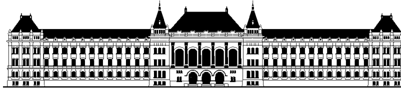


SZAKDOLGOZAT FELADAT

Smodics Roland Szilveszter

mérnök-informatikus hallgató részére

Mesterséges intelligencia fejlesztése csapatjátékokhoz

A számítógépes játékok között nagy népszerűségnek örvendenek a különféle – valós vagy kitalált – csapatsportokon alapuló játékok. Az ilyen jellegű játékok mesterséges intelligenciája nemtriviális feladat, az egyéni viselkedésen kívül a csapat közös célja elérése érdekében csoportos viselkedés megvalósítására is szükség van, továbbá az egyes posztok implementációja mind egyedi megoldásokat igényelhet.

A szakdolgozat feladata egy korábbi, az önálló laboratórium tárgy keretein belül az Unreal játékmotor segítségével implementált – gépi játékos nélküli – Quidditch csapatjáték továbbfejlesztése, elsősorban a mesterséges intelligencia megvalósításával. A játékban több, teljesen különböző viselkedést igénylő poszt is szerepel, melyek között akadnak együttműködést igénylő, illetve teljesen egyéni, a többi játékost figyelmen kívül hagyó szerepek is. A feladatnak része, hogy az egyes szerepeknek leginkább megfelelő mesterséges intelligencia készüljön el, amely akár egymástól független, különböző viselkedések és stratégiák megvalósítását is jelentheti.

A szakdolgozat feladatai a következők:

- Tekintse át és hasonlítsa össze a csoportos viselkedést szimuláló mesterséges intelligencia megvalósításának módszereit, különös tekintettel a sportjátékokban elterjedt módszerekre.
- Tervezze meg a gépi játékos mesterséges intelligenciáját. A lényegesen eltérő posztokra használjon különböző, az adott posztra leginkább megfelelő módszert.
- Implementálja a mesterséges intelligenciával kiegészített játékot az Unreal játékmotor segítségével.
- Értékelje az elkészült játékot játékélmény, valamint a gépi játékos kiszámíthatósága, változatossága és nehézsége szempontjából.

Beadási határidő: 2021. december 10.

Tanszéki konzulens: Dr. Magdics Milán, egyetemi docens

Budapest, 2021. október 5.

Dr. Kiss Bálint

egyetemi docens
tanszékvezető

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Irányítástechnika és Informatika Tanszék

Smodics Roland Szilveszter

MESTERSÉGES INTELLIGENCIA FEJLESZTÉSE CSAPATJÁTÉKOKHOZ

Quidditch Game

KONZULENS

Dr. Magdics Milán

BUDAPEST, 2021

Tartalomjegyzék

Összefoglaló ... **6**

Abstract ... **7**

1 Bevezetés ... **8**

1.1 Quidditch Játék ... 8

1.1.1 Bemutatás ... 8

1.1.2 Szabályok ... 8

1.1.3 Motiváció ... 9

1.2 Unreal Engine Játékmotor ... 9

1.2.1 Bemutatás ... 9

1.2.2 Unreal 4 ... 9

1.2.3 Blueprint rendszer ... 10

1.2.4 Blueprint Fájl Felépítése ... 10

2 MI Videójátékokban ... **13**

2.1 Bevezetés ... 13

2.2 Mesterséges intelligencia játékokban ... 14

2.3 Állapotgépek ... 15

2.4 Viselkedésfák ... 16

3 Megvalósítás ... **18**

3.1 Karakterek ... 18

3.1.1 Mesh ... 18

3.1.2 Animáció ... 19

3.1.3 Character Blueprints ... 20

3.2 AI Controller ... 24

3.2.1 Bemutatás ... 24

3.2.2 Chaser ... 24

3.2.3 Seeker ... 27

3.3 Labdák ... 29

3.3.1 Bludger ... 29

3.3.2 Golden Snitch ... 30

3.3.3 Quaffle ... 32

3.4 Egyéb Komponensek ... 33
3.4.1 Landscape ... 33
3.4.2 Field ... 33
3.4.3 Minimap ... 36
3.4.4 Pontrendszer ... 37
3.4.5 Audio ... 37
4 Értékelés ... 38
4.1 Játékélmény ... 38
4.2 Mesterséges intelligencia minősége ... 38
4.2.1 Cikesz ... 38
4.2.2 Gurkók ... 38
4.2.3 Fogó ... 39
4.2.4 Hajtók ... 39
5 Összefoglalás ... 40
5.1 Quidditch Game ... 40
5.2 Fejlesztési lehetőségek ... 40
Irodalomjegyzék ... 41

HALLGATÓI NYILATKOZAT

Alulírott **Smodics Roland**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2021. 12. 09.

...…………………………………………….

Smodics Roland

Összefoglaló

Az elmúlt évek alatt a számítógépes játékok közül egyre inkább kiemelkednek a különféle – valós vagy kitalált – csapatsportokon alapuló játékok. Az ilyen játékokban általában több különböző szereppel találkozhatunk, melyeknek mind saját, önálló gondolkodásuk van, amikkel az alapvető emberi reakciókat, illetve viselkedést, próbálják imitálni. A különböző szerepek mesterséges intelligenciájának megvalósítása nemtriviális feladat, az egyéni viselkedésen kívül a csapat közös célja elérése érdekében csoportos viselkedés megvalósítására is szükség van, továbbá az egyes posztok implementációja mind egyedi megoldásokat igényelhet.

A szakdolgozat a korábbi, Önálló Laboratórium tárgy keretein belül elkezdett Quidditch játék implementációjának továbbfejlesztésén és bemutatásán alapszik, elsősorban a mesterséges intelligencia megvalósításával. A dolgozat, a játékban szereplő különböző viselkedést igénylő szereplők megvalósítását is bemutatja, melyek között akadnak együttműködést igénylő, illetve teljesen egyéni viselkedést követő szerepek is. A csapatjátékokban a mesterséges intelligencia megvalósítása komplex feladat, hiszen nem csak a külső hatásokhoz kell alkalmazkodnia, hanem a többi – szintén mesterséges intelligencia által vezérelt – szereplőhöz is, ami alapján egy megfelelő stratégiát tudnak összhangban követni. A feladathoz hozzátartozik, a megfelelő kutatómunka, illetve a témakör és a játékmotor megismerése is.

Abstract

In recent years, computer-based games, whether real or fictional, have become increasingly prominent among computer games. In such games, we usually encounter several different roles, all of which have their own independent thinking, with which they try to imitate basic human reactions and behavior. Implementing artificial intelligence in different roles is a non-trivial task, in addition to individual behavior, group behavior is required to achieve a common goal for the team, and the implementation of each position may all require individual solutions.

The dissertation is based on the further development and presentation of the implementation of the Quidditch game, which started within the framework of the previous class „Önálló Laboratórium", primarily with the implementation of artificial intelligence. The dissertation also presents the implementation of the characters in the game who require different behaviors, including roles that require cooperation and following completely individual behaviors. Implementing artificial intelligence in team games is a complex task, as it has to adapt not only to external influences, but also to other actors, also driven by artificial intelligence, so that they can follow an appropriate strategy in harmony. The task also includes getting to know the appropriate research work, as well as the topic and the game engine.

1 Bevezetés

A szakdolgozat feladataként egy Quidditch játékot hoztam létre, amely megvalósít négy különböző mesterséges intelligenciával vezérelt szereplőt. A feladathoz tartozik a játék teljes implementációja, beleértve a karaktereket, az animációkat, illetve a különböző játékélményt növelő komponenseket. A munkához hozzátartozik a mesterséges intelligencia témájú kutatómunka és a tudományág megismerése, illetve az Unreal Engine játékmotor megismerése.

1.1 Quidditch Játék

1.1.1 Bemutatás

A Kviddics (*Quidditch*) egy kitalált sport, amely J.K. Rowling Harry Potter című könyv- és filmsorozatában mutatkozott be. Népszerű sport, amelyet repülő seprűn lovagló boszorkányok és varázslók űznek.

A mérkőzéseket egy nagy, ellipszis alakú pályán, két csapattal játsszák, mindkét oldalon három különböző magasságú póznával, amely tetején találhatóak a gólkarikák. Hét emberből áll egy csapat: 3 Hajtó (*Chaser*), 2 Terelő (*Beater*), 1 Őrző (*Keeper*) és 1 Fogó (*Seeker*).

1.1.2 Szabályok

A játékot három különböző (összesen négy) labdával játsszák. A legnagyobb a *Kvaff* (*Quaffle*), egy nagy piros, három helyen homorú labda, amit a három hajtó passzolgat egymás között, és megpróbálják átdobni az ellenfél pálya végében felállított három póznájának egyikén. Ha ez sikerül, a csapat 10 ponttal gazdagodik. A második labda a *Gurkó* (*Bludger*), amiből két darab van a pályán. Mindkettő a pályán cikázva próbálja lelökni a játékosokat a seprűjükről. A két terelő feladata a csapattagjainak védelme és az ellenfél csapatának támadása az ütőiknek a segítségével, amivel a gurkókat terelhetik. A harmadik, legkisebb labda az *Aranycikesz* (*Golden Snitch*). A cikesz olyan gyorsan cikázik a pályán, hogy alig lehet észrevenni, és a fogó feladata, hogy elkapja. Ha a cikeszt elkapta valamelyik fogó, akkor az a csapat 150 pontot kap, és a meccs véget ér.

