep);
    }
}
```

Programski kod 35. BuildingPlacement-11. dio

Tipkama Q i E igrač može povećati ili smanjiti veličinu mosta, što je korisno ako je npr. početna vrijednost dužina premala za prelazak preko rijeke.

```
private void AdjustBridgeLength(float adjustment)
{
    bridgeLength = Mathf.Clamp(bridgeLength + adjustment, minBridgeLength, maxBridgeLength);

    if (currentBuildingPreview != null)
    {
        Vector3 currentScale = currentBuildingPreview.transform.localScale;

        currentBuildingPreview.transform.localScale = new Vector3(currentScale.x, currentScale.y, bridgeLength);
    }
}
```

Programski kod 36. BuildingPlacement-12. dio

Metoda AdjustBridgeLength, koja uzima kao parametar ranije spomenuti bridgeLengthStep, prilagođava duljinu mosta. Ovdje se trenutna duljina mosta bridgeLength prilagođava za vrijednost parametra adjustment. Mathf.Clamp() osigurava da se duljina mosta drži unutar ograničenja postavljenih s minBridgeLength i maxBridgeLength. Duljina mosta ne može biti manja ili veća od tih granica. Vector3 currentScale dohvaća trenutnu veličinu mosta, te se nakon toga veličina predloška mosta postavlja prema vrijednosti bridgeLength, što se može vidjeti u kodu 36.

Pri završetku izgradnje most se ne dodaje u igračeve zgrade, niti se može obrisati iz razloga jer ne predstavlja bitnu zgradu (nije zgrada koja proizvodi resurse ili vojnike) te se želi potaknuti igrača da dobro razmisli gdje će postaviti most.

4.6. Treniranje jedinica

Sljedeći element, treniranje jedinica je bitan element strategijskih igara u stvarnom vremenu. U ovoj fazi, igrači koriste resurse i vojne zgrade za stvaranje jedinica koje će koristiti u borbi, prikupljanju resursa ili obrani. Proces treniranja uključuje odabir jedinica, određivanje vremena treniranja, te primjenu unapređenja kako bi jedinice bile efikasnije u igri.

Postoje 3 zgrade koje mogu trenirati jedinice a to su: vojarna, konjica i artiljerija.

Vojarna može trenirati lako i teško oklopljene vojnike te strijelce. Konjica može trenirati oklopljene vojnike na konju. Artiljerija trenira katapulte. Sve nabrojane jedinice su preuzete iz Unity Asset Storea [12].

```
public class UnitStats : MonoBehaviour
{
    public string unitName;
    public int hp;
    public int baseHp;
    public int attack;
    public int baseAttack;
    public int defense;
    public int baseDefense;
    public int capacity;
    public int baseCapacity;
    public float buildSpeed;
    public float baseBuildSpeed;
    public float productionSpeed;
    public float baseProductionSpeed;
    public float attackRange;
    public float baseAttackRange;
    public float attackSpeed;
    public float movementSpeed;
    public float baseMovementSpeed;
    public float sightRange;
    public float baseSightRange;

    public int foodCost;
    public int stoneCost;
    public int woodCost;
    public int moneyCost;
    public int coalCost;
    public int goldCost;
    public int metalCost;

    public bool isEnemy;

    private UnitManager unitManager;
    private EnemyAttackManager enemyAttackManager;

    5 references
    public bool IsAlive => hp > 0;
```

Programski kod 37. UnitStats

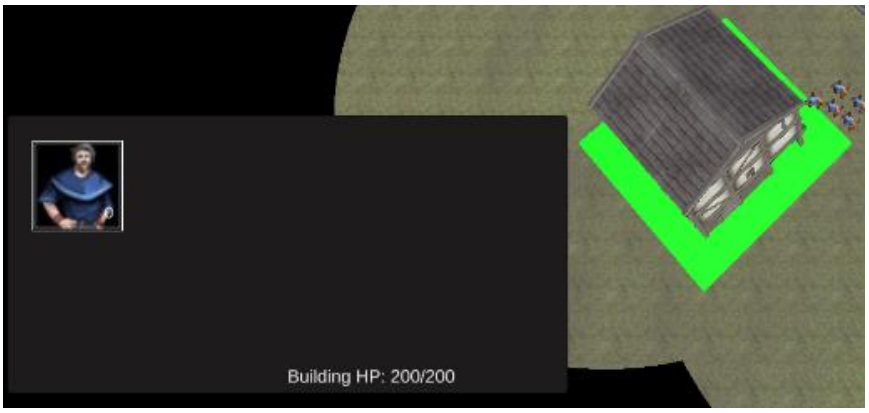
Svaka jedinica ima svoje statistike poput zdravlja, napada, obrane, brzine kretnje, brzine napada i troškova treniranja, što je vidljivo na slici 12 i kodu 37.

Slika 12. Prikaz zdravlja, napada i troškova treniranja jedinice

Nadalje će biti pobliže pojašnjen proces treniranja jedinica, koji osim njihovog treniranja uključuje i odabir pozicije na kojoj će se stvoriti.

4.6.1. Proces odabira jedinice za treniranje

Igrač započinje proces odabira jedinice klikom na bilo koju od zgrada koja ima mogućnost treniranja trupa. Nakon što klikne na zgradu koja ima tu mogućnost, dobije izgled prikazan na slici 13:



Slika 13. Izgled sučelja kod treniranja jedinica

```
public class UnitProductionCanvasController : MonoBehaviour
{
    public GameObject unitProductionPanel;
    public Transform unitButtonContainer;
    public GameObject unitButtonPrefab;

    public TextMeshProUGUI buildingHpText;

    private GameObject currentBuilding;
    private Transform spawnPoint;

    private Dictionary<UnitProduction, int> unitQueue = new Dictionary<UnitProduction, int>();
    private Dictionary<UnitProduction, TextMeshProUGUI> unitCountTexts = new Dictionary<UnitProduction, TextMeshProUGUI>();
    private float currentProductionTime = 0f;
    private UnitProduction currentUnitProduction = null;

    private Dictionary<UnitProduction, int> remainingUnitsToProduce = new Dictionary<UnitProduction, int>();
    public ResourceManager resourceManager;
    public UnitManager unitManager;
    public Popup popup;

    public GameObject selectionIndicatorPrefab;
    private GameObject currentIndicator;

    Unity Message | 0 references
    private void Start()
    {
        resourceManager = FindObjectOfType<ResourceManager>();
        popup = FindObjectOfType<Popup>();
        unitManager = FindObjectOfType<UnitManager>();
    }
```

Programski kod 38. UnitProductionCanvasController-1.dio

Programski kod 38 pokazuje UnitProductionPanel koji predstavlja panel na kojem se nalaze jedinice koje se mogu trenirati. U unitButtonPrefabove se spremaju predlošci svih jedinica koje se mogu trenirati za svaku zgradu. Svaka zgrada također ima i uniQueue koji predstavlja

mogućnost treniranja jedinica po redu, te spawnPoint koji predstavlja mjesto gdje će se jedinica stvoriti.

```
spawnPoint = building.transform.Find("SpawnPoint");

Transform[] iconSlots = {
unitButtonContainer.Find("IconSlot1"),
unitButtonContainer.Find("IconSlot2"),
unitButtonContainer.Find("IconSlot3"),
unitButtonContainer.Find("IconSlot4")

foreach (Transform slot in iconSlots)
{
    foreach (Transform child in slot)
    {
        Destroy(child.gameObject);
    }
    slot.gameObject.SetActive(false);
}
```

Programski kod 39. UnitProductionCanvasController-2. dio

Nakon što se postavi aktivan panel i popuni varijabla spawnpoint. Pronalaze se IconSlotovi, na kojima su prikazane jedinice za treniranje, te ih se čisti prije dodavanja novih, prikazano u kodu 39. U komponenti BuildingUnits se nalazi niz jedinica koje zgrada ima koje se mogu trenirati.

Unutar svakog iconSlota, postavlja se nova ikona za svaku jedinicu. IconObject se ovdje koristi kao UI element za prikazi ikone jedinice. IconImage će dohvatiti sličicu jedinice te će ju prikazati u sučelju.

```
GameObject countTextObject = new GameObject("CountText");
countTextObject.transform.SetParent(iconObject.transform, false);
TextMeshProUGUI countText = countTextObject.AddComponent<TextMeshProUGUI>();
countText.fontSize = 18;
countText.alignment = TextAlignmentOptions.BottomRight;

RectTransform countTextRect = countTextObject.GetComponent<RectTransform>();
countTextRect.anchorMin = countTextRect.anchorMax = countTextRect.pivot = new Vector2(1f, 0f);
countTextRect.anchoredPosition = new Vector2(-10f, 10f);

if (unitQueue.ContainsKey(unitProduction))
{
    countText.text = remainingUnitsToProduce[unitProduction].ToString();
}
else
{
    countText.text = "0";
}

unitCountTexts[unitProduction] = countText;
```

Programski kod 40. UnitProductionCanvasController-3. dio

Osim toga, u donjem desnom dijelu se dodaje i broj jedinica koje se trebaju trenirati. Postavlja se početni broj 0 ako nema jedinica ako se ne nalaze u redu, što se vidi u kodu 40.

```
Button slotButton = iconSlots[index].GetComponent<Button>();
if (slotButton == null)
{
    slotButton = iconSlots[index].gameObject.AddComponent<Button>();
}

slotButton.onClick.RemoveAllListeners();
slotButton.onClick.AddListener(() => AddUnitToQueue(unitProduction, countText));

UnitItem unitItem = iconObject.AddComponent<UnitItem>();

UnitStats unitStats = unitProduction.unitPrefab.GetComponent<UnitStats>();
unitItem.unitStats = unitStats;
```

Programski kod 41. UnitProductionCanvasController-4. dio

Ako ikona nema komponentu gumb, onda će joj se dodati. Klikom na ikonu se dodaje funkcionalnost za dodavanje jedinica u red putem metode AddUnitToQueue(). Komponenta unitItem predstavlja kontroler nad pop-upom prozora koji će se dogoditi kad igrač pređe mišem preko ikone jedinice, te će dobiti informacije o jedinici koje su prije toga izvučene iz komponente unitStats poput zdravlja i napada.

