Matematikai és Informatikai Intézet

Mesterséges intelligencia a számítógépes játékokban

Készítette
Pataki Tamás
Programtervező Informatikus BSc

Témavezető
Dr. Kovásznai Gergely
Egyetemi docens

Eger, 2025

Tartalomjegyzék

Bevezetés

A szakdolgozati a téma választás idején sok lehetőség volt elérhető számomra. Végül amikor választanom kellet olyan témát szerettem volna amely egy sajátos kihívást ad és egyben az érdekeltségi közömbe is tartozik. A számítógépes játékok és azoknak a fejlesztése már korábban is érdekelt, és ez adta meg a löketet a téma választásához.

Emellett a harckocsik között történő csaták is érdekelnek, főleg a II. világháborúban használatos fegyverekkel. Emiatt egy olyan játékot szerettem volna fejleszteni melyben meglehetett tapasztalni az akkori időben kifejlesztetett harckocsiknak az erejét. Csavarként gondoltam ki azt a funkciót, hogy a különböző harckocsik elemeit meglehessen cserélni egymás közt, ezzel új lehetőségeket nyitva a taktikákra és egy fantázia elemet belerakva a játékba.

Emiatt választottam a projektem megvalósítására Unity[1] játékmotort, mely lehetőséget adott ennek kifejlesztésére 3D térben.

A szakdolgozatom az alábbi linken érhető el: `https://github.com/patakitamas2002/Tank-Game`

1. fejezet

Tématerület áttekintése

2. fejezet

Felhasznált technológiák

2.1. Unity

A fejlesztésem alapjának a Unity-t választottam népszerűsége és sokoldalúsága miatt. Rengeteg eszközt ad a fejlesztő kezébe, így szinte bármilyen típusú projektet lehet vele készíteni, akár a játékfejlesztésen túl is.

2.1.1. A Unity-ről általánosan

A Unity először 2005-ben jelent meg, amióta folyamatos továbbfejlesztés átváltoztatta teljesen. 2D és 3D játékok feljlesztésére szolgál több platformon is, emellett interaktív szimulációkra használható és virtuális valóságra való fejlesztésre is van támogatása.

Adott bevételig ingyenes linceszel rendelkezik emiatt az indie és újonc játékfejlesztők körében nagyon elterjedt és tág közösségre tett szert. Emiatt sokféle oktatóvideó található, és rengeteg közösség által készített játék tartozékot (pl. hangok, modellek) lehet letölteni vagy megvásárolni.

2.1.2. Gameobject-ek

Egyik alapeleme a Unity-nek a GameObjectek, melyeknek rengeteg féle felhasználási módjuk van. Bármilyen karakter, pálya vagy tárgy egy Gameobject egy jeleneten belül. Alapból nincsen konkrét funkciójuk, de komponenseket lehet hozzájuk csatolni, melyek meghatározás a működésüket. Ezért a GameObjecteket tekinthetjük úgy, mint egy tárolóegység.

2.1.3. Komponensek

A komponensek felelnek a GameObject-ek viselkedéséért, rengeteg különféle funkcionalitást nyújtanak. Alapból rengetek komponens van beépítve a Unity-be, de a szkriptelési

funkcionalitással saját komponensekkel lehet ezt a tárat tovább bővíteni. A leggyakrabban használt beépített komponensek:

- Transform (Ezt minden GameObject tartalmazza)
- Mesh renderer
- Collider
- Rigidbody
- Szkriptek

2.1.4. Szkriptelés

A szkriptek egyféle komponens a GameObjectnek. A játék logikájának és interakcióinak megírására szolgálnak. A szkriptek főként C# nyelven készülnek, és az egyes játékobjektumok viselkedését, mozgását, valamint a játékbeli eseményeket szabályozzák.

A Monobehavior-ből nevű osztályból öröklődik a legtöbb szkript, mely többféle beépíttet életciklus tesz elérhetővé:

- *Start*
- *Update*
- *FixedUpdate*
- *OnCollisionEnter / OnTriggerEnter*

A *Start* metódus egyszer fut le a szkript indításakor, amikor az adott objektum betöltődik, és az objektum engedélyezve van. Általában inicializálási feladatokhoz van alkalmazva, például változók értékeinek beállítását hajtja végre.