1.1.3 Motiváció

A Harry Potter világa nagyon elterjedt és népszerű a mai generációs fiatalok körében, ennek ellenére alig néhány egyszerű Quidditch játék program létezik, amik ráadásul nem is adják át igazán az eredeti sporthoz hasonló játékélményt. Személy szerint engem is megfogott ez a világ, így egy saját Quidditch szimulációs játék mellett döntöttem. A játékfejlesztés már régebben is érdekelt, és majdnem minden egyetemi beadandót, illetve házi feladatot valamilyen játék formájában adtam le (sakk, aknakereső, honfoglaló, snake game stb). Témalabor keretében az SZFE hallgatóival közösen készítettünk három játék projektet, Önálló labor keretein belül pedig szintén játékfejlesztés témában dolgoztam. A mesterséges intelligencia fejlesztése szintén egy nagyon érdekes témakör, amit meg akartam ismerni és egy kicsit mélyebben beleásni magam, illetve a jövőben is szeretnék ilyen területen játékfejlesztőként elhelyezkedni, így adta magát a lehetőség, hogy a szakdolgozat keretein belül hasonló témával foglalkozzak.

1.2 Unreal Engine Játékmotor

1.2.1 Bemutatás

Az Unreal Engine az 1990-es évek végén bemutatott, Epic Games által kifejlesztett játékmotor. A C++ nyelven írt Unreal Engine nagyfokú hordozhatóságot biztosít, és támogatja az asztali, mobil-, konzol- és virtuális valóság platformok széles skáláját. Népszerű a nagy grafikus képességekkel rendelkező PC-s és konzolos játékok között, és számos játékhoz használják, valamint további felhasználásra talál filmkészítésben és egyéb üzleti alkalmazásokban.

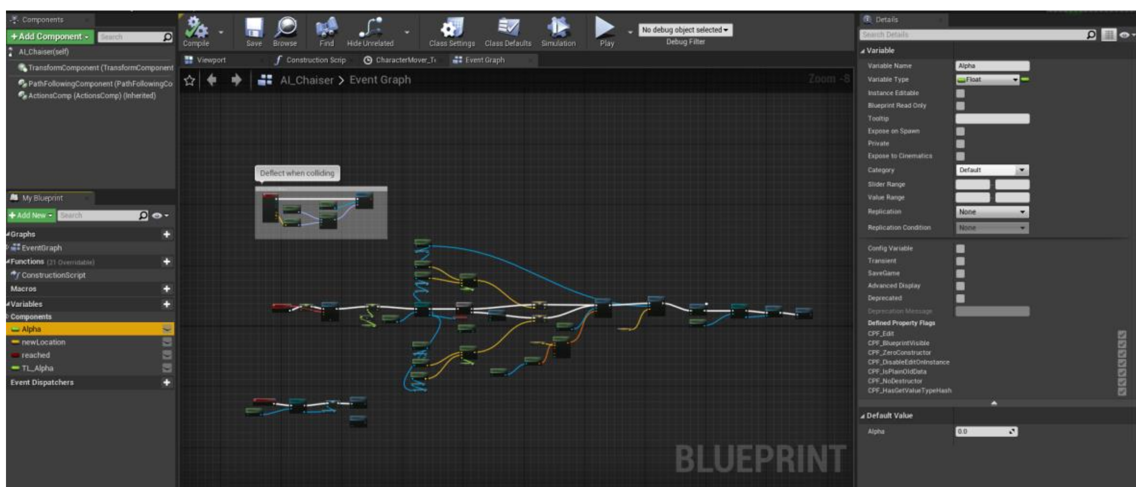
1.2.2 Unreal 4

A legújabb generáció az Unreal Engine 4, amely 2014-ben jelent meg előfizetéses modellben. Az Unreal Engine 4 előtt a játékmotor egy saját, Unreal Script nevű szriptnyelvet használt, amely a Java-hoz hasonlóan objektum-orientált volt és az osztályokat az általuk meghatározott osztályhoz elnevezett egyedi fájlokban határozták meg. 2012-ben a Game Developers konferencián az Epic bejelentette, hogy az Unreal Scriptet eltávolítják az Unreal Engine 4-ből a C++ javára. A vizuális szkripteket a

Blueprints Visual Scripting rendszer támogatná, amely a korábbi Kismet vizuális szkriptrendszert helyettesíti.

1.2.3 Blueprint rendszer

Az Unreal Engine Blueprint Visual Scripting rendszere egy teljes játékmenet-szkriptrendszer, amely azon az elgondoláson alapul, hogy csomópont-alapú felületet használnak játékelemek létrehozására az Unreal Editorból (1-1. ábra). Mint sok elterjedt szkriptnyelv, ez is objektumorientált (OO) osztályok vagy objektumok meghatározására szolgál a motorban. A Blueprint osztály olyan eszköz, amely lehetővé teszi, hogy a meglévő játékosztályok mellé funkcionalitást is könnyedén lehessen kapcsolni. Az osztályok leírása egy külön Unreal Editor ablakban vizuálisan szerkeszthető C++ kód írás helyett, majd Blueprint fájlként mentve a játéktérhez adhatjuk az elkészült objektumot. (Természetesen a játékmotor lehetőséget ad a hagyományos C++ szkriptek írásához is.)



1-1. ábra Blueprint szerkesztőfelület

1.2.4 Blueprint Fájl Felépítése

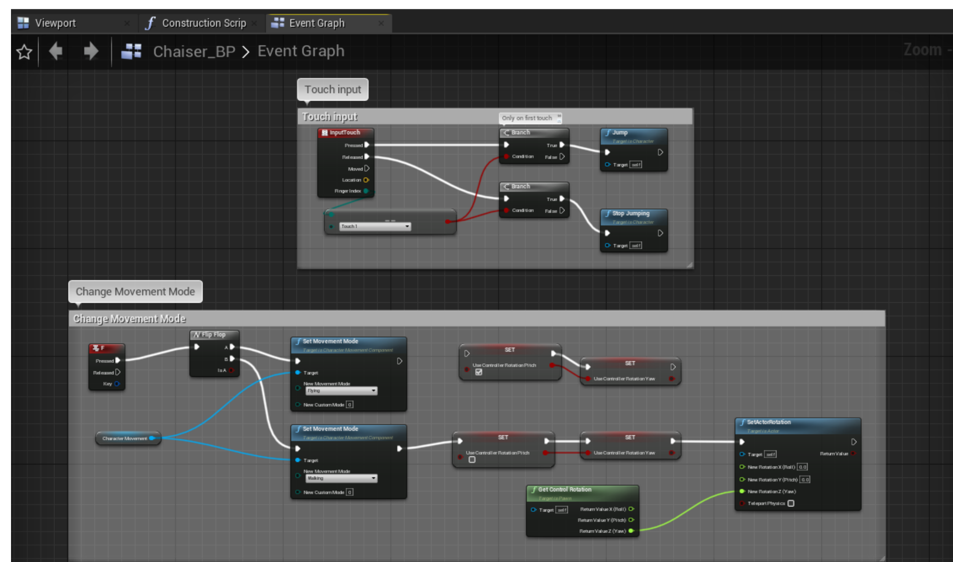
Amikor létrehozunk egy Blueprint fájlt, több különböző típus és sablon áll rendelkezésünkre. A leggyakoribbak az *Actor*, a *Pawn*, a *Character* és a *Player Controller*.

- Actor: Az *Actor* blueprint egy olyan objektum, amit megjeleníthetünk, elhelyezhetünk a játéktéren és adhatunk neki viselkedést, fizikai testet, és anyagot.
- Pawn: A P*awn* egy olyan Actor ami fölött a játékos átveheti az irányítást, vagy egy kontroller alapján irányítható.

- Character: A Character egy olyan Pawn, ami képes arra, hogy a világban sétáljon/mozogjon.
- Player Controller: A *Player Controller* egy olyan Actor, ami egy meghatározott Pawn irányításáért felel.