```
public void AddUnitToQueue(UnitProduction unitProduction)
{
    UnitStats unitStats = unitProduction.unitPrefab.GetComponent<UnitStats>();

    if (resourceManager.HasEnoughResources(
        unitStats.woodCost,
        unitStats.stoneCost,
        unitStats.goldCost,
        unitStats.foodCost,
        unitStats.moneyCost,
        unitStats.coalCost,
        unitStats.metalCost
    ))
    {
        resourceManager.DeductResources(
            unitStats.woodCost,
            unitStats.stoneCost,
            unitStats.goldCost,
            unitStats.foodCost,
            unitStats.moneyCost,
            unitStats.coalCost,
            unitStats.metalCost
        );

        if (unitQueue.ContainsKey(unitProduction))
        {
            unitQueue[unitProduction]++;
        }
        else
        {
            unitQueue[unitProduction] = 1;
        }

        if (remainingUnitsToProduce.ContainsKey(unitProduction))
        {
            remainingUnitsToProduce[unitProduction]++;
        }
        else
        {
            remainingUnitsToProduce[unitProduction] = 1;
        }

        if (unitCountTexts.ContainsKey(unitProduction))
        {
            unitCountTexts[unitProduction].text = remainingUnitsToProduce[unitProduction].ToString();
        }
    }
```

Programski kod 42. BuildingProduction-1. dio

Prilikom dodavanja u red, provjerit će se dali igrač ima dovoljno resursa za treniranje te jedinice, što se vidi u programskom kodu 42. Ako ima, dodat će se u red. Također će se i ažurirati broj jedinica koji je preostao za treniranje, kao i tekstualni prikaz broja jedinica ostao za treniranje.

```
private void StartNextUnitInQueue()
{
    if (unitQueue.Count == 0)
    {
        return;
    }

    foreach (var unitProduction in unitQueue.Keys)
    {
        currentUnitProduction = unitProduction;
        break;
    }
    if (currentUnitProduction != null)
    {
        UnitStats unitStats = currentUnitProduction.unitPrefab.GetComponent<UnitStats>();

        if (unitStats != null)
        {
            unitStats.productionSpeed = unitStats.baseProductionSpeed * unitManager.unitProductionMultiplier;
            currentProductionTime = unitStats.productionSpeed;
        }
        else
        {
        }

        // Update queue count and UI
        DecreaseQueueCount(currentUnitProduction);
    }
    else
    {
    }
}
```

Programski kod 43. BuildingProduction-2. dio

U programskom kodu 43 metoda StartNextUniInQueue() započinje treniranje sljedeće jedinice u redu. Prvo se gleda dali postoji koja jedinica u redu te ako nema, onda se završava rad ove metode. Ako postoji, prolazi se kroz ključeve rječnika uniQueue te se postavlja prva jedinica na currentUnitProduction te će ona biti trenirana.

Nakon toga se dohvaćaju UnitStats te jedinice te se uzima početna brzina proizvodnje te jedinice. Stvarna vrijednost brzine proizvodnje se dobije tako da se pomnoži početna vrijednost s multiplikatorom. Nakon što je postavljena ta jedinica, smanjuje se broj jedinica u redu koje treba proizvesti (DecreaseQueueCount()).

```
if (currentUnitProduction != null && currentProductionTime > 0)
{
    currentProductionTime -= Time.deltaTime;

    if (currentProductionTime <= 0)
    {
        CompleteUnitProduction();
    }
}
```

Programski kod 44. BuildingProduction-3. dio

Nakon što je vrijeme proizvodnje spremljeno u varijablu, u Update() dijelu se currentProductionTime smanjuje svakim frameom te kada padne na nulu poziva se metoda CompleteUnitProduction(), prikazana u programskom kodu 45.

```
private void CompleteUnitProduction()
{
    if (spawnPoint != null)
    {
        Vector3 spawnPosition = FindValidSpawnPosition(spawnPoint.position, currentUnitProduction.unitPrefab);
        if (spawnPosition != Vector3.zero)
        {
            GameObject newUnit = Instantiate(currentUnitProduction.unitPrefab, spawnPosition, Quaternion.identity);
            PopulationManager.Instance.AddUnit(newUnit, false);
            SelectionManager.Instance.playerUnits.Add(newUnit);
        }
    }

    if (remainingUnitsToProduce.ContainsKey(currentUnitProduction))
    {
        remainingUnitsToProduce[currentUnitProduction]--;
        if (remainingUnitsToProduce[currentUnitProduction] <= 0)
        {
            remainingUnitsToProduce.Remove(currentUnitProduction);
        }

        if (unitCountTexts.ContainsKey(currentUnitProduction))
        {
            unitCountTexts[currentUnitProduction].text = remainingUnitsToProduce.ContainsKey(currentUnitProduction)
                ? remainingUnitsToProduce[currentUnitProduction].ToString()
                : "0";
        }
    }

    currentUnitProduction = null;
    currentProductionTime = 0;

    if (unitQueue.Count > 0)
    {
        StartNextUnitInQueue();
    }
}
```

Programski kod 45. BuildingProduction-4. dio

Ova metoda završava proces proizvodnje jedinice. Nakon što je pronađena valjana pozicija za stvaranje jedinice (čiji proces će biti objašnjen u sljedećem poglavlju), ta jedinica se stvara na toj poziciji te se dodaje u populaciju, i u SelectionManager koji u sebi sadrži sve jedinice i zgrade (igrača i neprijatelja). Ažurira se broj preostalih jedinica za proizvodnju (remainingUnitsToProduce) i odgovarajući tekstualni prikaz na korisničkom sučelju. Ako više nema jedinica za proizvodnju tog tipa, uklanja se iz reda.

Resetiraju se currentUnitProduction i currentProductionTime varijable. Ako u redu ima još jedinica, započinje se s treniranjem nove pozivanjem StartNextUnitInQueue(). Na slici 14 se može vidjeti red jedinica koje čekaju biti trenirane.

Slika 14. Red jedinica koje čekaju biti trenirane

```
public class PopulationManager : MonoBehaviour
{
    public int maxPopulation = 0;
    private int currentPopulation = 0;
```

Programski kod 46. PopulationManager

Kao što se vidi i u kodu 46, PopulationManager sadrži dvije varijable, maxPopulation i currentPopulation te prati njihovo stanje. Ovisno o populaciji te njezinoj popunjenosti, jedinice se mogu ili ne mogu trenirati.

4.6.2 Proces pronalaženja pozicije za stvaranje

Prije nego što se jedinica stvori u svijetu, potrebno joj je naći ispravnu poziciju za stvaranje. Tu služi metoda FindValidSpawnPosition().

```
private Vector3 FindValidSpawnPosition(Vector3 initialPosition, GameObject unitPrefab)
{
    float checkRadius = 0.5f;
    float initialSpawnRadius = 10f;
    float radiusIncrement = 1f;
    int maxAttemptsPerRadius = 10;
    int maxRadiusExpansions = 5;

    float currentSpawnRadius = initialSpawnRadius;

    for (int expansion = 0; expansion < maxRadiusExpansions; expansion++)
    {
        for (int i = 0; i < maxAttemptsPerRadius; i++)
        {
            Vector3 randomPosition = GetRandomPositionAround(initialPosition, currentSpawnRadius);
            if (!IsPositionOccupied(randomPosition, checkRadius))
            {
                return randomPosition;
            }
        }

        currentSpawnRadius += radiusIncrement;
    }

    return Vector3.zero;
}



1 reference
private bool IsPositionOccupied(Vector3 position, float radius)
{
    LayerMask unitLayerMask = LayerMask.GetMask("PlayerTroops");

    Collider[] colliders = Physics.OverlapSphere(position, radius, unitLayerMask);
    return colliders.Length > 0;
}

1 reference
private Vector3 GetRandomPositionAround(Vector3 originalPosition, float radius)
{
    float randomAngle = Random.Range(0f, 360f);
    float randomDistance = Random.Range(0.5f, radius);

    Vector3 offset = new Vector3(Mathf.Sin(randomAngle) * randomDistance, 0f, Mathf.Cos(randomAngle) * randomDistance);
    return originalPosition + offset;
}
```

Programski kod 47. BuildingProduction-5. dio

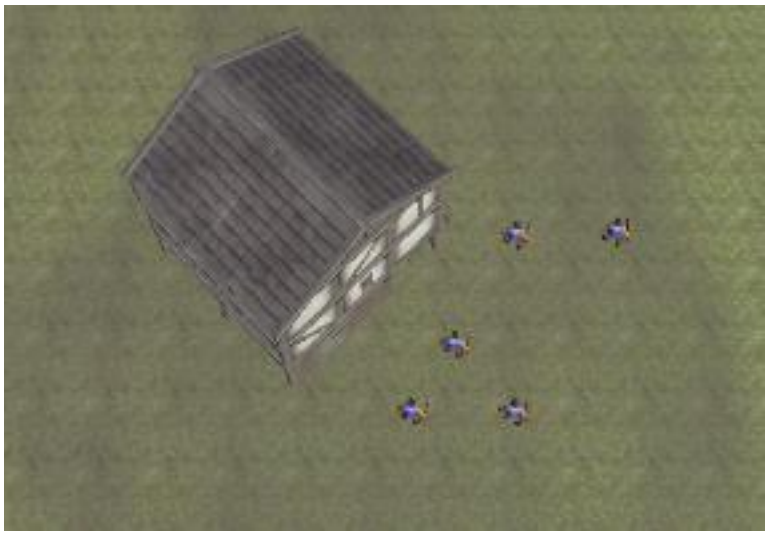
Traženje valjanje pozicije funkcionira na ovaj način: kao što se vidi i u kodu 47, prvo se provjerava pozicija unutar zadanog početnog radijusa te ako nije zauzeta, vraća se kao ispravna. Ako se nije uspjela naći ispravna pozicija unutar radijusa, on se proširuje te se opet pokušava naći ispravna pozicija.

Ako se ipak ne uspije naći ispravna pozicija vraća se Vector3.zero što predstavlja neuspjeh u traženju pozicije.

Metoda IsPositionOccupied vraća true ili false ovisno o tome dali je pozicija koja je primljena kao parametra zauzeta tj. dali se na njoj već nalazi jedinica. Koristi se Physics.OverlapSphere() kako bi se provjerilo da li postoje jedinice unutar određenog radijusa oko pozicije. Ako postoji barem jedna jedinica, metoda vraća true, što znači da je pozicija zauzeta.

Za dobivanje nasumične pozicije unutar određenog radijusa je zaslužna metoda GetRandomPositionAround(). Generira se nasumična pozicija oko neke početne točke koja joj je proslijeđena kao parametar tj. točka stvaranja koja je postavljena za svaku zgradu kao prazan objekt. Korištenjem trigonometrijskih funkcija se dobiva pomak u X i Z osi i vraća se nova

nasumična pozicija. Slika 15 prikazuje kako na kraju izgleda nasumično stvaranje jedinica oko zgrade.



Slika 15. Prikaz nasumičnog stvaranja radnika

4.7. Kretanje jedinica

Nakon što je igrač trenirao određeni broj jedinica, potrebno je te jedinice poslati na resurse, na obranu ili u napad. Za to je potrebno osim sustava kretanja imati i sustav označavanja jedinica koje se želi pomaknuti.

4.7.1. Označavanje jedinica

Svakoj jedinici koju igrač može proizvesti je dodijeljena komponenta SelectableUnit:

```
public class SelectableUnit : MonoBehaviour
{
    Unity Message | 0 references
    private void Start()
    {
        SelectionManager.Instance.unitList.Add(this.gameObject);
    }
```

Programski kod 48. SelectableUnit

Prilikom stvaranja jedinica se dodaje na listu jedinica koje se mogu označiti, prikazano u programskom kodu 48.

Postoje 3 načina označivanja jedinica: klikom, shift + klik i selectboxom.

Za označavanje jedinica koriste se dvije skripte, SelectableUnitClick i SelectionManager.