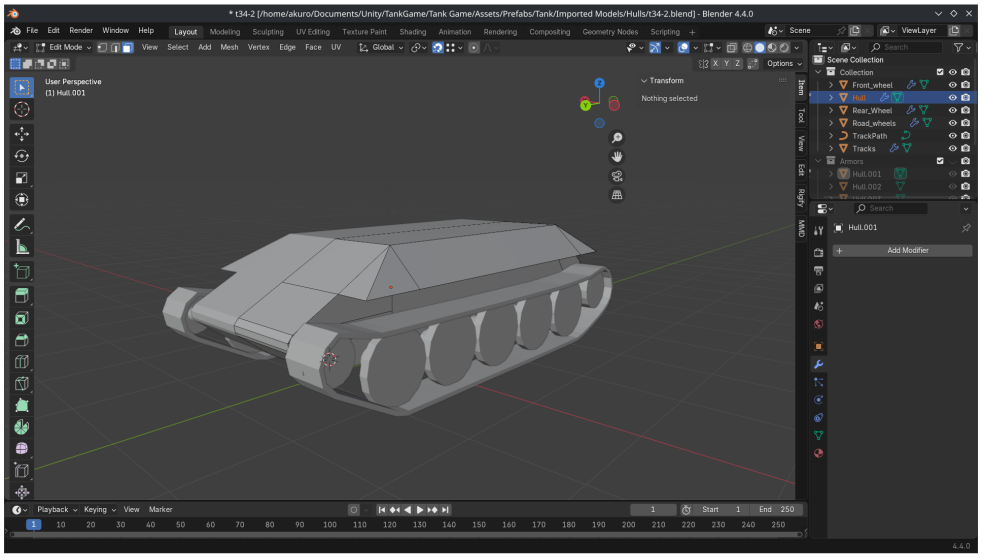
Az *Update* metódus minden képkockánál lefut, így ide kerülnek azok a kódok, amelyeknek folyamatosan, valós időben kell futniuk, például a mesterség intelligenciák irányítása vagy a kezelőfelületen levő folyamatosan frissülő részek.

A *FixedUpdate*, az *Update*-hez hasonlóan, folyamatosan fut, viszont minden fizikai képkockán fut le, a megjelenített képkockáktól függetlenül. Ide a fizikai számítások kerülnek, mivel megadott időközönként fut újra.

Az *OnCollisionEnter* akkor fut le amikor két tárgy amelyek van *Collider* komponense összeütközik. Ütközések vagy fizika lövedékek kezelésére alkalmas.

2.2. Blender

A modellezéshez Blender-t választottam, mely egy ingyenes és nyílt forráskódú 3D modellező és animációs program. A Blender egy nagyon széles körben használt és alkalmazott program a 3D grafika világában, és ingyenessége miatt könnyen kipróbálható és használható.



2.1. ábra. Harckocsi modell Blender-ben

A Blender újabb verzióiban rengeteg eszközzel rendelkezik, mely egyszerűsíti a modellezést és annak tanulását. Emellett tág felhasználói közössége miatt rengeteg segítség található internetes fórumokon, illetve rengeteg tanító videó is áll rendelkezésre, melyek nagyrészben segítettek e programnak a használatában.