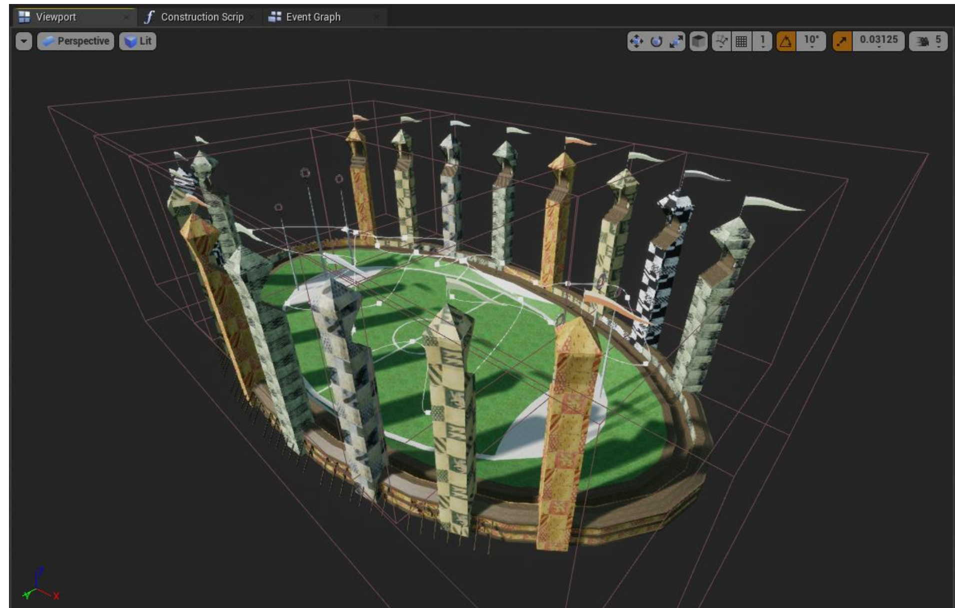
A Blueprintet megnyitva számos különböző ablak és fül áll rendelkezésre, melyek közül a legfontosabbak az *Event Graph*, a *Viewport*, a *Components*, a *Details* és a *My Blueprint.*

- Event Graph: A Blueprint szerkesztőfelülete (1-2. ábra). Itt lehet az objektum viselkedését leírni a vizuális szkriptrendszerre. Egy adott csomópont (node) rendelkezik bemeneti és kimeneti paraméterekkel, amik által egy hagyományos C++ kódhoz hasonlóan lehet függvényeket hívni vagy akár eventeket létrehozni.



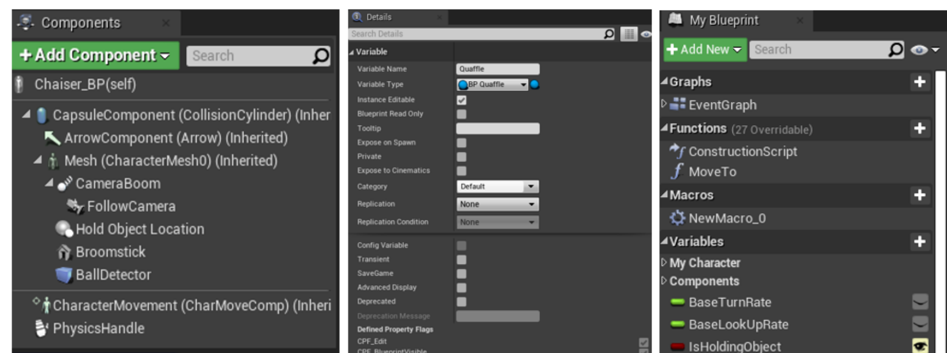
1-2. ábra - Event Graph

- Viewport: A *Viewport* felületen lehet az adott Blueprint osztály megjelenítési beállításait kezelni, pl. az adott komponensek elhelyezését vagy a kamera nézetét (1-3. ábra).



1-3. ábra - Viewport

- Components, Details, My Blueprint: Ezeken az ablakokon az osztályhoz szorosan tartozó vagy kapcsolódó elemeket lehet megadni, illetve ezek beállításait kezelni (1-4. ábra). A My Blueprint fülön az adott osztály tagváltozóit, függvényeit és eseményeit lehet létrehozni, illetve ezekhez hozzáférni. A Details panelen ezeknek a részletesebb információit láthatjuk/állíthatjuk, mint például a változó típusa, kezdőértéke vagy akár egyéb fizikai beállításai (gravitáció, ütközés beállítások stb.)



1-4. ábra - Kezelőpanelek

2 MI Videójátékokban

2.1 Bevezetés

Az MI, azaz mesterséges intelligencia (Artificial Intelligence) az állatok vagy emberek által mutatott természetes intelligenciával szemben, a gépek által tanusított intelligencia. Ezt a területet az „intelligens ágensek" tanulmányozásaként határozzák meg: minden olyan rendszer, amely érzékeli környezetét, és olyan lépéseket tesz, amelyek maximalizálják annak esélyét, hogy elérje céljait. Egyes beszámolók a „mesterséges intelligencia" kifejezést használják olyan gépek leírására, amelyek utánozzák azokat a „kognitív" funkciókat, amelyeket az emberek az emberi elmével társítanak, mint például a tanulás és a problémamegoldás, azonban ezt a definíciót a fő MI-kutatók elutasítják[3].

A mai világban egyre elterjedtebbek az MI-alkalmazások, főként a különböző webes keresőmotorok (pl. Google), az ajánlórendszerek (a YouTube, az Amazon és a Netflix), az emberi beszéd megértésére specifikálódott szoftverek (például a Siri és az Alexa), az önvezető autók (pl. Tesla), és a legmagasabb szinten lévő stratégiai játékrendszerek (mint például a sakk és a Go). Ahogy a gépek egyre alkalmasabbá válnak, az „intelligenciát" igénylő feladatok gyakran kikerülnek az MI definíciójából. Például az optikai karakterfelismerést gyakran kizárják a mesterséges intelligenciának tekintett dolgokból, mivel rutintechnológiává vált. A mesterséges intelligenciát 1956-ban kezdték különálló tudományágként besorolni. Az MI-kutatás számos különféle megközelítést kipróbált és elvetett megalapítása óta, beleértve az agy szimulációját, az emberi problémamegoldás modellezését, a formális logikát, a tudás nagy adatbázisait és az állatok viselkedésének imitálását.

Az MI-kutatás különböző részterületei meghatározott célok és eszközök felhasználása köré összpontosulnak. Az kutatás hagyományos céljai közé tartozik az érvelés, a tudásreprezentáció, a tervezés, a tanulás, a természetes nyelvi feldolgozás, az észlelés, valamint a tárgyak mozgatásának és manipulálásának képessége. Az általános intelligencia (egy tetszőleges probléma megoldásának képessége) a terület egyik hosszú távú célja. Az ilyen típusú problémák megoldására a mesterséges intelligencia kutatói problémamegoldó technikák széles skáláját adaptálták és integrálták, beleértve a keresési és matematikai optimalizációt, a formális logikát, a mesterséges neurális hálókat, valamint a statisztikákon, valószínűségszámításon és közgazdaságtanon alapuló

módszereket. A mesterséges intelligencia emellett számítástechnikára, pszichológiára, nyelvészetre, filozófiára és sok más területre is támaszkodik[3].

2.2 Mesterséges intelligencia játékokban

Amikor egy videojátékban az emberi gondolkodásmódot próbálják utánozni egy nem a felhasználó által vezérelt karakterekben (NPC-n), a reagáló, adaptív vagy intelligens viselkedések generálására mesterséges intelligenciát használnak. A mesterséges intelligencia az 1950-es évek kezdete óta a videojátékok szerves részét képezi. A videojátékokban a mesterséges intelligencia azonban eltér az akadémiai MI-től és egy külön részterületet képez. A gépi tanulás vagy a döntéshozatal helyett inkább a játékélmény javítását szolgálja. A modern játékok gyakran az útkereséssel és döntési fákkal valósítják meg az NPC-k tevékenységeit.

A „*Game AI*” kifejezést az algoritmusok széles skálájára használják, amelyek magukban foglalják a vezérléselméletből, a robotikából, a számítógépes grafikából és általában a számítástechnikából származó technikákat is, így a videojátékok mesterséges intelligencia gyakran nem minősül „igazi mesterséges intelligenciának”. A technikák nem feltétlenül segítik elő a számítógépes tanulást vagy más szabványos kritériumokat, csupán automatikus reakciókat égetnek bele a programba, így egy előre meghatározott és korlátozott bemeneti halmazra, válaszként szintén egy előre meghatározott és korlátozott válaszkészlet áll rendelkezésre.

Számos iparág és vállalati hang azt állítja, hogy az úgynevezett videojátékos mesterséges intelligencia hosszú utat tett meg abban az értelemben, hogy forradalmasította az emberek és a technológia minden formájával való interakcióját, bár sokan szakértő kutatók szkeptikusak az ilyen állításokkal kapcsolatban, és különösen azzal kapcsolatban, hogy az ilyen technológiák megfelelnek a kognitív tudományokban szokásosan használt „intelligencia” definíciójának.

A mesterséges intelligencia területén az emberek azonban azzal érveltek, hogy a videojátékok mesterséges intelligenciája nem valódi intelligencia, hanem egy reklámszó, amelyet olyan számítógépes programok leírására használnak, amelyek egyszerű rendezési és egyeztetési algoritmusokat használnak az intelligens viselkedés illúziójának megteremtésére[1].

A játék algoritmusait a játékon belül számos, egymástól meglehetősen eltérő területen használják. A legnyilvánvalóbb az NPC-k vezérlése a játékban, bár jelenleg a viselkedésfa (2.4) a legáltalánosabb vezérlési eszköz. Ezek a fák gyakran „mesterséges butaságot” eredményeznek, például ismétlődő viselkedést vagy abnormális viselkedést olyan helyzetekben, amelyeket a fejlesztők nem terveztek.