```
public class SelectableUnitClick : MonoBehaviour
{
    private Camera myCam;

    public LayerMask clickable;
    public LayerMask ground;
    Unity Message | 0 references
    void Start()
    {
        myCam = Camera.main;
    }

    Unity Message | 0 references
    void Update()
    {
        if (Input.GetMouseButtonDown(0))
        {
            RaycastHit hit;
            Ray ray = myCam.ScreenPointToRay(Input.mousePosition);

            if (Physics.Raycast(ray, out hit, Mathf.Infinity, clickable))
            {
                if (Input.GetKey(KeyCode.LeftShift))
                {
                    SelectionManager.Instance.ShiftClickSelect(hit.collider.gameObject);
                }

                else
                {
                    SelectionManager.Instance.ClickSelect(hit.collider.gameObject);
                }
            }
            else
            {
                if (!Input.GetKey(KeyCode.LeftShift))
                {
                    SelectionManager.Instance.DeselectAll();
                }
            }
        }
    }
}
```

Programski kod 49. SelectableUnitClick

Skripta prikazana u kodu 49 služi da bi se unosom putem miša jedinice u igri odabrale. Postavljene su varijable myCam, clickable i ground. U varijabli myCam tipa Camera se sprema glavna kamera igre iz koje će se ispaljivati raycastovi, u varijabli clickable tipa Layer se sprema sloj jedinica koje se može označiti te se u varijabli ground istog tipa sprema sloj terena, koji se koristi u drugoj skripti.

Unutar Update() dijela se provjerava dali je pritisnuta lijeva tipka miša te ako pogodi objekt koji je ranije definiran kao clickable se provjerava dali je tipka Shift pritisnuta, ako je će se pozvati ShiftClickSelect() iz SelectionManagera, ako shift nije pritisnut će se pozvati ClickSelect iz iste skripte. Ako se ne pogodi jedinica označena kao clickable, poziva se metoda DeselectAll() koja odznačuje sve trenutno označene jedinice.

```
public class SelectionManager : MonoBehaviour
{

    [SerializeField] GameObject panelPrefab;
    public List<GameObject> unitList;
    public List<GameObject> unitsSelected;

    public List<GameObject> playerUnits;
    public List<GameObject> playerBuildings;
    public List<GameObject> enemyUnits;
    public List<GameObject> enemyBuildings;
```

Programski kod 50. SelectionManager-1.dio

Skripta SelectionManager prikazana kodom 50 upravlja označavanjem, odznačavanjem i brisanjem jedinica. Varijable te skripte su GameObject panelPrefab koji će se pojaviti ispod jedinice kad je ona označena iz razloga da igrač može bolje uočiti označene jedinice, nakon toga imamo liste unitList i unitsSelected, gdje unitList predstavlja listu svih jedinica u igri dok unitsSelected predstavlja trenutno odabrane jedinice.

Liste playerUnits,playerBuildings,enemyUnits i enemyBuildings sadrže igračeve i neprijateljske zgrade i jedinice.

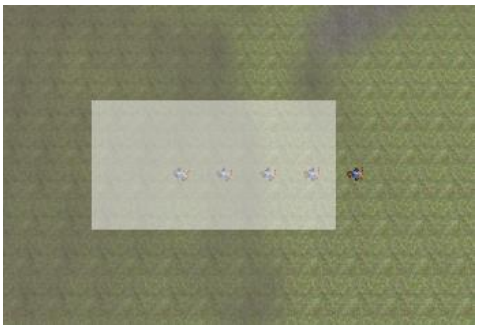
```
public void ClickSelect(GameObject unitToAdd)
{
    DeselectAll();
    AddUnitSelection(unitToAdd);
    unitsSelected.Add(unitToAdd);
    unitToAdd.GetComponent<UnitMovement>().enabled = true;
}

1 reference
public void ShiftClickSelect(GameObject unitToAdd)
{
    if (!unitsSelected.Contains(unitToAdd))
    {
        unitsSelected.Add(unitToAdd);
        AddUnitSelection(unitToAdd);
        unitToAdd.GetComponent<UnitMovement>().enabled = true;
    }
    else
    {
        unitsSelected.Remove(unitToAdd);
        RemoveUnitSelection(unitToAdd);
    }
}

1 reference
public void BoxSelect(GameObject unitToAdd)
{
    if (!unitsSelected.Contains(unitToAdd))
    {
        unitsSelected.Add(unitToAdd);
        AddUnitSelection(unitToAdd);
        unitToAdd.GetComponent<UnitMovement>().enabled = true;
    }
}
```

Programski kod 51. SelectionManager-2. dio

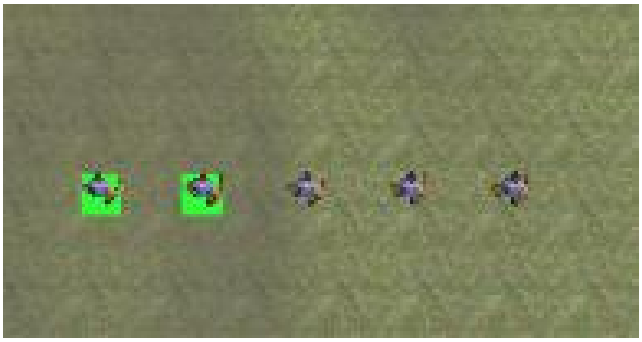
Ove metode predstavljaju ranije spomenute 3 metode označivanja. Prikazano kodom 51, metoda ClickSelect() prvo odznačava sve jedinice, te dodaje kliknutu jedinicu u listu označenih jedinica, te se toj jedinci dodaje komponenta koja je zaslužna za kretanje te jedinice.

ShiftClickSelect metoda omogućava da se više jedinica odabere tako što se drži shift dok se klikće na jedinice koje se želi označiti. Postoji provjera da ako je par jedinica već označeno i ako se klikne na jednu dok se drži shift tipka, ona će biti odznačena. Prikaz toga se nalazi na slici 17.

Uz pomoć BoxSelect metode se može označiti više jedinica odjednom, te se radi tako da igrač povuče kutiju od jedne točke do druge dok drži lijevi klik. Prikaz na slici 16.



Slika 16. Izgled označavanja sa selectboxom



Slika 17. Izgled shiftclick označavanja

Također prilikom označavanja vlastite ili neprijateljske jedinice, može se vidjeti informacija o toj jedinici, kao što se može vidjeti na slici 18.

Slika 18. Izgled opisa jedinice

4.7.2. Pomicanje jedinica

Za kretanje jedinica se nakon njihovog označavanja koristi desni klik miša, te se klikne na određeno mjesto gdje igrač želi da jedinice odu.

```
private void HandleRightClick()
{
    if (Input.GetMouseButtonDown(1))
    {
        Ray ray = myCam.ScreenPointToRay(Input.mousePosition);
        RaycastHit hit;

        resourceGatherer.StopCurrentTask();

        if (Physics.Raycast(ray, out hit, Mathf.Infinity, treeLayer))
        {
            TreeResource tree = hit.collider.GetComponent<TreeResource>();
            if (tree != null)
            {
                resourceGatherer.GoToTree(tree);
                return;
            }
        }

        if (Physics.Raycast(ray, out hit, Mathf.Infinity, buildingLayer))
        {
            BuildingProgress building = hit.collider.GetComponent<BuildingProgress>();
            if (building != null)
            {
                resourceGatherer.GoToBuilding(building);
                return;
            }
        }

        if (Physics.Raycast(ray, out hit, Mathf.Infinity, groundLayer))
        {
            myAgent.SetDestination(hit.point);
        }
    }
}
```

Programski kod 52. UnitMovement

Tu pomaže skripta UnitMovement, koja kada igrač klikne desni klik miša, gleda se koji je objekt Raycast pronašao. Kao što se vidi u programskom kodu 52, ako je detektirao drvo, radnika se npr. šalje da ide sjeći drva. Ako ga se pošalje na zgradu, on će otići do nje i ako nije dovršena, nastavit će gradnju. Ako je detektiran teren, poslat će se na lokaciju koja je kliknuta ranije. Slika 19 prikazuje slanje radnika desno.

Slika 19. Slanje radnika desnim klikom miša

4.8. Istraživanje vještina

Kada sagradi školu ili određene zgrade, igrač ima mogućnost istraživanja vještina koje mu mogu dati prednost nad neprijateljem. Postoje vještine u školi koje može istražiti, te postoje vještine za koje treba bodove. Bodovi se dobivaju izgradnjom specifičnih zgrada po prvi puta.

4.8.1. Vještine iz škole

Prvo će se obraditi vještine koje igrač može istražiti u školi. Prilikom klika na školu u svijetu, otvara se panel kao na slici 20:

Slika 20. Prikaz panela škole

Panel sadrži nekoliko ikona koje funkcioniraju kao i gumbovi, te svaki od njih predstavlja vještinu koja se može istražiti. Neke od njih su brža proizvodnja resursa, brža gradnja zgrada, jači napad jedinica. Također je dodan i pop-up prozor, koji kaže igraču što točno će ta vještina napraviti i koliko košta njeno istraživanje. Klikom na bilo koju od ikona, pokreće se metoda UnlockSkill(), prikaza dolje:

```
public void UnlockSkill(int skillIndex)
{
    Skill skill = skills[skillIndex];

    if (skill.isUnlocked || (skill.requiredSkillIndex != -1 && !skills[skill.requiredSkillIndex].isUnlocked))
    {
        return;
    }

    if (resourceManager.HasEnoughResources(skill.woodCost, skill.stoneCost, skill.goldCost, skill.foodCost, skill.moneyCost, skill.coalCost, skill.metalCost))
    {
        resourceManager.DeductResources(skill.woodCost, skill.stoneCost, skill.goldCost, skill.foodCost, skill.moneyCost, skill.coalCost, skill.metalCost);
        skill.isUnlocked = true;

        ApplySkillEffect(skill);

        if (skill.skillIcon != null)
        {
            Destroy(skill.skillIcon.gameObject);
        }
    }
    else
    {

    }
}
```

Programski kod 53. SkillManager-1.dio

Metoda prikazana u kodu 53 dobiva skillIndex (koji se postavi svakoj vještini na početku) vještine koja se želi istražiti. Gleda se da vještina nije otključana i da nema nekih preduvjeta za istraživanje. Ako su oba uvjeta vratila true, oduzimaju se resursi potrebni za istraživanje te vještine te se njen efekt primjenjuje u ApplySkilEffect() metodi.