3. fejezet

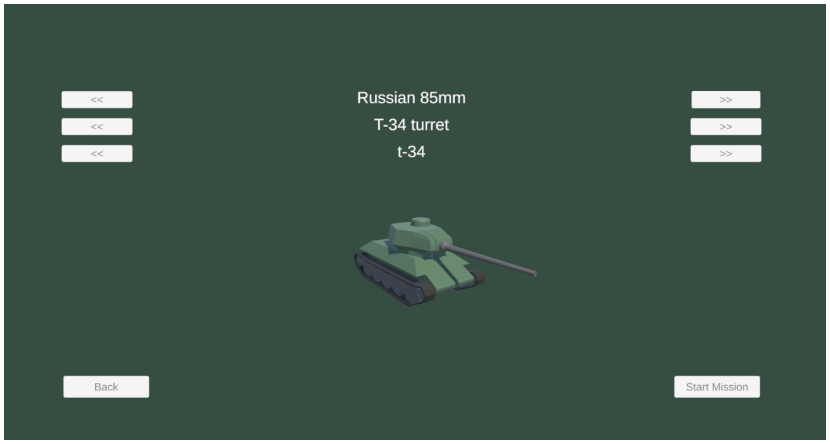
A játék bemutatása

3.1. A játékmenet

3.1.1. Menü

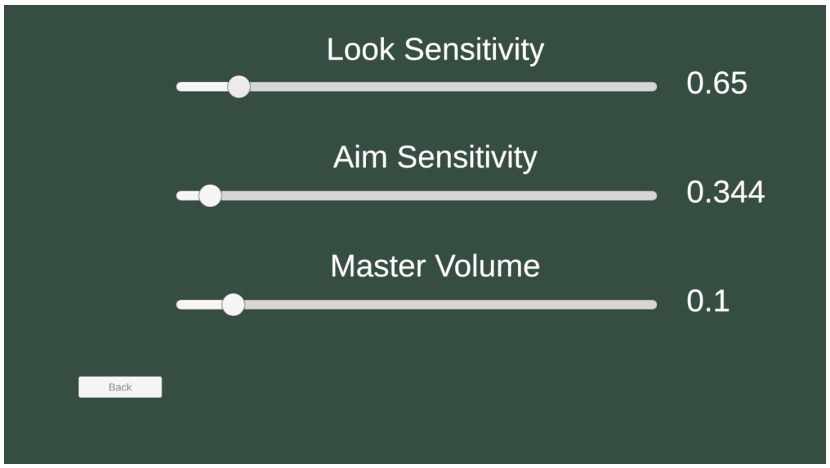
A játékot elindításakor a főmenübe kerülünk, ahol 3 gomb fogad, a játékmenet elindítása, beállítások és a kilépés.

A játékmenet elindítására kattintva megjelenik a harckocsi kiválasztására szolgáló menü. Ebben menüben a felső részen a gombok segítségével választhatóak a harckocsi részei, és a gombok között található azoknak a megnevezése. A képernyő közepén látható a jelenleg kiválasztott harckocsi előnézete, mely minden alkatrész váltás során változik. A jobb alsó sarokban található a játék indító gomb, mely indítja a játékmenetet és a bal alsó sarokban a főmenübe visszalépő gomb.



3.1. ábra. Harckocsi választási menü

A beállításoknál 3 csúszka található, melyek mellett a jelenlegi értékük található. Ezek szolgálnak a játékbeli érzékenység és a hangerő beállítására.



3.2. ábra. Beállítási menü

3.1.2. Csata

A tank kiválasztása után elindul a csata, ahol egy sziklás pályán jelenik meg a játékos kiválasztott harckocsi, és egy csoport ellenséges harckocsit, melyek egy útvonalat követnek célpontjukig. A játékos célja, hogy az ellenfeleket megsemmisítse mielőtt az ellefnelek a célpontjukhoz érnek. Ha bármelyik ellenfél eléri a célpontot vagy a játékos harckocsija megsemmisül akkor a játékos veszít.

3.2. A megvalósítás

3.2.1. A harckocsi

A *Tank* komponens sok dologért felel. Legfőként a különböző részek létrehozásáért, statisztikáiknak az összekombinálásáért és az életerő-t is kezeli.

3.1. kód. A harckocsi létrehozása

```
1 public static GameObject CreateTank(GameObject hull, GameObject turret,
      GameObject barrel, HealthBar hpbar, Transform transform)
2 {
3     Tank newTank = new GameObject("Tank", typeof(Tank), typeof(Rigidbody),
          typeof(BoxCollider)).GetComponent<Tank>();
4     newTank.transform.position = transform.position;
5     newTank.hull = Instantiate(hull.GetComponent<Hull>(), newTank.transform)
          ;
6     newTank.turret = Instantiate(turret.GetComponent<Turret>(), newTank.hull
          .transform.GetChild(0).transform);
7     newTank.barrel = Instantiate(barrel.GetComponent<Barrel>(), newTank.
          turret.transform.GetChild(0).transform);
8     newTank.healthBar = hpbar;
9     return newTank.gameObject;
```

```
10 }
```

Ez a komponens egyéni tapadási fizikáért is felel, mely gyorsabb sebességnél történő fordulásnál nagy mértékben csökkenti a harckocsi sebességét és a fordulás irányába viszi a tankot. Ennek hiányában forduláskor nem lenne tapadás és oldalirányban tudna csúszni. A *SidewaysFriction* a *FixedUpdate*-ben fut le akkor, amikor a harckocsi legalább egyik lánctalpa a földön van.

3.2. kód. Tapadási kód

```
1 private const float sidewaysFrictionFactor = 0.04f;
2 private void SidewaysFriction()
3 {
4     Vector3 rightDirection = transform.right;
5     float sidewaysSpeed = Vector3.Dot(rb.velocity, rightDirection);
6     Vector3 sidewaysVelocity = rightDirection * sidewaysSpeed;
7     Vector3 newVelocity = rb.velocity - sidewaysVelocity *
          sidewaysFrictionFactor;
8     rb.velocity = newVelocity;
9 }
```

A harckocsi 3 részre van felosztva:

- Páncéltest
- Lövegtorony
- Ágyú

Ezentúl, ha a harckocsi élete 0 vagy annál kisebb értékre csökken le, akkor megsemmisültnek számít, mely könnyen észrevehető abból, hogy a lövegtorony lerepül ennek esetén.