Az útkeresés, az MI másik gyakori felhasználási módja, ami széles körben látható a valós idejű stratégiai játékokban. Az útkeresés az a módszer, amellyel meghatározható, hogyan juttassunk el egy NPC-t a térkép egyik pontjáról a másikra, figyelembe véve a terepet és akadályokat. A videojátékok gyakran használnak gyors és egyszerű „rács alapú útkeresést”, ahol a terepet egy merev, egyenletes négyzetekből álló rácsra képezik le, és egy útkereső algoritmust, alkalmaznak a rácsra. Egyes játékok a merev rács helyett szabálytalan sokszögeket használnak, és navigációs hálót állítanak össze a térkép azon területeiből, ahová az NPC-k elsétálhatnak. Harmadik módszerként a fejlesztők számára néha kényelmes, ha manuálisan választják ki azokat a navigációs útpontokat, amelyeket az NPC-knek használniuk kell a navigáláshoz. Az ilyen útpontok viszont néha természetellenes mozgást hozhatnak létre. A statikus útkeresésen túl a navigáció a *Game AI* egy részterülete, amely arra összpontosít, hogy az NPC-k dinamikus környezetben navigálhassanak, megtalálják a célhoz vezető utat, miközben elkerülik a más objektumokkal való ütközést.

2.3 Állapotgépek

Az állapotgép egy olyan absztrakt gép, amely egy adott pillanatban véges számú állapotból pontosan egyben lehet. A különböző bemenetekre egyik állapotból a másikba válthat, amit átmenetnek nevezünk. Az állapotgépek meghatározásához ismernünk kell az állapotait, a kezdeti állapotát, illetve az átmeneteket kiváltó bemeneteit. Két típusú állapotgépet különböztetünk meg: determinisztikus, vagy nem determinisztikus. Az állapotgépek viselkedése a modern társadalom számos eszközén megfigyelhető, amelyek előre meghatározott cselekvési sorozatot hajtanak végre attól függően, hogy milyen eseménysorral mutatkoznak be. Egyszerű példák erre a különféle automaták, a liftek, a közlekedési lámpák, vagy a kombinációs zárak.

A véges állapotgép kisebb számítási teljesítménnyel rendelkezik, mint néhány más számítási modell, például a Turing-gép. A számítási teljesítmény megkülönböztetés azt jelenti, hogy vannak olyan számítási feladatok, amelyeket egy Turing-gép képes

elvégezni, de egy állapotgép nem, hiszen egy állapotgép „memóriája” korlátozva van az állapotainak száma által. Az állapotgépeket az automataelmélet általánosabb területén tanulmányozzák.

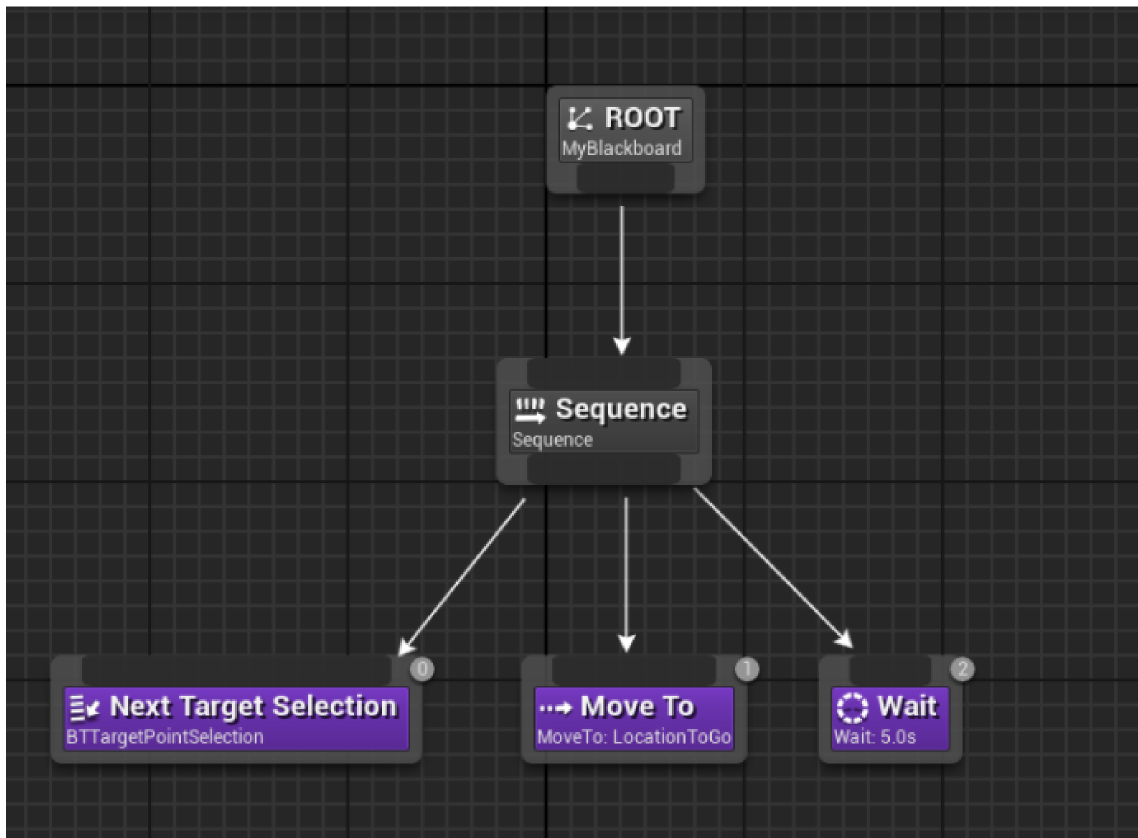
2.4 Viselkedésfák

A viselkedésfa (*Behavior Tree*), a vezérelt objektum lépései végrehajtásának matematikai modellje, amit számítástechnikában, robotikában és videojátékokban használnak. A viselkedésfák matematikailag irányított körmentes gráfok, amelyek modulárisan írják le a feladatok véges halmaza közötti váltást. Erősségük abban rejlik, hogy képesek nagyon összetett, egyszerű feladatokból álló feladat-sorozatokat létrehozni anélkül, hogy az egyes feladatok végrehajtásával foglalkozniuk kellene. Bizonyos szinten hasonlítanak a hierarchikus állapotgépekre, azzal a fő különbséggel, hogy a viselkedés alapja egy feladat, nem pedig egy állapot.

A viselkedésfa a számítógépes játékiparból származik, és hatékony eszköznek bizonyult az NPC karakterek viselkedésének modellezésére. A viselkedésfák használatával összetett viselkedést lehet létrehozni úgy, hogy csak az NPC műveleteit programozzák. Ezek után olyan fastruktúrát hoznak létre, aminek levélcsomópontjai műveletek, és a belső csomópontjai határozzák meg az NPC döntéshozatalát. A viselkedésfák vizuálisan intuitívak, könnyen megtervezhetők, tesztelhetők, és több modularitást, skálázhatóságot és újra-felhasználhatóságot biztosítanak, mint a többi viselkedés-létrehozási módszer. Az esemény vezérelt viselkedésfák megoldották a klasszikus viselkedésfák skálázhatósági problémáit azáltal, hogy megváltoztatták a fa belső végrehajtási módját, és egy új típusú csomópontot vezettek be, amely képes reagálni az eseményekre és megszakítani a futó csomópontokat. Manapság az esemény vezérelt viselkedésfa fogalma szabvány, és a legtöbb megvalósításban használatos, bár az egyszerűség kedvéért még mindig „viselkedési fának” nevezik őket[5].

A viselkedésfát grafikusan irányított faként ábrázolják, amelyben a csomópontok gyökér-, vezérlés- vagy feladat-csomópontokként (*Taskok*) vannak besorolva (2-1. ábra). Grafikusan egy vezérlés-csomópont gyermekei alatta helyezkednek el, balról jobbra rendezve. A viselkedésfa végrehajtása a gyökértől indul, amely bizonyos gyakorisággal *tick*-et küld gyermekének. A *tick* egy engedélyező jel, amely lehetővé teszi egy leágazott feladat végrehajtását. Ha a viselkedésfában egy feladat végrehajtása befejeződött, akkor visszaadja a szülőnek az irányítást. A vezérlés-csomópont az azt alkotó részfeladatok

vezérlésére szolgál. A vezérlőfolyam-csomópont lehet elágazás (*Selector*), vagy szekvencia (*Seqvence*). Felváltva futtatják az egyes részfeladatokat, és amikor egy részfeladat befejeződik és visszatér a szülőhöz, a vezérlés-csomópont eldönti, hogy végrehajtja-e a következő részfeladatot, vagy sem[4].



2-1. ábra - Behavior Tree[2]

3 Megvalósítás

A játékhoz elengedhetetlen kellékek közé tartoznak az olyan komponensek, amik nélkül önmagában a felhasználó semmit nem érzékelne a játékból. Ilyenek a karakterek, a pálya, a labdák, valamint a pontrendszer. Természetesen ide sorolható a mesterséges intelligenciák által vezérelt szereplők, illetve a hozzájuk tartozó kontrollerek. Ebben a fejezetben a felsoroltak megvalósításáról lesz szó.