```
private void ApplySkillEffect(Skill skill)
{
    switch (skill.effect)
    {
        case Skill.SkillEffect.FoodProduction:
            resourceManager.IncreaseFoodProduction(skill.effectValue);
            break;
        case Skill.SkillEffect.BuildingHP:
            buildingProgress.UpgradeHP(skill.effectValue);
            break;
        case Skill.SkillEffect.BuildSpeed:
            buildingProgress.IncreaseBuildSpeed(skill.effectValue);
            break;
        case Skill.SkillEffect.TreeGathering:
            resourceManager.IncreaseWorkerCapacity(skill.effectValue);
            break;
        case Skill.SkillEffect.MineProduction:
            resourceManager.IncreaseMineProduction(skill.effectValue);
            break;
        case Skill.SkillEffect.Education:
            resourceManager.IncreaseMoneyProduction(skill.effectValue);
            break;
        case Skill.SkillEffect.UnitAttack:
            unitManager.IncreaseUnitAttack(skill.effectValue);
            break;
        case Skill.SkillEffect.BridgeUnit:
            resourceManager.UnlockBridgeUnit();
            break;
    }
}
```

Programski kod 54. SkillManager-2.dio

U metodi ApplySkillEffect() ovisno o pritisnutom gumbu, će se izvršiti metoda povećanja multiplikatora za vještinu povezanu s gumbom.

```
public void IncreaseFoodProduction(float percentage)
{
    foodProductionMultiplier += percentage / 100f;
}

1 reference
public void IncreaseStoneProduction(float percentage)
{
    stoneProductionMultiplier += percentage / 100f;
}

2 references
public void IncreaseMineProduction(float percentage)
{
    mineProductionMultiplier += percentage / 100f;
}

2 references
public void IncreaseMoneyProduction(float percentage)
{
    moneyProductionMultiplier += percentage / 100f;
}
```

Programski kod 55. ResourceManager

Nakon toga, kao što se vidi u kodu 55, multiplikatori za povećaju za određeni postotak, kao što se može vidjeti u primjeru iznad gdje se događa povećavanja multiplikatora za proizvodnju resursa.

4.8.2. Vještine iz stabla vještina

Osim vještina iz škole, igrač može istražiti vještine u stablu vještina. Te vještine daju snažnija poboljšanja, ali postoji ograničen broj koji se može istražiti.

```
private void Start()
{
    buildingManager = FindObjectOfType<BuildingManager>();
    unitManager = FindObjectOfType<UnitManager>();
    buildingManager.OnBuildingCompleted += AwardSkillPoint;
    resourceManager = FindObjectOfType<ResourceManager>();

    UpdateSkillPointsUI();
}

1 reference
private void AwardSkillPoint(string buildingName)
{
    if (usableSkillPoints < totalSkillPoints)
    {
        usableSkillPoints++;
        UpdateSkillPointsUI();
    }
}
```

Programski kod 56. SkillTreeManager

TotalSkillPoints je ograničen na 6, te se dobije izgradnjom određenih tipova zgrada po prvi put. Nakon što se zgrada izgradi, poziva se metoda AwardSkillPoint, koja igraču dodjeljuje bod koji može iskoristiti. To se može vidjeti na slici 21 i u programskom kodu 56.

Slika 21. Prikaz vještina stabla vještina i opis vještine

Isto kao i kod vještina iz škole, multiplikatori se također povećavaju za određeni postotak. Kao što se može vidjeti na slici 22 nakon uzimanja poboljšanja napada za 10 %, sada radnik ima napad 11.

Slika 22. Poboljšanje napada

4.9. Funkcije neprijatelja

Kako su obrađene sve funkcionalnosti kod igrača, sada će se objasniti neprijateljeve mogućnosti.

4.9.1. Startna faza

Neprijatelj započinje svoju igru na drugom kraju terena u usporedbi s igračem, te igra započinje sa skriptom StartLevel.

```
public class StartLevel : MonoBehaviour
{
    public GameObject playerUnitPrefab;
    public Transform playerSpawnPoint;
    public int startingPlayerUnits = 5;
    public float spawnSpacing = 2.0f;

    public GameObject enemyUnitPrefab;
    public Transform enemySpawnPoint;
    public int startingEnemyUnits = 5;

    Unity Message | 0 references
    void Start()
    {
        SpawnUnits(playerUnitPrefab, playerSpawnPoint, startingPlayerUnits, true);
    }

    1 reference
    void SpawnUnits(GameObject unitPrefab, Transform spawnPoint, int unitCount, bool isPlayerUnit)
    {
        for (int i = 0; i < unitCount; i++)
        {
            Vector3 spawnPosition = spawnPoint.position + new Vector3(i * spawnSpacing, 0, 0);
            GameObject unit = Instantiate(unitPrefab, spawnPosition, Quaternion.identity);

            UnitMovement unitMovement = unit.GetComponent<UnitMovement>();
            if (unitMovement != null)
            {
                if (isPlayerUnit)
                {
                    unitMovement.enabled = false;
                    SelectionManager.Instance.unitList.Add(unit);
                    SelectionManager.Instance.playerUnits.Add(unit);
                }
                else
                {
                    unitMovement.enabled = true;
                    SelectionManager.Instance.enemyUnits.Add(unit);
                }
            }
        }
    }
}
```

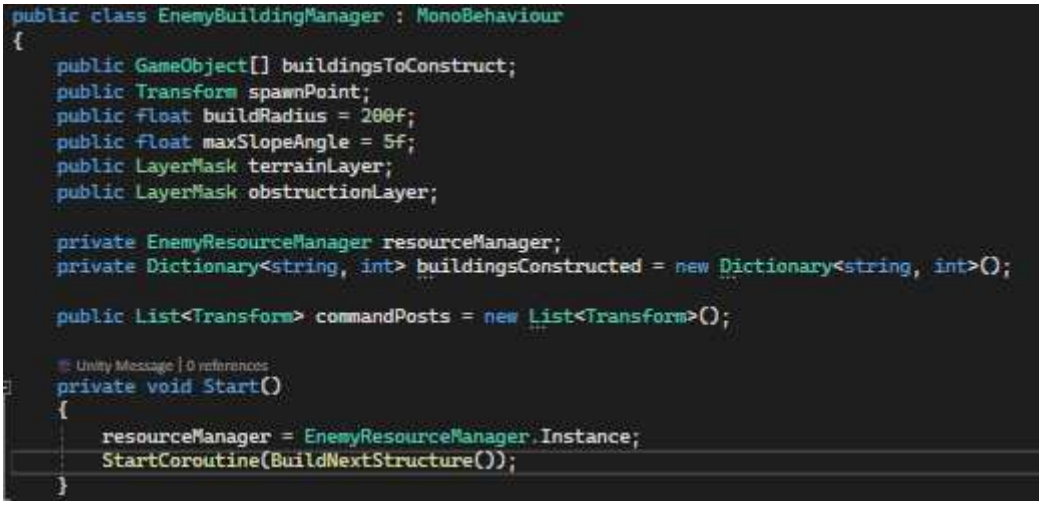
Programski kod 57. StartLevel

Kao što se vidi i u kodu 57, start neprijatelja i igrača je određen pozicijom enemySpawnPointa i playerSpawnPointa. Oboje startaju s 5 radnika (jedinice koje su postavljene kao unit prefab) te ovisno dali je neprijatelj ili igrač, će ih dodati u odgovarajuću listu jedinica.

Naravno, neprijatelj kreće u igru i sa svojim početnim resursima te pripadajućim EnemyResourceManagement skriptom, koja kao kod igrača kontrolira dobivanje i potrošnju resursa.

4.9.2. Gradnja zgrada i proizvodnja resursa

Što se tiče gradnje zgrada, neprijatelj ima svoj menadžment gradnje zgrada te se sastoji od sljedećih elemenata, prikazanih u programskom kodu 58:



Programski kod 58. EnemyBuildingManager-1. dio

Gameobjekt buildingsToConstruct koji u sebi sadrži sve predloške zgrada koje neprijatelj može izgraditi, spawnPoint koji predstavlja početnu lokaciju neprijatelja, varijabla buildRadius unutar koje neprijateljev AI traži prostor za izgradnju zgrada, maxSlopeAngle koji predstavlja najveći dopušteni nagib terena, te sloj na kojem će neprijatelj graditi zgrade (teren). U obstructionLayer ulaze svi elementi koje neprijatelj treba izbjeći prilikom gradnje zgrade.

Neprijatelj prati izgrađene zgrade u rječniku buildingsConstructed. Unutar metode Start() se započinje korutina kojom će se nastojati sagraditi sve zgrade koje se mogu sagraditi.