Páncélzati rendszer

Minden harckocsi modellje fel van osztva több kis darabra, melynek van a beépített *Collider* komponense és a saját páncél komponense. Ezzel a módszerrel pontosan meg lehet adni hogy mely részen mekkora védelme van és mekkora sebzési szorzót kap. Például az ágyúnak és a lánctalpnak kisebb szorzója van, mivel könnyebben átüthető részeknek számítanak, miközben a harckocsi középpontja nem kap sebzést.

Páncéltest

Ez a rész adja meg az egész harckocsi teljes sebességét, gyorsulását, a motor hangját és az életerejének a nagy részét. Valamint hozzá tartoznak a lánctalpak is, melyek

ellenőrzik, hogy a harckocsi a földön van-e egy adott pillanatban egy rövid sugár segítségével, mely megnézi hogy a terep-be ütközött-e be. Általánosan ez rendelkezik a legtöbb páncél elemmel is. Valamint a test adja meg a lövegtorony elhelyezkedését.

Lövegtorony

A lövegtorony forgatását a játékos nézőpontjából kilőtt sugár segítségével implementáltam, mely visszaad egy koordinátát, és miután a lövegtorony ezt megkapja, elkezd fordulni megadott sebességgel annak irányába. Emellett a lövegtorony adja meg a belső kamera pozícióját, az ágyú elhelyezkedését is.

3.3. kód. Lövegtorony forgatása

```
1  public void RotateTowards(Transform aimPoint)
2  {
3      Vector3 targetDirection = (aimPoint.position - transform.position).
           normalized;
4      float target = Mathf.Atan2(targetDirection.x, targetDirection.z) * Mathf
           .Rad2Deg - transform.parent.eulerAngles.y;
5      Quaternion trav = Quaternion.Euler(0, target, 0);
6      transform.localRotation = Quaternion.RotateTowards(transform.
           localRotation, trav, stats.RotationSpeed * Time.deltaTime);
7  }
```

Ágyú

Az ágyú emelése hasonlóan működik a toronyhoz forgatásához képest. Viszont megadott a maximum magassági és depressziószögük, melyeken nem mehet túl a függőleges fordulása.

3.4. kód. Lövegtorony irányítása

```
1  public void Elevate(Transform aimPoint)
2  {
3      Vector3 targetDirection = aimPoint.position - transform.position;
4      float range = Mathf.Sqrt(targetDirection.x * targetDirection.x +
           targetDirection.z * targetDirection.z);
5      float target = Mathf.Atan2(-targetDirection.y, range) * Mathf.Rad2Deg -
           transform.parent.eulerAngles.x;
6      if (target < -180) target += 360;
7      target = Math.Clamp(target, -stats.maxElevation, stats.maxDepression);
8      Quaternion trav = Quaternion.Euler(target, 0, 0);
9      transform.localRotation = Quaternion.RotateTowards(transform.
           localRotation, trav, stats.ElevationSpeed * Time.deltaTime);
10 }
```

Játékos nézőpontja

A játékos nézőpontja 2 darab kamerával van megoldva, külső és belső nézetes, melyeknek az érzékenységét külön-külön lehet beállítani. Az egérgörgővel lehet csökkenteni a kamera látószögét, segíthet nagyobb távolságon történő célzásban.