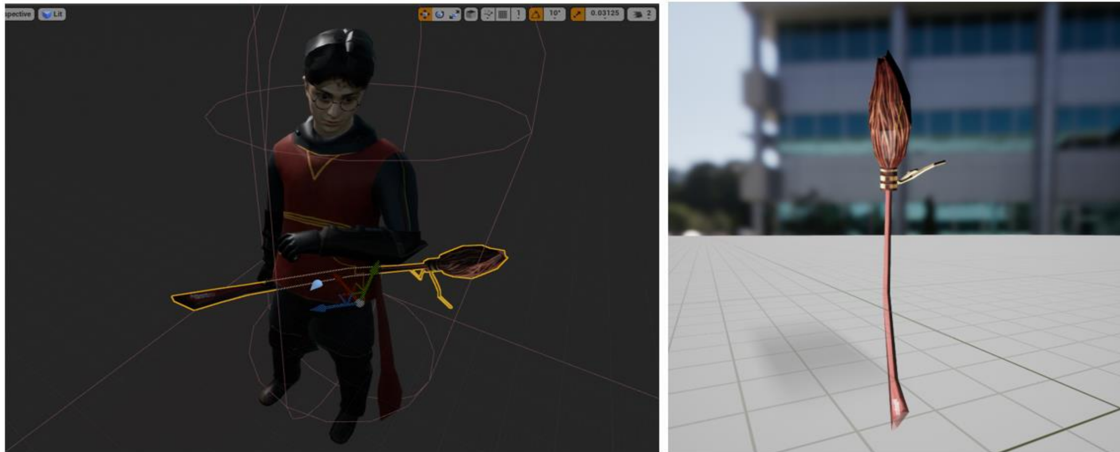
3.1 Karakterek

3.1.1 Mesh

A karakterek fizikai modellje egy letöltött modell, ami Blender modellező program segítségével van animálva a játékhoz[7]. A modell minden rétegéhez egy külön anyag (*material*) van hozzárendelve, így külön a bőr, külön a haj, a ruha és a védőfelszerelés (3-1. ábra). A modell maga is több rétegből áll, ezért lehetőség van külön animálni az egyes részeket (például repülés esetén a ruha lobogásának animálása).

3-1. ábra - Karakter modell

Minden játékosnál van egy seprű, amin repülnek. A seprű egy külön *Static Mesh* ami a karakter kézfejéhez van csatolva egy *Socket*-en keresztül (3-2. ábra).

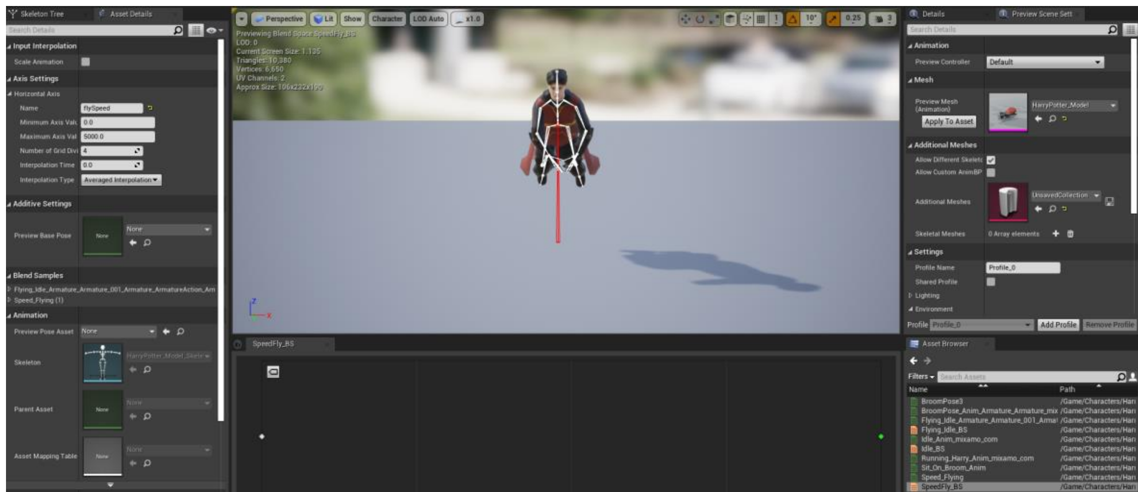


3-2. ábra - Seprű csatolása

3.1.2 Animáció

A repülési animációk megvalósítása a Blender modellező program segítségével történt, és egy egydimenziós BlendSpace fájlban vannak használva. A BlendSpace alapvetően az Unreal Engine egy speciális eszköze, amely lehetővé teszi az animációk keverését két bemeneti érték alapján (3-3. ábra). A két animáció egyetlen bemeneten alapuló egyszerű keverése végrehajtható az Animation Blueprintben elérhető standard csomópontok egyikével. A BlendSpaceben lehetőség van több animáció összetettebb keverésére is több érték alapján, így az x- és y-tengely értékétől függően válthatunk az animációk között.

A használt *SpeedFly_BS* egyetlen bemeneti változót használ az x-tengelyen, aminek a nagysága által változtat az alap repülés és a gyorsított repülés animációja között. A bemeneti változót a BlendSpace-hez tartozó Animation Blueprint fájlban lehet módosítani, illetve bármilyen egyéb viselkedést megadni az animációnak. A játékban a *SpeedFly_BS* bemeneteként átveszi az adott karakter aktuális előre mutató sebességvektorát, így a sebesség függvényében egy nagyon szép átmenet látható az animációban, ahogy minél gyorsabban repül a karakter, annál inkább fekszik rá a seprűre (3-4. ábra).



3-3. ábra - BlendSpace felület

3-4. ábra - Gyorsulás Animáció

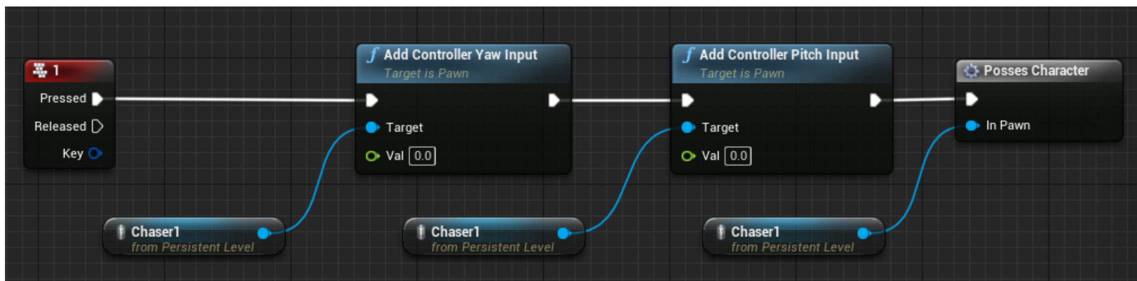
3.1.3 Character Blueprints

Egy karakter alapvető komponenseit és viselkedését a *Character_BP* osztály tartalmazza. Viselkedés alatt jelen esetben az olyan feladatok leírását értjük, ami minden karakter alapvető működéséhez szükséges, mint például a repülés megvalósítása, azon belül a forgás, gyorsítás, fékezés, az ütközés detektálása és lekezelése és a *Quaffle*-lel való interakció. Ezen felül ez az osztály felelős a felhasználói inputok kezeléséért.

Az Unreal Engine külön felületet biztosít a bemeneti események bekötéséhez és a tengelyleképezések kezeléséhez. A bemeneti események a billentyűk lenyomására és felengedésére szolgálnak, míg a tengelyleképezések folyamatos bemeneteket tesznek lehetővé. A játékmotor olyan felületet biztosít, amivel a tengelyeket és a bemeneti eseményeket elkülöníti a viselkedéstől azáltal, hogy egy indirekt réteget iktat be a beviteli

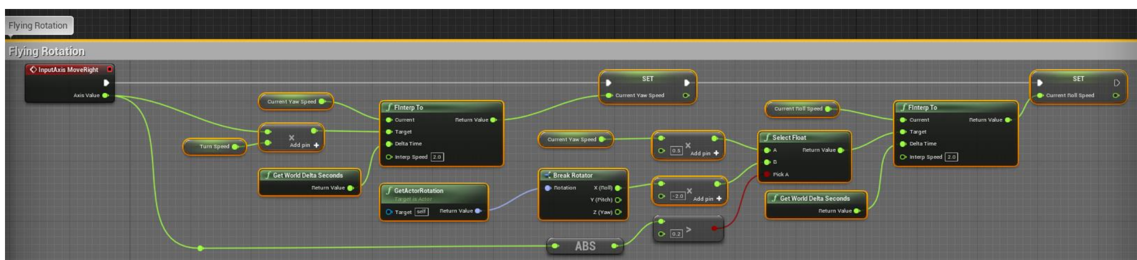
viselkedés és az azt meghívó billentyűk közé. A felület lehetőséget ad a beviteli események konfigurálására függetlenül attól, hogy milyen platformon használják a programot. Így a felhasználó saját magának beállíthatja, hogy melyik inputra milyen tengely vagy esemény hívódjon meg[8].

A karakterek közül egy játékost a felhasználó vezérel, a többit pedig egy-egy *AI Controller*. A játékosok között a felhasználó váltogathat, hogy melyiket szeretné vezérelni, majd amikor átvált egy másik karakterre, az amelyiket eddig irányította folytatja az alapértelmezett *AI Contorller* vezérlőjének követését (3-5. ábra). Azt, hogy melyik játékost irányítja a felhasználó, egy *Bool* változóval (*possessed*) figyeljük, és az *Event Possessed* eseménnyel állítjuk.