```
IEnumerator BuildNextStructure()
{
    while (true)
    {
        foreach (GameObject buildingPrefab in buildingsToConstruct)
        {
            BuildingCost cost = buildingPrefab.GetComponent<BuildingCost>();

            if (!CheckBuildLimit(cost))
            {
                continue;
            }

            if (!CheckPrerequisites(buildingPrefab))
            {
                continue;
            }

            while (!resourceManager.HasEnoughResources(cost.woodCost, cost.stoneCost, cost.goldCost, cost.foodCost, cost.moneyCost, cost.coalCost, cost.metalCost))
            {
                yield return new WaitForSeconds(1f);
            }

            resourceManager.DeductResources(cost.woodCost, cost.stoneCost, cost.goldCost, cost.foodCost, cost.moneyCost, cost.coalCost, cost.metalCost);
            Vector3 buildPosition = FindBuildPosition(buildingPrefab);

            if (buildPosition == Vector3.zero)
            {
                yield return new WaitForSeconds(2f);
                continue;
            }

            RaycastHit hit;
            if (Physics.Raycast(buildPosition + Vector3.up * 10, Vector3.down, out hit, Mathf.Infinity, terrainLayer))
            {
                buildPosition = hit.point;

                Quaternion buildingRotation = GetCustomBuildingRotation(buildingPrefab);

                GameObject builtBuilding = Instantiate(buildingPrefab, buildPosition, buildingRotation);
                BuildingProgress buildingProgress = builtBuilding.GetComponent<BuildingProgress>();
                if (buildingProgress != null)
                {
                    buildingProgress.CompleteBuilding();
                }

                SelectionManager.Instance.enemyBuildings.Add(builtBuilding);
                IncrementBuildingCount(cost.name);

                if (builtBuilding.name.Contains("CommandPost"))
                {
                    EnemyResourceGatherer resourceGatherer = FindObjectOfType<EnemyResourceGatherer>();
                    if (resourceGatherer != null)
                    {
                        resourceGatherer.SetCommandPost(builtBuilding.transform);
                    }
                }
            }
        }
        yield return new WaitForSeconds(2f);
    }
```

Programski kod 59. EnemyBuildingManager-2.dio

Korutina BuildNextStructure() prikazana u kodu 59 upravlja gradnjom zgrada za neprijatelja. Petlja je postavljena da se vrti da se izgrade sve zgrade iz liste buildingsToConstruct, te se nakon toga za svaku uzme njezina cijena i provjerava se je li napravljen maksimalni broj tog tipa zgrade. Sljedeća provjera nakon te je provjera je li neprijatelj ispunio uvjete za izradu te zgrade, te ako su ova dva uvjeta zadovoljena gleda se dali neprijatelj ima dovoljno resursa. Ako ima oduzimaju mu se te se uz pomoć metode FindBuildPosition pronalazi pozicija na kojoj bi neprijatelj mogao sagraditi zgradu.

```
Vector3 FindBuildPosition(GameObject buildingPrefab)
{
    int maxAttempts = 999;
    float checkRadius = 5f;

    for (int i = 0; i < maxAttempts; i++)
    {
        Vector3 randomPosition = spawnPoint.position + Random.insideUnitSphere * buildRadius;
        randomPosition.y = spawnPoint.position.y + 10f;

        RaycastHit hit;
        if (Physics.Raycast(randomPosition, Vector3.down, out hit, Mathf.Infinity, terrainLayer))
        {
            Vector3 buildPosition = hit.point;

            if (buildPosition.y <= 52.5f)
            {
                continue;
            }

            float slopeAngle = Vector3.Angle(hit.normal, Vector3.up);
            if (slopeAngle <= maxSlopeAngle)
            {
                Collider[] obstructions = Physics.OverlapSphere(buildPosition, checkRadius, obstructionLayer);
                if (obstructions.Length == 0)
                {
                    return buildPosition;
                }
                else
                {

                }
            }
        }
    }

    return Vector3.zero;
}
```

Programski kod 60. EnemyBuildingManager-3.dio

Kao što se može vidjeti i u kodu 60 ova metoda pronalazi ispravnu poziciju na kojoj bi neprijatelj mogao postaviti zgradu. Varijabla maxAttemps predstavlja maksimalan broj pokušaja traženja pozicije, dok checkRadius gleda dali postoji neka prepreka (poput drveta, zgrade) u nekom postavljenom radijusu.

Za odabir nasumične pozicije uzima se početna pozicija neprijatelja te se uzima pozicija, te će se generirati nasumični Vector3 unutar sfere buildRadiusa. Također je postavljeno da pozicija mora biti na ravnom terenu, da mora biti manja od prije postavljenog maxSlopeAnglea. Za kraj se gleda dali postoje prepreke u radijusu te pozicije, te ako ne postoje, vraća se pozicija.

Nakon što je pronađena pozicija, instancira se nova zgrada na toj poziciji. Nasuprot igraču gdje igrač treba radnike da bi dovršili gradnju zgrade, neprijatelj ne mora slati radnike na gradnju zgrade, nego je napravljeno da on čim postavi zgrade da one budu označene kao dovršene.

```
public class EnemyResourceGatherer : MonoBehaviour
{
    public LayerMask resourceLayer;
    public float gatherRange = 50f;
    public int maxPeasants = 5;
    private Transform enemyCommandPost;

    private List<GameObject> enemyPeasants = new List<GameObject>();
    private List<TreeResource> availableResources = new List<TreeResource>();
```

Programski kod 61. EnemyResourceGatherer-1. dio

Za skupljanje resursa kod neprijatelja je zaslužna skripta EnemyResourceGatherer, prikazana u kodu 61. Unutar nje se postavlja sloj resourceLayer koji predstavlja stabla, gatherRange koji predstavlja radijus unutar kojeg će se tražiti resurs (stabla), broj radnika koje će slati na to te referencu na upravljački center gdje će donijeti resurse. Lista enemyPeasants sadržava listu svih neprijateljevih radnika, a lista availableResource sadržava listu svih stabala unutar gatherRangea.

```
private void FindNearbyResources()
{
    Collider[] hitColliders = Physics.OverlapSphere(enemyCommandPost.position, gatherRange, resourceLayer);
    availableResources.Clear();

    foreach (var hitCollider in hitColliders)
    {
        TreeResource resource = hitCollider.GetComponent<TreeResource>();
        if (resource != null && resource.woodAmount > 0)
        {
            availableResources.Add(resource);
        }
    }
}
```

Programski kod 62. EnemyResourceGatherer-2. dio

Lista availableResource se puni na ovaj način, unutar gatherRangea koje ima središte u zgradi se pronalaze kolideri koji imaju sloj resourceLayer te se spremaju u hitColldiers.. Ti kolideri se nakon toga vrte unutar petlje i gleda se dali imaju komponentu TreeResource te ako imaju tu komponentu i woodAmount im nije 0, spremaju se u listu.

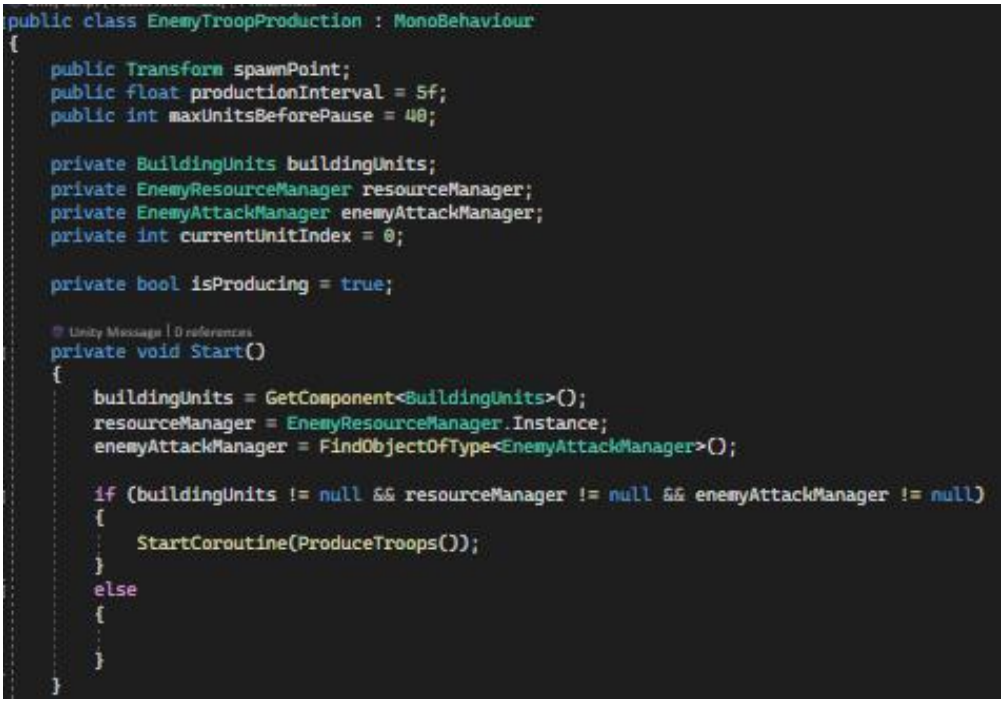
```
private void AssignPeasantsToGatherResources()
{
    if (enemyCommandPost == null)
    {
        Debug.LogWarning("No command post found for enemy peasants.");
        return;
    }

    FindNearbyResources();

    int peasantsAssigned = 0;
    foreach (GameObject peasant in enemyPeasants)
    {
        if (peasantsAssigned >= maxPeasants) break;

        ResourceGatherer gatherer = peasant.GetComponent<ResourceGatherer>();
        if (gatherer != null && availableResources.Count > 0)
        {
            NavMeshAgent navMeshAgent = gatherer.GetComponent<NavMeshAgent>();
            if (navMeshAgent == null)
            {
                navMeshAgent = gatherer.gameObject.AddComponent<NavMeshAgent>();
            }

            TreeResource nearestTree = availableResources[0];
            gatherer.AssignCommandPost(enemyCommandPost);
            gatherer.GoToTree(nearestTree);
            peasantsAssigned++;
        }
    }
}
```

Programski kod 63. ResourceGatherer-3. dio

Dodijeljivanje radnika koji će ići po resurse funkcionira na ovaj način: kao što je prikazano u kodu 63, gledaju se radnici iz liste radnika te se gleda dali imaju komponentu ResourceGatherer, ako je imaju uzima se prvo stablo iz liste stabala, radnicima se dodjeljuje upravljački center gdje će donositi resurse, i šalje ih se prema stablu. Oni kao i igračevi radnici prolaze kroz istu petlju dolaska do stabla, sječe i vraćanja do zgrade.

4.9.3. Treniranje jedinica

Neprijatelj kao i igrač ima sposobnost treniranja jedinica koje će kasnije slati u napad na igrača. Za treniranje jedinica je zaslužna skripta EnemyTroopProduction.