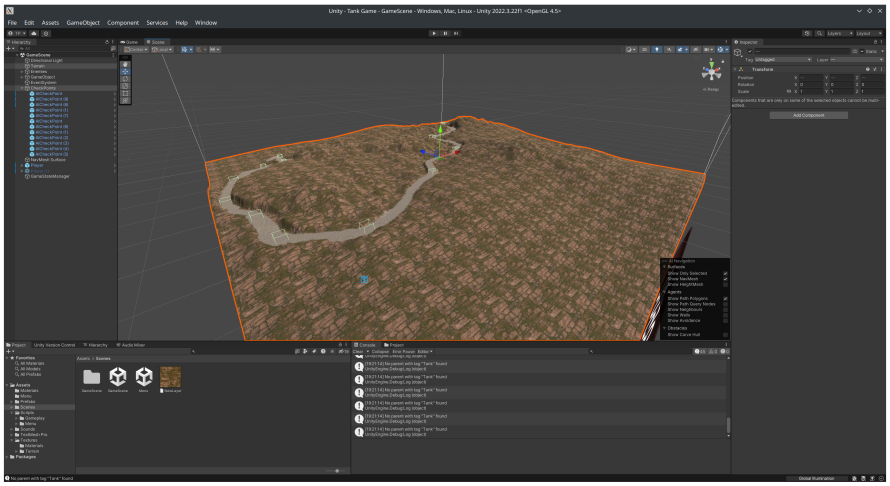
3.2.2. Pálya

Kialakítása

A pálya kialakításához a Unity terepszerkesztőjét használtam, mely ecsetszerű eszközöket segítségével a terepen magasabb és mélyebb részeket lehetett rajzolni. Így egy hegyes-dombos pályát készítettem egy kivájt útvonallal.

Textúrázás

A textúrázáshoz szintúgy a terepszerkesztő eszközöket használtam, mellyel az útvonalat egy ecsettel lehetett festeni. A textúrákat a *Poly Haven*[4] oldalról töltöttem le.

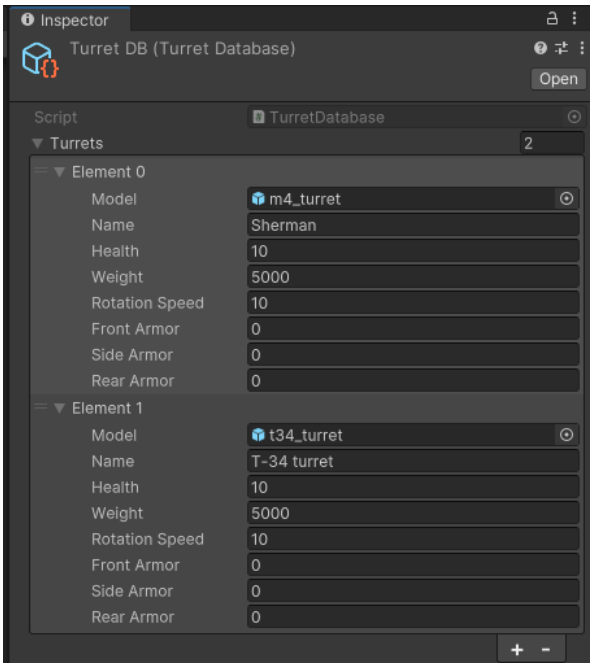


3.3. ábra. A Unity szerkesztőjében a pálya

3.2.3. Menü

Harckocsi kiválasztása

A harckocsi részeit a Unity-ba beépített *ScriptableObject*-ek segítségével tudtam eltárolni és ezekből újra elérni a harckocsi kiválasztásakor.



3.4. ábra. *ScriptableObject* kezelőfelülete

Beállítások

A beállításokban az érzékenységet és a hangerőt lehet állítani, melyekre csúszkák szolgálnak. Ezeket az értékeket

Szünet menü

Egy egyszerűbb szünet menü van a játékban, mellyel újralehet indítani a játékmenetet vagy visszamenni a fő menübe. Ilyenkor a játékmenet teljesen megáll illetve a kurzor újra látható és mozgathatóvá válik.

3.2.4. Irányítási rendszer

A bevitelhez a Unity új beviteli rendszerét használtam segítségül, melynek használatával egyszerűen lehetett újabb beviteli billentyűket hozzáadni, valamint azokat csoportosítani és együttesen ki vagy bekapcsolni.

Célzás

A célzást az egér mozgásának olvasása teszi lehetővé. A játékmenet közben a kurzor viszont nem látható és a játékablak közepére van lezárva, amíg a játékos fel nem hozza a szünet menüt vagy a játékmenetnek vége nem lesz.

3.2.5. Lövedékek

A lövedékek fizikai objektumként működnek, megadott sebességük és súlyuk van, viszont gravitáció nincs rájuk hatással.

Három féle lövedék van beimplementálva jelenleg a játékba:

- páncéltörő
- robbanótöltetes pácéltörő
- robbanótöltetes lövedék

Mindhárom típus egy absztrakt *Bullet* osztályból öröklődik, így a *Start* és az *OnCollisionEnter* metódusaik közösek, valamint implementálniuk kell *GetMaxPenetration, CalculateDMG* és*CalculatePenetration*