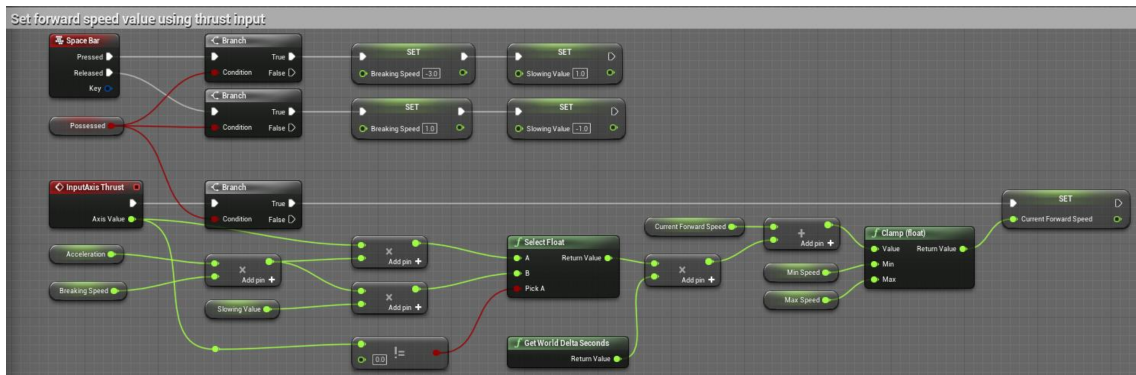


3-5. ábra - Karakter váltás

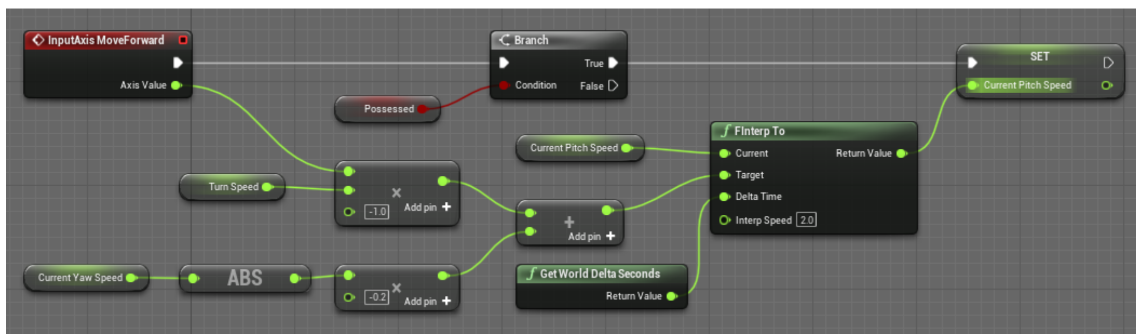
A játékos mozgásához és forgatásához a bemeneti tengelyek közül a *Thrust, Move Forward* és *Move Right* eseményeket használjuk. A projekt beállításainál az adott tengelyeknek pozitív, illetve negatív értékeket megadva lehet beállítani a tengely irányát, és így vezérlést adni a tengelyekhez kötött viselkedéshez az adott konzolon. Mind a három tengely esetében, az osztály saját változóinak (*Current Forward Speed, Current Roll Speed, Current Yaw Speed, Current Pitch Speed*) értékei kerülnek kiszámításra (3-6. ábra, 3-7. ábra, 3-8. ábra) és beállításra az alábbi ábrákon látható módon:



3-6. ábra - Repülési forgás kiszámítása

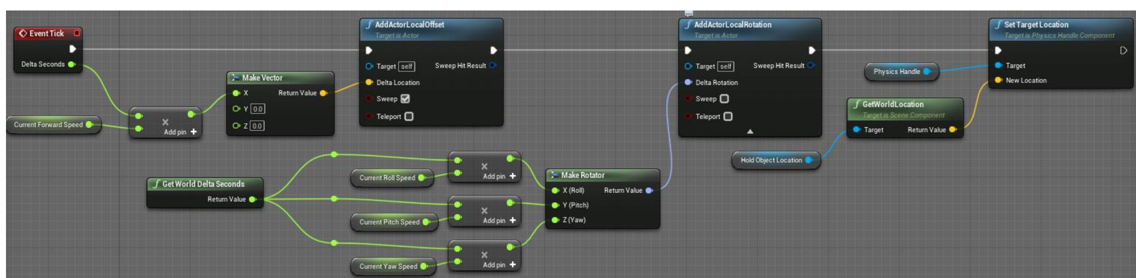


3-7. ábra - Repülési sebesség beállítása



3-8. ábra - X-tengely irányú forgás kiszámítása

Miután a megfelelő forgási és gyorsulási sebességek kiszámítása elkészült a bemenetek alapján, minden időpillanatban módosul a karakter aktuális helyzete és forgása (3-9. ábra).

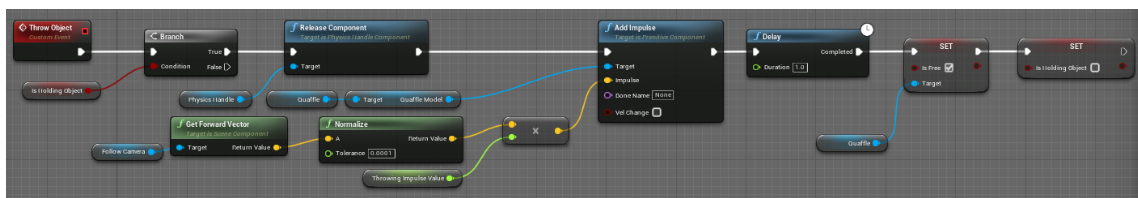


3-9. ábra - Karakter mozgatása sebességek alapján

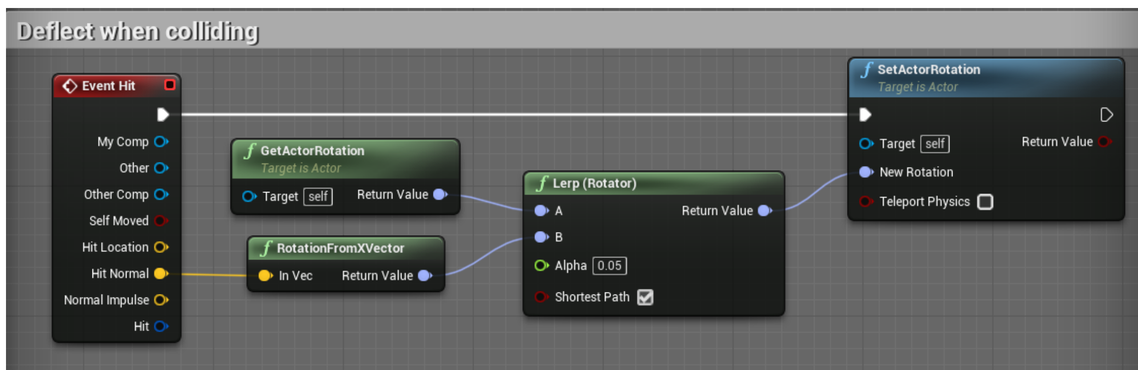
A *Quaffle* eldobása minden játékosnak alapvető képessége, így ez is a *Character_BP* osztály tagfüggvénye kell, hogy legyen. A *Quaffle*-t a hajtók egy *Physics Handle* típusú komponenshez hozzácsatolva tudják „birtokolni”, és a *Hold Object Location* (*Scene Component*) pozícióban tartják. A *Physics Handle* komponens fizikai objektumok megfogására és mozgatására alkalmas komponens, miközben lehetővé teszi a megragadott objektum számára, hogy továbbra is használja a fizikát[9]. A labda eldobásakor a *Physics Handle* komponensből kivesszük a *Quaffle*-t, majd egy bizonyos

nagyságú (kívülről állítható) impulzust adunk neki, így a labda elrepül. A dobás iránya *AI_Conroller* vezérlésű játékos esetén mindig a játékos előre mutató vektora, felhasználó által vezérelt játékos esetén pedig a kamera forgatásával a felhasználó döntheti el a dobás irányát (3-10. ábra).

A karakterek ütközésének detektálása és lekezelése szintén a *Character_BP* osztályban van implementálva (3-11. ábra). Ütközés esetén, a felület (ütközési pontjában vett) normálvektorának irányába egy Lerp függvény segítségével elfordul a karakter. Ez minden Event Hit eseményhíváskor megismétlődik, így egy folytonos lefordulás történik a felületről.



3-10. ábra - Quaffle eldobása



3-11. ábra - Ütközések kezelése

3.2 AI Controller

3.2.1 Bemutatás

Míg a *Player Controller* egy felhasználóra támaszkodik a viselkedéssel kapcsolatos döntések meghozatalában, egy *AI Controller* inkább a környezet és a játékvilág bemeneteire koncentrál. Az *AI Controller* feladata, hogy megfigyelje a körülötte lévő világot, és döntéseket hozzon, és ennek megfelelően reagáljon anélkül, hogy egy emberi játékos bármilyen módon beleavatkozna a döntések meghozatalába. Bármely *Pawn* típusú objektum rendelkezik egy *AI Controller* komponenssel, ami az adott objektumot vezérli, ha azt jelenleg nem ember vezérli[10].