```
public class EnemyTroopProduction : MonoBehaviour
{
    public Transform spawnPoint;
    public float productionInterval = 5f;
    public int maxUnitsBeforePause = 40;

    private BuildingUnits buildingUnits;
    private EnemyResourceManager resourceManager;
    private EnemyAttackManager enemyAttackManager;
    private int currentUnitIndex = 0;

    private bool isProducing = true;

    Unity Message | 0 references
    private void Start()
    {
        buildingUnits = GetComponent<BuildingUnits>();
        resourceManager = EnemyResourceManager.Instance;
        enemyAttackManager = FindObjectOfType<EnemyAttackManager>();

        if (buildingUnits != null && resourceManager != null && enemyAttackManager != null)
        {
            StartCoroutine(ProduceTroops());
        }
        else
        {

        }
    }
```

Programski kod 64. EnemyTroopProduction-1. dio

Programski kod 64 prikazuje EnemyTroopProduction skriptu koja je komponenta svake neprijateljske zgrade koja može trenirati jedinice. Spawnpoint kao i kod igračevih zgrada predstavlja poziciju gdje će se jedinice početno postaviti, productionInterval predstavlja vrijeme potrebno za treniranje dok maxUnitsBeforePause predstavlja broj jedinica koji će biti istrenirano prije prestanka treniranja o čemu će više biti rečeno kasnije.

BuildingUnits sadrži jedinice koje zgrada može trenirati, resourceManager kontrolira resurse, EnemyAttackManager kontrolira napad jedinica i currentUnitIndex predstavlja trenutni indeks jedinice koje se trenira. Na početku se pokreće korutina ProduceTroops, čije objašnjenje slijedi u nastavku.

```
IEnumerator ProduceTroops()
{
    while (true)
    {
        if (!isProducing || enemyAttackManager.IsAttackInProgress())
        {
            yield return new WaitForSeconds(productionInterval);
            continue;
        }

        if (SelectionManager.Instance.enemyUnits.Count >= maxUnitsBeforePause)
        {
            yield return new WaitForSeconds(productionInterval);
            continue;
        }

        yield return new WaitForSeconds(productionInterval);

        if (buildingUnits.units.Length == 0) yield break;

        UnitProduction unitToProduce = GetNextUnit();
        if (unitToProduce != null && CanAffordUnit(unitToProduce))
        {
            ProduceUnit(unitToProduce);
        }
    }
}

1 reference
private void ProduceUnit(UnitProduction unitProduction)
{
    Vector3 spawnPosition = FindValidSpawnPosition(spawnPoint.position, unitProduction.unitPrefab);
    if (spawnPosition != Vector3.zero)
    {
        GameObject newUnit = Instantiate(unitProduction.unitPrefab, spawnPosition, Quaternion.identity);
        SelectionManager.Instance.enemyUnits.Add(newUnit);
        enemyAttackManager.enemyUnits.Add(newUnit);
        DeductUnitCost(unitProduction);
    }
}
```

Programski kod 65. EnemyTroopProduction-2.dio

Unutar korutine ProduceTroops() neprijatelj će proizvoditi jedinice pod 3 uvjeta, da je boolean isProducing postavljen na true i da neprijatelj nije poslao napad na igrača te da je broj neprijateljevih jedinica manji od maxUnitsBeforePause. Petlja je zaustavljena na broj sekundi određen productionIntervalom, te se dohvaća prva jedinica metodom GetNextUnit(), što se vidi i u programskom kodu 65.

```
private UnitProduction GetNextUnit()
{
    if (currentUnitIndex >= buildingUnits.units.Length)
    {
        currentUnitIndex = 0;
    }

    UnitProduction unitToProduce = buildingUnits.units[currentUnitIndex];
    currentUnitIndex++;
    return unitToProduce;
```

Programski kod 66. EnemyTroopProduction-3. dio

Metoda GetNextUnit() prikazana kodom 66 dohvaća jedinicu koja će se trenirati ovisno o njenom indeksu. Nakon što je jedinica dohvaćena vrijednost varijable currentUnitIndex se uvećava za 1, tako da kad se korutina opet pozove, drugačija jedinica će biti stvorena. Nakon

što su sve jedinice stvorene jedanput, currentUnitIndex se resetira te se opet počinje s treniranjem prve trenirane jedinice. Ovime neprijatelj ima raznovrsnost unutar vojske.

Ako neprijatelj ima dovoljno resursa, metoda ProduceUnit() će se pozvati koja prima predložak jedinice koju treba trenirati.

Unutar metode će se pronaći ispravna pozicija kao i kod igrača, te će se instancirati nova jedinica na toj poziciji. Također će se dodat u popis neprijateljevih jedinica, kao i u popis jedinica koje će kasnije poslati u borbu.

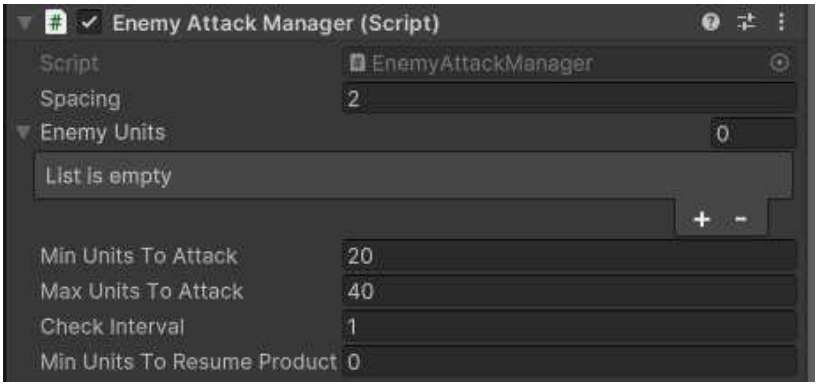
4.9.4. Borba protiv igrača

Za borbu protiv igrača, zaslužne su dvije skripte, UnitStateMachine i EnemyAttackManager. EnemyAttackManager odlučuje kada će neprijatelj ići u borbu dok je uloga UnitStateMachinea tijekom borbe kada radi promatranje i traži igračeve jedinice.

```
private void Start()
{
    unitsRequiredForAttack = Random.Range(minUnitsToAttack, maxUnitsToAttack);

    enemyUnitManager = GetComponent<EnemyUnitManager>();

    if (enemyUnitManager == null)
    {
        return;
    }

    StartCoroutine(CheckUnitCount());
}

1 reference
private IEnumerator CheckUnitCount()
{
    while (!attackLaunched)
    {
        yield return new WaitForSeconds(checkInterval);

        if (enemyUnits.Count >= unitsRequiredForAttack && !attackLaunched)
        {
            LaunchAttack();
        }
    }
}
```

Programski kod 67. EnemyAttackManager-1. dio

EnemyAttackManager na početku odabire nasumični broj između minimalnog i maksimalnog broja jedinica koje može poslati u napad. Te dva parametra su postavljena na početku unutar Inspectora. U početku se također pokreće korutina CheckUnitCount() koja prati neprijateljev broj jedinica za napad što se vidi u programskom kodu 67. Kada neprijatelj sakupi dovoljan broj jedinica, pokreće se napad metodom LaunchAttack(). Slika 23. prikazuje kako to izgleda postavljeno.



Slika 23. Isječak komponente EnemyAttackManager

```
private void LaunchAttack()
{
    attackLaunched = true;

    EnemyTroopProduction[] troopProducers = FindObjectsOfType<EnemyTroopProduction>();
    foreach (EnemyTroopProduction producer in troopProducers)
    {
        producer.StopProduction();
    }

    StartCoroutine(AttackPlayerTargets());
}

1 reference
private IEnumerator AttackPlayerTargets()
{
    while (enemyUnits.Count > minUnitsToResumeProduction)
    {
        GameObject playerTarget = FindNearestPlayerTarget();

        foreach (var unit in enemyUnits)
        {
            if (unit != null)
            {
                UnitStateMachine stateMachine = unit.GetComponent<UnitStateMachine>();
                if (stateMachine != null)
                {
                    stateMachine.SetTarget(playerTarget);
                }
            }
        }

        while (playerTarget != null)
        {
            enemyUnitManager.MoveTroopsToTarget(enemyUnits, playerTarget.transform.position);


            if (IsTargetDestroyed(playerTarget))
            {
                playerTarget = null;
                break;
            }

            yield return new WaitForSeconds(0.5f);
        }


        playerTarget = FindNearestPlayerTarget();
    }


    EnemyTroopProduction.ResumeAllProductions();
    attackLaunched = false;
}
```

Programski kod 68. EnemyAttackManager-2. dio

Unutar metode LaunchAttack() prikazanoj u kodu 68, nakon što se pronađu sve zgrade koje treniraju jedinice, zaustavlja se treniranje za svaku. Nakon toga se pokreće korutina AttackPlayerTargets().

Korutina se odvija sve dok neprijatelj ostane bez jedinica poslanih u napad. Dok neprijatelj ima jedinica, traži se najbliža meta koju neprijatelj može napasti, tj. najbliža igračeva jedinica ili zgrada.

```
private GameObject FindNearestPlayerTarget()
{
    List<GameObject> playerUnits = SelectionManager.Instance.playerUnits;
    List<GameObject> playerBuildings = SelectionManager.Instance.playerBuildings;

    GameObject nearestTarget = null;
    float nearestDistance = Mathf.Infinity;


    foreach (var unit in playerUnits)
    {
        if (unit == null) continue;
        float distance = Vector3.Distance(unit.transform.position, transform.position);
        if (distance < nearestDistance)
        {
            nearestDistance = distance;
            nearestTarget = unit;
        }
    }


    foreach (var building in playerBuildings)
    {
        if (building == null) continue;
        float distance = Vector3.Distance(building.transform.position, transform.position);
        if (distance < nearestDistance)
        {
            nearestDistance = distance;
            nearestTarget = building;
        }
    }

    return nearestTarget;
}
```

Programski kod 69. EnemyAttackManager-3. dio

Proces pronalaska najbliže mete opisuje programski kod 69, metoda inicijalizira dvije liste, jednu s igračevim jedinicama a drugu sa zgradama. Postavlja se početna vrijednost varijable nearestDistance.

Nakon toga petlja prolazi kroz svaku jedinicu i zgradu te ako nađe neku jedinicu ili zgradu koja ima manju udaljenost od neprijatelja, nju postavlja kao nearestTarget.

Nakon što je pronađena najbliža meta, svakoj neprijateljskoj jedinice se pretražuje dali ima komponentu EnemyStateMachine. Ako ima, poziva se metoda SetTarget().