3.5. kód. Absztrakt *Bullet* osztály metódusai

```
1  public abstract class Bullet : MonoBehaviour
2  {
3      void Start()
4      {
5          startPosition = transform.position;
6          remainingPen = GetMaxPenetration();
7          gameObject.GetComponent<Rigidbody>().AddForce(transform.forward *
               Veloctiy, ForceMode.VelocityChange);
8          transform.localScale = Vector3.one * Caliber / 100;
9          Destroy(gameObject, maxTime);
10     }
11     void OnCollisionEnter(Collision collision)
12     {
13         Debug.Log("Collision");
14         if (hasCollided)
15         {
16             return;
17         }
18         float distanceTravelled = Vector3.Distance(startPosition, transform.
               position);
19         hitArmor = collision.contacts[0].otherCollider.GetComponent<Armor>();
20         if (hitArmor == null)
21         {
22             Debug.Log("Hit␣non-Armored␣object");
23             hasCollided = true;
24             return;
25         }
26         remainingPen = CalculatePenetration(collision, distanceTravelled);
27         if (remainingPen <= 0)
28         {
```

```
29              Debug.Log("Hit␣armor,␣no␣Penetration␣left");
30              hasCollided = true;
31              return;
32          }
33
34          hitArmor.RegisterDamage(CalculateDMG());
35          Debug.Log("Damage␣dealt:␣" + CalculateDMG());
36          hasCollided = true;
37      }
38      protected abstract float GetMaxPenetration();
39      protected abstract float CalculateDMG();
40      protected abstract float CalculatePenetration(Collision collision, float
            distanceTravelled);
41
42  }
```

Páncéltörő lövedék

A páncél törő lövedék a DeMarre formula alapján lett modellezve. Ez adja meg a súly, átmérő és sebesség alapján a maximális átütési erejét egy adott lövedéknek.

$$P_r \times \left(\frac{V}{V_r}\right)^{1{,}4283} \times \left(\frac{D}{D_r}\right)^{1{,}0714} \times \left(\frac{W}{\varnothing}\right)^{0{,}7143} \div \left(\frac{W_r}{\varnothing_r}\right)^{0{,}7143} \tag{3.1}$$

Ezentúl a lövedéknek a megtett távolsága és a találat szöge is figyelembe van véve. Ezek befolyásának a mértékét kettő változóval lehet állítani. A távolságnál az 1-es érték figyelmen kívül hagyja a szorzót, az 1 alatti érték, melyet a legtöbb lövedék használ, nagyobb távolságnál csökkenti az átütés mértékét, míg az 1 feletti érték növeli. A szögteljesítménynél az 0-ás érték nem veszi figyelembe a szög mértékét, az 1-es érték a páncélvastagságnak cos értékét nézi, mely egyenlő az adott szög alapján figyelembe vett vastagságával és a leggyakrabban használt 1 feletti érték gyengíti nagyobb szög esetén. Ha a távolsági és a szögteljesíményi értékeket 1-re állítjuk akkor egy másfajta lövedék típust lehet elérni. Ezekből a változók beállításával lehet további típusokat beállítani.

3.6. kód. Átütés kiszámítása

```
1  protected override float CalculatePenetration(Collision collision, float
       distanceTravelled)
2  {
3      Armor hitArmor = collision.contacts[0].otherCollider.GetComponent<Armor
           >();
4      double rad = Vector3.Angle(transform.forward, -collision.contacts[0].
           normal) * Mathf.Deg2Rad;
5      float effectivePen = remainingPen * (float)Math.Pow(DistanceFalloff,
           distanceTravelled / 1000); //Distance falloff
```

```
6       effectivePen = effectivePen * (float)Math.Pow(Math.Cos(rad),
            AnglePerformance); //Angle falloff
7       return effectivePen - hitArmor.KineticResistance;
8   }
```

A lövedék sebzését mely maximális értéke kaliberétől és a sebességétől függ, és utána az átütött páncél értékéhez és a lövedék átütő erejétől függően csökkentjük.

Robbanótöltetes lövedék

Az alábbi lövedék egy felülethez érvén egy robbanást olyan módon utánoz hogy a megadott átmérőjű gömbön belül először megnézni az összes objektum között hogy bármyelik páncélnak számít-e, majd a robbanás középpontjától számítva mely páncéllemeznek a legkevesebb a vastagsága és a távolságának a keveréke.