A játékban lehetőség van a három hajtó között játékidőben váltani. Az éppen aktuálisan vezérelt játékost ilyenkor az alapértelmezett *Character_BP* vezérli, ami így *Player Controllerként* működik, ha viszont megvonjuk tőle a vezérlést és átadjuk egy másik játékosnak, akkor életbe lép az *AI Controller* és onnantól kezdve az alapján viselkedik a karakter.

3.2.2 Chaser

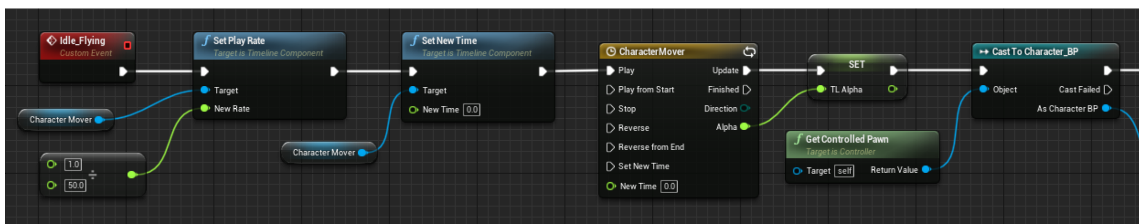
A hajtók mesterséges intelligenciájának megvalósítása bonyolult feladat, hiszen nem elég a saját viselkedésével foglalkoznia a karakternek, hanem figyelnie kell a többi játékos (szintén *AI Controller* által vezérelt) viselkedésére is, a megfelelő stratégia kialakításának céljából.

Minden hajtó alaphelyzetben egy *Idle_Flying* módban van, majd (a viselkedésfákhoz hasonlóan) a különböző eseményekre ebből az állapotból vált át, és kezdi el a megfelelő feladatok végrehajtását, amikkel, ha végzett, visszatér az *Idle_Flying* módba (3-16. ábra). A feladatok közti váltást minden döntési változóra minden pillanatban ellenőrizni kell. Ezért, és a karakter mozgatásáért, egy külön Timeline típusú objektum felel (3-13. ábra).

Az idővonalak (*Timeline*) olyan speciális komponensek a Blueprinteken belül, amik idő alapú animációt biztosítanak, *float* változók, vektorok vagy színek alapján, amelyek az idővonal kulcspontjaival változtathatók[11]. Saját szerkesztőfelületük van (3-12. ábra), ahol lehet állítani az egyes kulcspontokat, illetve az adott függvény kimeneti értékét. Kifejezetten egyszerű feladatok kezelésére készültek, mint például ajtónyitás,

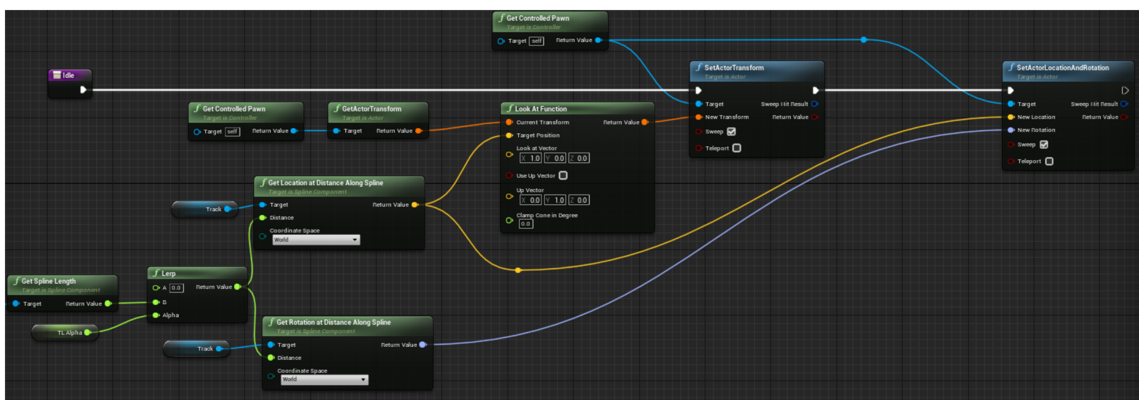
világítás módosítása, szereplők mozgatása vagy egyéb időközpontú változtatások végrehajtására.

3-12. ábra - Timeline szerkesztő



3-13. ábra - Timeline beállítása

Az *Idle_Flying* mód egy kereső üzemmód, amikor a karakter bejárja a pályát a saját területén, így folyamatosan helyezkedik és próbál a labda közelébe kerülni (3-14. ábra). Minden hajtóhoz tartozik egy-egy pálya, amit követve pásztázzák a saját területüket. A pálya egy *Spline* típusú komponens, amit a *Field* osztály tárol, és köt össze a hajtókkal. A játékos minden pillanatban a *Timeline* által visszakapott *TL_Alpha* érték alapján változtatja a helyzetét a *Spline*-hoz viszonyítva, így lényegében leköveti a pályát.



3-14. ábra - Idle_Flying

A kontroller folyamatosan vizsgálja, hogy a *Quaffle* az adott pillanatban szabad-e vagy valaki birtokolja. Amennyiben szabad, a vezérelt hajtó megvizsgálja, hogy ő van-e a legközelebb a labdához (*Active_Chaser*), és ha igen, akkor megszerzi a labdát. Ha a

Quaffle nem szabad vagy a karakter nem az *Active_Chaser*, akkor a hajtó marad/visszatér az *Idle_Flying* módba. Ha egy hajtó birtokolja a labdát akkor két dolgot tehet vele: megpróbál gólt dobni, vagy elpasszolja. Passzolás esetén a hajtó megkeresi a hozzá legközelebb álló játékost (a saját csapatából), megközelíti amíg a saját területe tart, majd eldobja a labdát a csapattársa irányába. A passz fogadása automatikusan működik, hiszen ilyenkor a csapattárs válik *Active_Chaser*-ré hiszen ő lesz a legközelebb a labdához, így elindul felé és megfogja. Gól dobás esetén a játékos odarepül az egyik góldobó pozícióba (a három gólkarikával szemben), majd átdobja az adott karikán (3-15. ábra). Az *Idle_Flying* mód önmagában kiszámítható lenne, viszont mivel az *Active_Chaser* minden pillanatban más, így folyamatosan változik az, hogy a hajtó éppen milyen tevékenységet folytat (labdáért megy, passzol stb.).

3-15. ábra - Try to Score

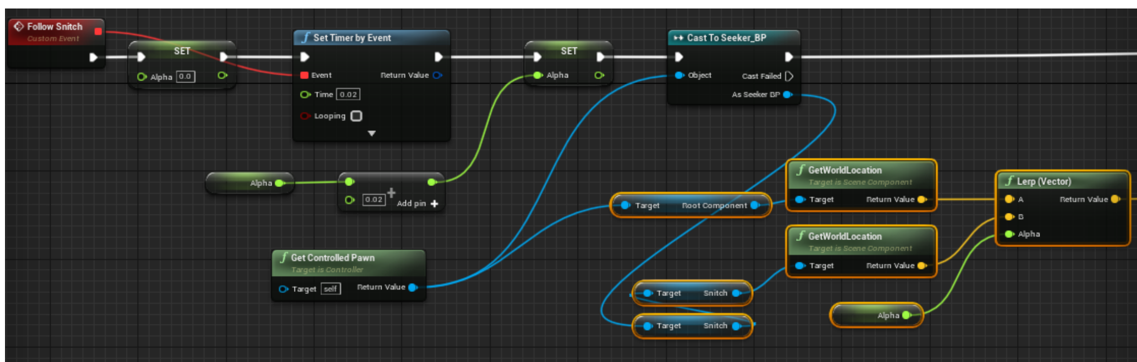
3-16. ábra - Viselkedésfa

3.2.3 Seeker

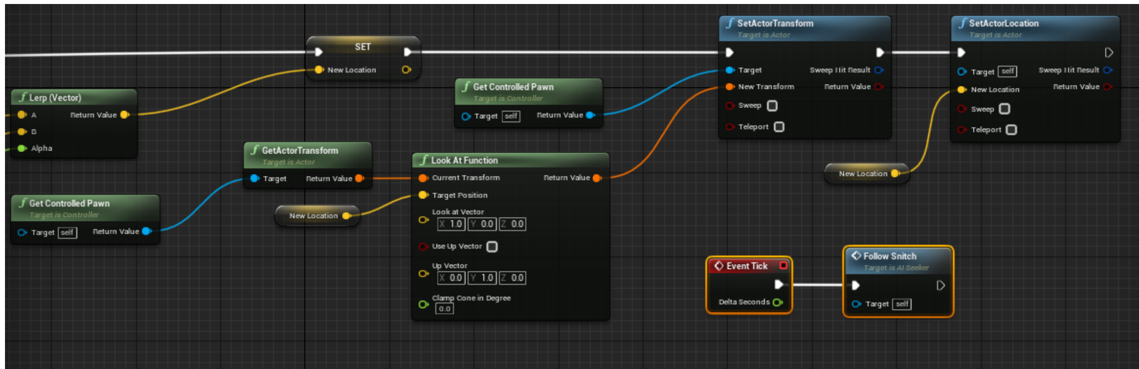
A fogó (*Seeker*) feladata, hogy a játék során megkeresse, és elkapja a cikeszt (*Golden Snitch*). A *Seeker_BP* osztály leszármazik a *Character_BP* osztályból, így megörökli a tulajdonságait és az alapvető viselkedését. A fogónak – bár nem ez a fő feladata – ugyan-úgy lehetősége van megszerezni, tartani vagy eldobni a *Quaffle*-t mint a hajtóknak, viszont a játék körülményeit más szemszögből kell vizsgálnia, így külön *AI-Controller* vezérli.