```
public void SetTarget(GameObject target)
{
    if (target == null) return;

    UnitStats targetStats = target.GetComponent<UnitStats>();
    BuildingProgress targetBuilding = target.GetComponent<BuildingProgress>();

    if (targetStats != null)
    {
        currentTarget = targetStats;
        currentBuildingTarget = null;
        ChangeState(UnitState.MovingToTarget);
    }
    else if (targetBuilding != null)
    {
        currentBuildingTarget = targetBuilding;
        currentTarget = null;
        ChangeState(UnitState.MovingToBuilding);
    }
}
```

Programski kod 70. UnitStateMachine-1. dio

SetTarget() izgleda ovako, traži se komponenta UnitStats ili BuildingProgress ovisno dali se radi o jedinici ili zgradi. Ako targetStats nije null, znači da se radi o jedinici te će neprijateljeve jedinice krenuti do nje a ako targetBuilding nije null, znak je da se radi o zgradi.

Nakon toga u skripti EnemyAttackManager neprijatelj miče svoje jedinice do igračeve mete, te ako priđe dovoljno blizu, umanjuje joj zdravlje. Ako je jedinica ili zgrada uništena, neprijatelj će ponovno krenuti s potragom.

```
private void Start()
{
    unitStats = GetComponent<UnitStats>();
    navMeshAgent = GetComponent<NavMeshAgent>();
    animator = GetComponent<Animator>();

    StartCoroutine(ContinuousDetectAndAttack());
}

private void Update()
{
    if (currentTarget != null && Vector3.Distance(transform.position, currentTarget.transform.position) <= unitStats.attackRange)
    {
        ChangeState(UnitState.Attacking);
        if (!isAttacking)
        {
            StartCoroutine(AttackEnemyUnit());
        }
    }
    else if (currentBuildingTarget != null && Vector3.Distance(transform.position, currentBuildingTarget.transform.position) <= unitStats.attackRange)
    {
        ChangeState(UnitState.Attacking);
        if (!isAttacking)
        {
            StartCoroutine(AttackEnemyBuilding());
        }
    }
    else
    {
        DetectAndAttack();
    }
}
```

Programski kod 71. UnitStateMachine-2. dio

Dok se jedinice kreću, cijelo vrijeme im se odvija korutina ContinuousDetectAndAttack(), što se vidi u kodu 71.

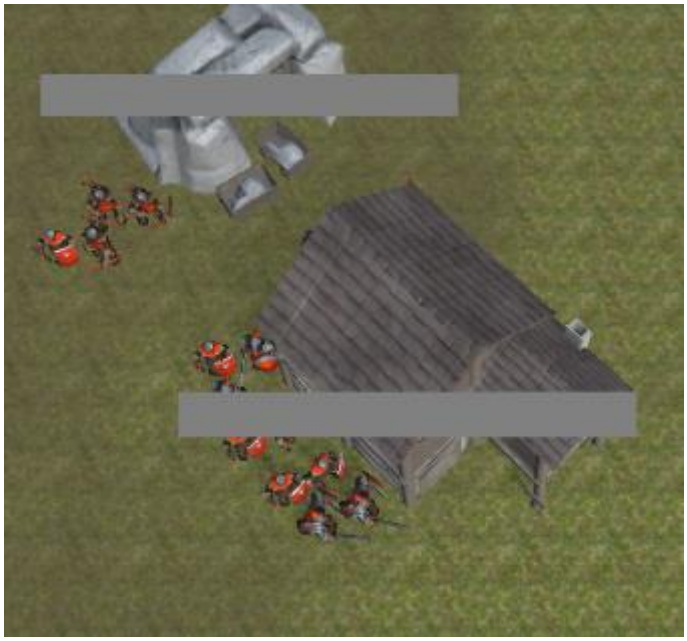
```
private IEnumerator ContinuousDetectAndAttack()
{
    while (unitStats.IsAlive)
    {
        DetectAndAttack();
        yield return new WaitForSeconds(0.5f);
    }
}

private void DetectAndAttack()
{
    currentTarget = null;
    currentBuildingTarget = null;

    Collider[] enemiesInRange = Physics.OverlapSphere(transform.position, detectionRange, enemyLayer);

    if (enemiesInRange.Length > 0)
    {
        float closestDistance = detectionRange;
        foreach (Collider enemy in enemiesInRange)
        {
            UnitStats enemyStats = enemy.GetComponent<UnitStats>();
            if (enemyStats != null && enemyStats.IsAlive)
            {
                float distance = Vector3.Distance(transform.position, enemy.transform.position);
                if (distance < closestDistance)
                {
                    currentTarget = enemyStats;
                    closestDistance = distance;
                }
            }
        }

        if (currentTarget != null)
        {
            MoveToTarget();
            return;
        }
    }

    Collider[] buildingsInRange = Physics.OverlapSphere(transform.position, detectionRange, buildingLayer);

    if (buildingsInRange.Length > 0)
    {
        float closestBuildingDistance = detectionRange;
        foreach (Collider building in buildingsInRange)
        {
            BuildingProgress buildingProgress = building.GetComponent<BuildingProgress>();
            if (buildingProgress != null && buildingProgress.currentBuildPoints > 0)
            {
                float distance = Vector3.Distance(transform.position, building.transform.position);
                if (distance < closestBuildingDistance)
                {
                    currentBuildingTarget = buildingProgress;
                    closestBuildingDistance = distance;
                }
            }
        }

        if (currentBuildingTarget != null)
        {
            MoveToBuilding();
        }
    }
}
```

Programski kod 72. EnemyStateMachine-3. dio

Svrha korutine prikazana u programskom kodu 72 je da dok god je neprijateljeva jedinica živa, može pratiti određen radijus oko sebe te ako primijeti igračevu jedinici ili zgradu dok se kreće, da nju napadne. Ovo je napravljeno iz razloga da neprijatelj ne bi slijepo išao prema jedinici koju je odredio prije.

Svaka jedinica ima određeni radijus definiran od detectionRangea, te unutar njega traži kolidere koje imaju sloj enemyLayer. Ako ga imaju, jedinica će krenuti prema tom objektu.

Isto vrijedi i za zgrade, ako jedinica primijeti zgradu unutar radijusa, promijenit će metu te će prvo napasti nju, kao što se vidi na slici 24.



Slika 24. Neprijatelj napada

```
private IEnumerator AttackEnemyUnit()
{
    isAttacking = true;
    while (currentTarget != null && currentTarget.IsAlive)
    {
        yield return new WaitForSeconds(unitStats.attackSpeed);

        int damage = Mathf.Max(unitStats.attack - currentTarget.defense, 0);
        currentTarget.TakeDamage(damage);

        if (!currentTarget.IsAlive)
        {
            currentTarget = null;
            isAttacking = false;

            DetectAndAttack();
            yield break;
        }
    }
    isAttacking = false;
}

1 reference
private IEnumerator AttackEnemyBuilding()
{
    isAttacking = true;
    while (currentBuildingTarget != null && currentBuildingTarget.currentBuildPoints > 0)
    {
        yield return new WaitForSeconds(unitStats.attackSpeed);

        currentBuildingTarget.TakeDamage(unitStats.attack);

        if (currentBuildingTarget.currentBuildPoints <= 0)
        {
            if (SelectionManager.Instance != null && SelectionManager.Instance.playerBuildings.Contains(currentBuildingTarget.gameObject))
            {
                SelectionManager.Instance.playerBuildings.Remove(currentBuildingTarget.gameObject);
            }

            Destroy(currentBuildingTarget.gameObject);

            currentBuildingTarget = null;
            isAttacking = false;

            DetectAndAttack();
            yield break;
        }
    }
    isAttacking = false;
}
```

Programski kod 73. EnemyStateMachine-4. dio

Jedinice su cijelo vrijeme spremne za napad. U ranije prikazanoj metodi Update(), jedinice mogu pozvati jednu od dvije metode, AttackEnemyUnit() ili AttackEnemyBuilding().

Ako se radi o jedinici, pozvat će se AttackEnemyUnit(), koja će raditi štetu jedinici koja se napada ovisno o brzini napada. Ako meta više nije živa, ponovno se poziva DetectAndAttack().

Kod napada zgrade je isto, zgrada prima štetu svakih nekoliko sekundi te kad više nema zdravlja, ukloni je se s igračeve liste zgrada. Nakon toga se ponovno traži nova jedinica ili zgrada za napasti.

4.10. Zvuk

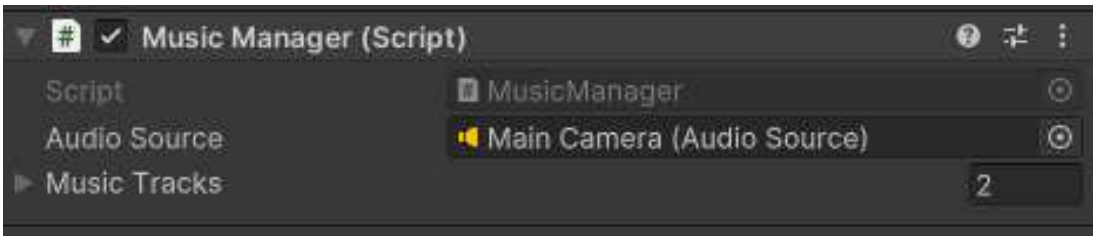
Osim vizualnog, ne treba zaboraviti na učinak zvuka u igrama, gdje stvara atmosferu i poboljšava doživljaj igre.

```
public class MusicManager : MonoBehaviour
{
    public AudioSource audioSource;
    public AudioClip[] musicTracks;
    private int currentTrackIndex = 0;

    Unity Message | 0 references
    void Start()
    {
        PlayNextTrack();
    }

    Unity Message | 0 references
    void Update()
    {
        if (!audioSource.isPlaying)
        {
            PlayNextTrack();
        }
    }