3.7. kód. Lövegtorony irányítása

```
1   protected override float GetMaxPenetration()
2   {
3       return (float)Math.Pow(ExplosiveMass, 2 / 3) / 6;
4   }
5   protected override float CalculatePenetration(Collision collision, float
        distanceTravelled)
6   {
7       float armor = GetWeakestArmorThickness(collision);
8       Debug.Log("Armor␣thickness␣hit:␣" + armor);
9       return remainingPen - armor;
10  }
11  float GetWeakestArmorThickness(Collision collision)
12  {
13      Collider[] hitColliders = Physics.OverlapSphere(transform.position, 3f);
14      float weakest = float.MaxValue;
15      for (int i = 0; i < hitColliders.Length; i++)
16      {
17          Debug.DrawRay(transform.position, hitColliders[i].transform.position
                - transform.position, Color.blue, 5f);
18          if (Physics.Raycast(
19              transform.position, hitColliders[i].transform.position -
                    transform.position,
20              out RaycastHit hit, 10f, layerMask: ~(1 << LayerMask.NameToLayer(
                    "CollisionBox"))))
21          {
22              Debug.Log(hit.collider.name + "␣-␣" + hitColliders[i].transform.
                    name);
23              if (hit.collider.transform != hitColliders[i].transform)
24                  continue;
25          }
```

```
26          Armor armor = hitColliders[i].GetComponent<Armor>();
27          if (armor == null) continue;
28          float resistance = GetResistance(hitColliders[i], armor);
29          Debug.Log(armor.name + ":␣" + resistance);
30          if (weakest == 0)
31              weakest = resistance;
32
33          else if (resistance < weakest)
34          {
35              hitArmor = armor;
36              weakest = resistance;
37          }
38      }
39      return weakest;
40  }
41  float GetResistance(Collider collider, Armor armor)
42  {
43      float distance = Vector3.Distance(transform.position, collider.
            ClosestPoint(transform.position));
44      return armor.KineticResistance / MyMath.InvSq(distance);
45  }
```

Robbanótöltetes páncéltörő lövedék

Ez a lövedék hasonlóan működik a sima páncéltörő lövedékhez képest, az átütő ereje ugyanazon a módon van kiszámítva. A különbség a sebzés kiszámításában van, de csak akkor lép életbe, ha a lövedék egy adott vastagsága páncélt átütött, miután a robbanótöltet mértékétől függően sebez.

3.2.6. Kezelőfelület

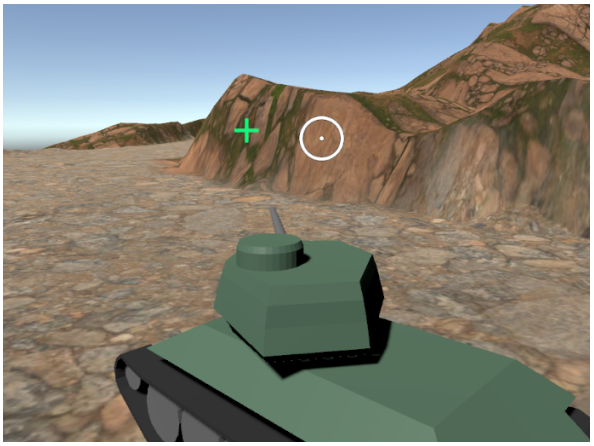
Életerő sáv

Két féle életerő sáv létezik a projektben. Az első típusú a kezelőfelületen fixen található a bal alsó sarokban, mely számokkal kijelzi a teljes és a jelenlegi életerőt, illetve hiányzó életerő mértéke szerint kevesebb része lesz kitöltve a sávnak. A másik típus az ellenfelek felett megtalálható, és a 3D térben helyezkednek el.

3.5. ábra. Életerő sávok

Célkereszt

A célkereszt 2 részből áll. A kör alakú irányzék a képernyő közepét és a kívánt célpontot jelzi, emellett lövés után pirossá változik, jelezve az újratöltés menetét. A második, kereszt alakú, pedig azt jelzi, hogy a cső jelenleg milyen irányba néz.



3.6. ábra. Célkeresztek

3.2.7. Hangok

A játékban csak 2 féle hang van jelenleg implementálva: a motor hang és a lövés hang. A motor hangja folyamatosan ismétlődve játszódik le, és a hangmagassága a sebesség szerint növekszik vagy csökken. A lövési hang természetesen csak lövés esetén hallható. A hangokat a

3.2.8. Ellenfelek

Navmesh

Az ellenfelek útkeresését a Unity-nak a beépített Navmesh rendszere segítségével implementáltam. Bár a Navmesh-be alapból van megadott módszer a mozgásra, de az emberszerű karakterekre van tervezve. Emiatt a sajátos mozgási megoldásra kellett hagyatkozzak. Így ugyanazt a mozgási rendszert használják mint a játékosok.