    2 references
    void PlayNextTrack()
    {
        audioSource.clip = musicTracks[currentTrackIndex];
        audioSource.Play();

        currentTrackIndex = (currentTrackIndex + 1) % musicTracks.Length;
    }
}
```

Programski kod 74. MusicManager

Programski kod 74 prikazuje zaduženu skriptu za zvuk, skriptu MusicManager, koja upravlja reprodukcijom glazbe u igri. Skripta automatski pušta pjesme iz niza i, kada jedna završi, prelazi na sljedeću izvršavanjem metode PlayNextTrack(). Nakon što se sve pjesme iz niza

reproduciraju, glazba se vraća na prvu pjesmu, omogućujući neprekidnu reprodukciju u petlji. Glazba korištena u MusicManageru je preuzeta iz Unity Asset Storea [9].



Slika 25. MusicManager komponenta

4.11. Završetak igre

Kraj igre u igrama strategije u stvarnom vremenu označava trenutak kada se ispune uvjeti za pobjedu poput uništavanja neprijateljevih jedinica i zgrada, osvajanja nekih područja ili ispunjavanja ciljeva neke misije. U ovoj igri kraj igre se događa kada neprijatelj ili igrač uništi sve protivničke jedinice i zgrade.

U ranije spomenutoj skript SelectionManager se nalaze sve jedinice i zgrade na terenu, igračeve i neprijateljske.

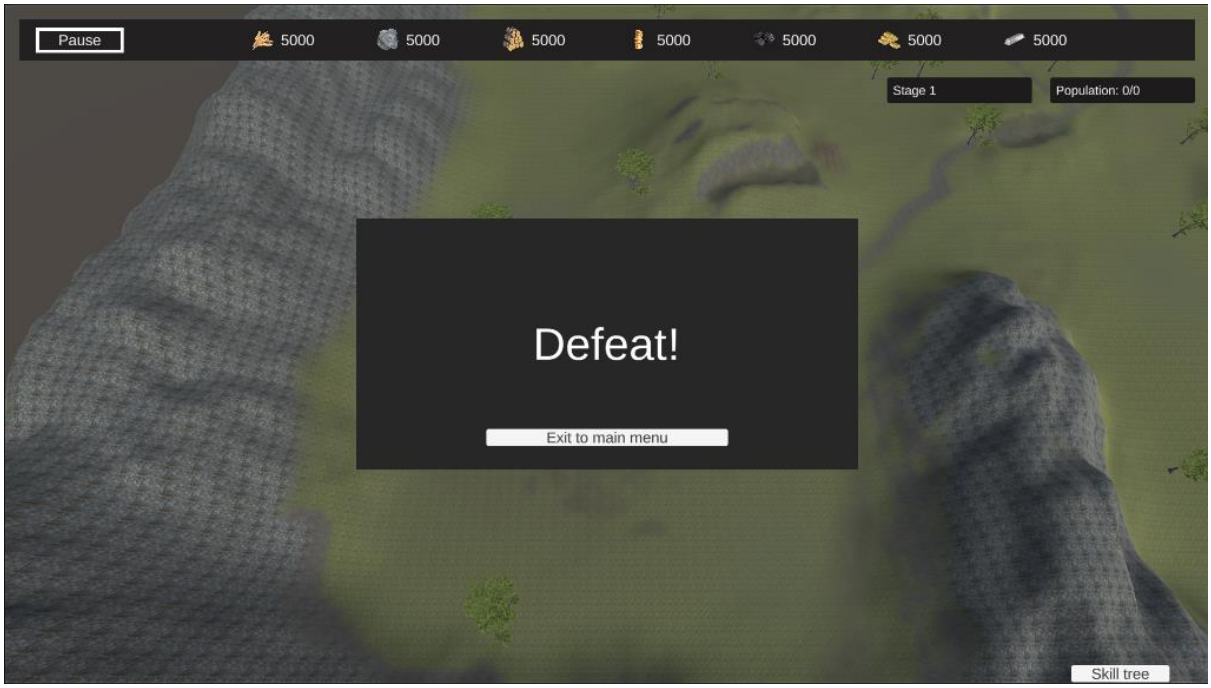
```
public class GameManager : MonoBehaviour
{
    public GameObject victoryScreen;
    public GameObject defeatScreen;

    private SelectionManager selectionManager;

    Unity Message | 0 references
    void Start()
    {
        selectionManager = SelectionManager.Instance;
    }

    Unity Message | 0 references
    void Update()
    {
        CheckVictoryOrDefeat();
    }

    1 reference
    void CheckVictoryOrDefeat()
    {
        if (selectionManager.playerUnits.Count == 0 && selectionManager.playerBuildings.Count == 0)
        {
            ShowDefeatScreen();
        }
        else if (selectionManager.enemyUnits.Count == 0 && selectionManager.enemyBuildings.Count == 0)
        {
            ShowVictoryScreen();
        }
    }

    1 reference
    void ShowVictoryScreen()
    {
        victoryScreen.SetActive(true);
        Time.timeScale = 0f;
    }

    1 reference
    void ShowDefeatScreen()
    {
        defeatScreen.SetActive(true);
        Time.timeScale = 0f;
    }
}
```

Programski kod 75. GameManager

Kao što se vidi u programskom kodu 75 u ovoj se skripti na početku inicijaliziraju 2 panela, jedan za pobjedu a drugi za poraz igrača te se varijabla selectionManager postavlja na instancu klase SelectionManager da se dobije pristup svim jedinicama i zgradama igrača i neprijatelja.

Metoda CheckVictoryOrDefeat() se pokreće unutar Update(), tako da se poziva svaki frame.

Provjerava se broj igračevih jedinica i zgrada te ako bude jednak nuli, prikazuje se panel poraza kao što se može vidjeti na slici. U slučaju da je broj neprijateljevih jedinica i zgrada jednak nuli, prikazuje se panel pobjede. U oba slučaja se pauzira igra te igrač ima gumb s kojim izlazi u glavni meni. Slika 26 prikazuje izgled ekrana ako igrač izgubi.



Slika 26. Prikaz poraza igrača

5. Zaključak

Kroz ovaj rad su opisani elementi i proces razvoja igre u Unity okruženju, vezani uz igre strategije u stvarnom vremenu. Pružen je proces i tijek rada ovog žanra igre počevši od resursa, zgrada pa do vještina i borbi. Unity se pokazao kao odličan odabir za ovakav tip projekta.

Iako su postignuti značajni rezultati poput implementacije osnovnog sustava neprijateljske umjetne inteligencije, postoje područja na kojima bi se moglo poraditi. Postoje stvari koje su se još mogle napraviti, ali nisu mogle zbog ograničenja i kompleksnosti. Neprijateljska AI u real-time strategijskim igrama predstavlja poseban izazov jer mora biti sposobna donositi brze i efikasne odluke u dinamičnom okruženju. Razvijanje sustava koji može prilagođavati strategije, reagirati na promjene na bojištu te donositi dugoročne taktičke odluke zahtijeva značajnu količinu vremena i pažnje.

Jedan od ključnih izazova u tom procesu je upravljanje resursima i koordinacija različitih aspekata AI sustava. Neprijateljski AI mora biti sposoban optimalno prikupljati i trošiti resurse, planirati izgradnju zgrada, te trenirati i raspoređivati jedinice na strateški ispravan način. Balansiranje tih elemenata postaje kritično kako bi igra pružila pravi izazov, a da ne postane prelagana ili preteška za igrača.

Osim neprijatelja, moguće poboljšanje je i raznovrsnija selekcija jedinica i zgrada. Na primjer, dodavanje specijaliziranih jedinica s jedinstvenim sposobnostima moglo bi značajno povećati složenost i dinamiku igre, jer bi igračima pružilo više opcija za kreiranje strategija, kao i veći broj potencijalnih odgovora na neprijateljske prijetnje.

Ovaj rad predstavlja temelj za daljnji razvoj igre te služi kao polazišna točka za buduća istraživanja i nadogradnje. Uvođenjem novih elemenata, optimizacijom postojećih sustava i proširivanjem mogućnosti neprijateljskog AI-a, igra bi mogla postati još izazovnija i privlačnija za igrače.

https://github.com/Kristijan-Kekic/PeasantsEvolve

Sažetak

Tema ovog diplomskog rada je bila izrada igre strategije u stvarnom vremenu koristeći Unity okruženje. U radu su detaljno opisani elementi i skripte korišteni u ovoj igri,te kako su one povezane međusobno. Objašnjeni su ključni aspekti igara ovog žanra poput načina dobivanja i potrošnje resursa, građenja zgrada, treniranja i kretanja jedinica te borbu protiv neprijatelja, što je bio cilj ovog rada. Tijekom izrade igre, naišlo se na određene izazove koje rad može istaknuti. Sama kompleksnost neprijateljevog AI sustava te njegova implementacija i problem ograničenog vremena su utjecali na nemogućnost implementacije svih planiranih značajki. Unatoč ovim izazovima, predložena su konkretna poboljšanja na nekoliko aspekta neprijateljevog sustava poput sustava odlučivanja, resursnog menadžmenta i koordinacije, te uključivanje raznovrsnijih jedinica i zgrada.

Ključne riječi: Unity, RTS, programiranje, videoigra, razvoj igre

Abstract

Topic of this masters's thesis was the development of a real-time strategy (RTS) game using the Unity environment. This paper provides detailed description of the elements and scripts used in this game, and how the scripts are connected. Key aspects of this genre are explained, such as resource gathering and consumption, building construction, unit training and movement, as well as combat against enemies. During game development, certain challenges arose which are also highlighted by the paper. In particular the complexity of the enemy's AI system and its implementation and the problem of limited time caused the disability to implement all planned features. In spite of these challenges, several specific improvements are proposed for various aspects of the enemy system, including the decision-making system, resource management, and coordination, and also include more diverse units and buildings.

Keywords: Unity, RTS, programming, video game, game development

Literatura

[1] Zenva (2023.), https://gamedevacademy.org/what-is-unity/, datum pristupa 15. 3. 2024.

[2] Udemy (2021), https://blog.udemy.com/unity-game-engine/, datum pristupa 18. 3. 2024.

[3] Microsoft(2024.), https://www.ageofempires.com/games/aoe/, datum pristupa 20. 3. 2024.

[4] GSC Game World (2023.), https://www.cossacks3.com/en, datum pristupa 21. 3. 2024.

[5] Unity Technologies (2023.), Unity documentation, https://docs.unity.com/, datum pristupa 24. 3. 2024.

[6] Unity Technologies (2023.), Physics Raycast, https://docs.unity3d.com/ScriptReference/Physics.Raycast.html, datum pristupa 26. 3. 2024.

[7] GBAndrewGB, (2. 5. 2018.), „VILLAGE HOUSES PACK“, https://assetstore.unity.com/packages/3d/characters/village-houses-pack-63695

[8] Dogmatic, (23. 10. 2019.), „AQUAS Lite - Built-In Render Pipeline“, https://assetstore.unity.com/packages/vfx/shaders/aquas-lite-built-in-render-pipeline-53519

[9] Owl Theory Music, (9. 10. 2023.), „Fantasy Ambience FREE SONG | Music Pack“, https://assetstore.unity.com/packages/audio/music/fantasy-ambience-free-song-music-pack-258807

[10] A dog's life software, (11. 1. 2021.), „Outdoor Ground Textures“, https://assetstore.unity.com/packages/2d/textures-materials/floors/outdoor-ground-textures-12555

[11] Laxer, (7. 8. 2017.), „Modular Wooden Bridge Tiles“, https://assetstore.unity.com/packages/3d/props/exterior/modular-wooden-bridge-tiles-29501

[12] Polygon Blacksmith, (25. 5. 2020.), Toon RTS Units, https://assetstore.unity.com/packages/3d/characters/toon-rts-units-67948

Popis slika

Popis programskog koda