Ellenőrző pontok

Az ellenfeleknek van egy megadott útvonaluk amit követnek, melyet ellenőrző pontok segítségével oldottam meg. Amikor az adott ellenfél belép egy adott ellenőrző pont érzékelőpont

3.8. kód. Ellenőrző pontok váltása

```
1  void OnTriggerEnter(Collider other)
2  {
3      if (other.transform.position == checkpoints[checkpointNumber].position)
4      {
5          checkpointNumber++;
6          if (stateMachine.currentState == AIStateID.Patrol)
7              agent.SetDestination(checkpoints[checkpointNumber].position);
8          if (isFinished) gameState.LoseGame();
9      }
10 }
```

Állapotgép

Állapotgép használatával van megoldva az ellenfelek viselkedésének szétválasztása. Azért választottam állapotgépet ehhez, mert egy tiszta és egyszerűen bővíthető módszert ad különböző viselkedések és állapotok megadására.

3.9. kód. Lövegtorony irányítása

```
1  public class AIStateMachine
2  {
3      public AIState[] states;
4      public AIStateID currentState;
5      AITank tank;
6      public AIStateMachine(AITank tank)
7      {
8          this.tank = tank;
9          states = new AIState[System.Enum.GetValues(typeof(AIStateID)).Length
               ];
10     }
```

```
11      public void RegisterState(AIState state)
12      {
13          int i = (int)state.GetID();
14          states[i] = state;
15      }
16      public AIState GetState(AIStateID id)
17      {
18          return states[(int)id];
19      }
20      public void Update()
21      {
22          GetState(currentState)?.Update(tank);
23      }
24      public void ChangeState(AIStateID id)
25      {
26          GetState(currentState)?.Exit();
27          currentState = id;
28          Debug.Log("Changing␣state␣to:␣" + id);
29          GetState(currentState)?.Enter(tank);
30      }
31  }
```

Állapotok

Kétféle állapot van:

- Járőrözés
- Támadás

Járőrözés

A járőrözés az alap állapot, melyben az ellenőrző pontok által megadott útvonalat követi az ellenfél.

Támadás

Erre az állapotra akkor vált az az ellenfél, ha az ellenfél előtt elmegyünk és akadály nélkül meglát. Átváltás után amíg nem a játékos megyünk akadály mögé addig az ellenfél a testét és a tornyát forgatja a játékos felé. Miután sikeresen célba vette a játékost az ellenfél lő. Ha a játékos kimegy az ellenfél látószögéből akkor az ellenfél megpróbálja utolérni a játékost. Ha adott időn belül nem látja meg újra a játékost, akkor az ellenfél visszatér járőrözési állapotba.

3.10. kód. Állapotgép

```
1  public bool CheckPlayerVisible()
2  {
3      if (Vector3.Distance(transform.position, player.position) > 100)
4          return false;
5      Vector3 relativeVector = transform.InverseTransformPoint(player.position
           );
6      float angle = Mathf.Atan2(relativeVector.x, relativeVector.z) * Mathf.
           Rad2Deg;
7      if (Mathf.Abs(angle) > 60) return false;
8      if (!Physics.Raycast(transform.position + transform.forward * 6, player.
           position - transform.position, out RaycastHit hit, 100,
           excludeCollisionBox)) return false;
9      return Vector3.Distance(hit.point, player.transform.position) > 3;
10 }
```

3.3. További tervek

A projektnek az alapját és a fő funkcióit sikerült eddigiekben kifejleszenem. Ezentúl szeretném ezt továbbfejleszteni, és kiegészíteni.

Tartalom

A projektem alapja miatt jelenleg készen áll több tartalom fogadására, mely alatt főleg nagyobb harckocsi alkatrésztár és újabb pályák értendőek.

Funkciók

Funkciók közül több ötletem is támadt a fejlesztés közben amikkel szeretném kiegészíteni a továbbiakban. Amit a legfontosabbnak tartok az a billentyűk megváltoztatásának lehetősége lenne, a testreszabhatóság érdekében, és a kontrolleres bevitel támogatása. Emellett további tervekben többféle ellenfél viselkedést és játékmódot implementálni, valamint animációkat és esetleges fizikát a lánctalpakhoz.

4. fejezet

Tesztelés

Teszteléshez manuális tesztelést alkalmaztam.

Összegzés

Irodalomjegyzék

[1] Unity *Unity Documentation*
https://docs.unity.com/

[2] The Tank Archives *Penetration Equations*
https://www.tankarchives.ca/2014/10/penetration-equations.html

[3] Freesound *Freesound*
https://freesound.org/

[4] Poly Haven *The Public 3D Asset Library*
https://polyhaven.com/