Az *AI_Seeker* kontroller egy külön *AI Controller*, aminek célja, hogy a fogó minden időpillanatban a cikesz helyzetét követve változtassa a saját helyzetét. Így lényegében leköveti a cikesz röppályáját amíg el nem kapja. A játékélmény kedvéért a mesterséges intelligenciával vezérelt fogó sosem fogja elérni azt a pontot amikor el lehet kapni a cikeszt, ám ha a játékos a fogóra váltva átveszi annak vezérlését, a felhasználó el tudja kapni.

A fogónak a vezérlése minden pillanatban egy *Follow_Snitch* eseményt vált ki, ami a karaktert a cikeszt követve mozgatja (3-17. ábra, 3-18. ábra). A konkrét mozgatást egy *Lerp* függvény segítségével végezzük, aminek kimenete alapján változtatjuk a karakter helyzetét. A *Lerp* függvénynek általában három bemenete és egy kimenete van, és a feladata, hogy a – 0 és 1 közé eső *Float* – *Alpha* érték alapján visszaadjon egy *A* és *B* közötti értéket, ahol *A* és *B* a függvény két másik bemenete. *Alpha = 0* esetén 100% *A* lesz a kimenet, *Alpha = 1* esetén 100% *B*, a kettő között pedig egy folytonos lineáris átmenet. A *Follow_Snitch* eseményben egy *Timer* segítségével növeljük a *Lerp* függvénynek átadott *Alpha* értéket, *A* és *B* értéknek pedig a saját, és a cikesz aktuális pozícióját adjuk át. Mivel az esemény minden időpillanatban kiváltódik, a fogó egy szép, folytonos íven követi le a cikesz mozgását.



3-17. ábra - Follow Snitch/1



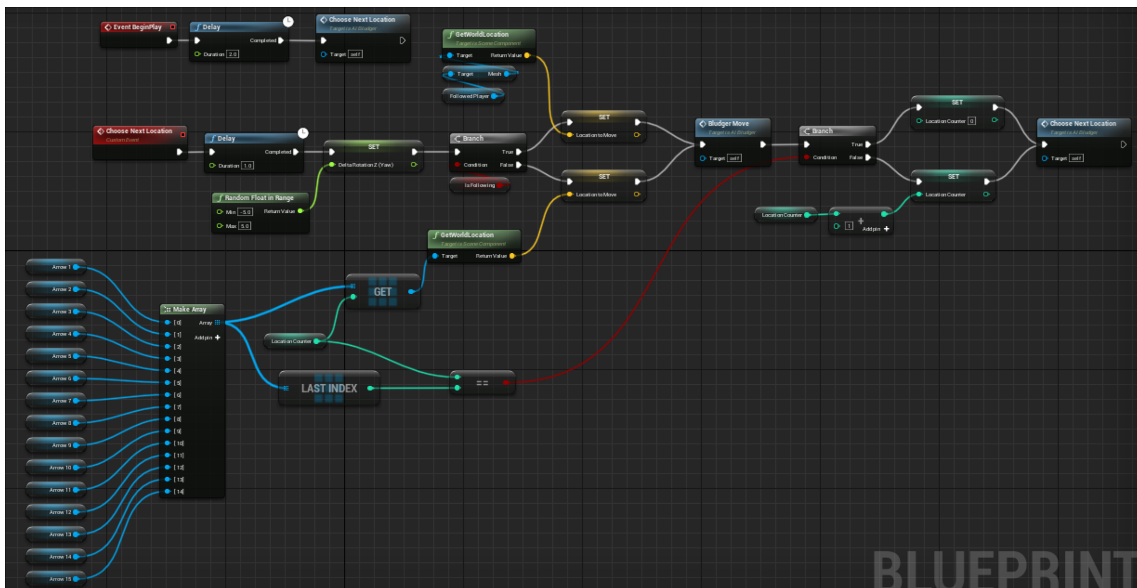
3-18. ábra - Follow Snitch/2

3.3 Labdák

3.3.1 Bludger

A játék szerint a gurkók a pályán cikázva repülnek azzal a céllal, hogy lelökjék a játékosokat a seprűről. Ezek alapján a gurkó viselkedésében szükség van egy alaphelyzeti viselkedésre és egy támadó viselkedésre. Az *AI_Bludger* osztály vezérli a gurkót a játék során (3-19. ábra). Minden gurkónak van egy halmaza *Arrow* típusú (aminek nincsen fizikai kiterjedése) komponensekből, amik egyenletesen el vannak osztva a pályán, és amik a lehetséges cél pozíciókat jelölik, ahova a gurkó repülhet. Fontos, hogy ez a halmaz a *Mesh* komponensen kívül, közvetlen a gyökér alatt helyezkedjen el a hierarchiában, hiszen így tudjuk elkerülni, hogy a gurkóval együtt a pozíciók is elmozduljanak.

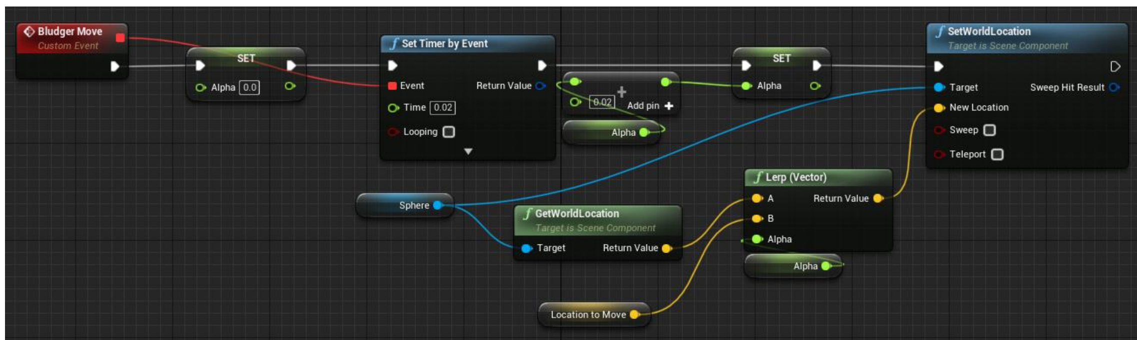
Két fontosabb esemény vezérli a gurkót: *Bludger_Move és Choose_Next_Location*. A játék kezdetén elindul a *Choose_Next_Location,* ami kiválasztja az új pozíciót ahová a gurkó menni fog, kiváltja a *Bludger_Move* eseményt (3-20. ábra), majd, ha mindennel végez újra indítja a *Choose_Next_Location*-t így egy ciklust generál, ami minden pillanatban folytonosan mozgatja a gurkót. Az új pozíciót az *Arrow* komponensek helyzetei adják meg.



3-19. ábra - AI_Bludger

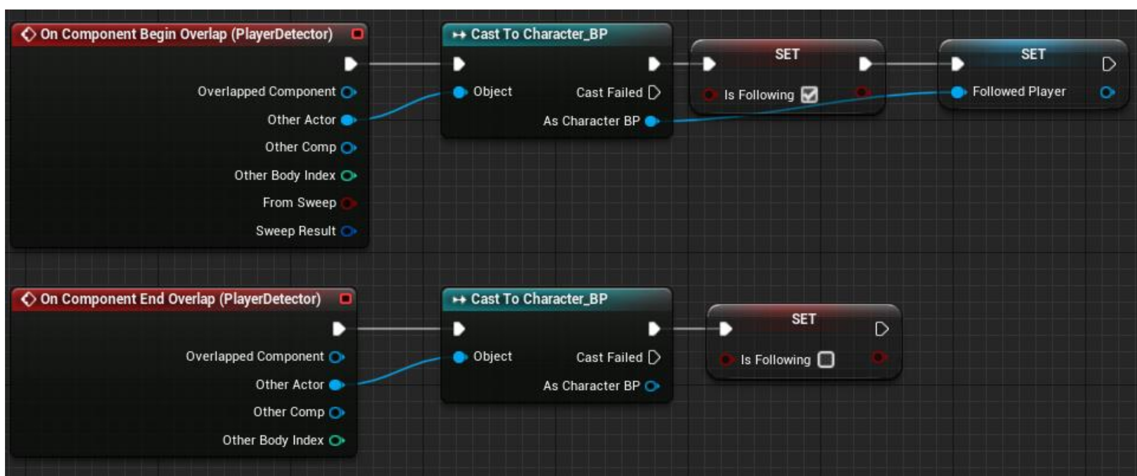
A konkrét mozgatás során egy *Timert* kell használni, ami minden pillanatban változtat egy *Alpha* értéket, amit átad egy *Lerp* függvénynek. A *Bludger_Move* esemény

az *A* értéknek átadja a saját jelenlegi pozícióját, *B* értéknek pedig a már kiszámolt *Next_Location* értéket, így a gurkó egy szép folytonos íven repül a pályán.



3-20. ábra - Bludger_Move

Támadás során a *Next_Location* érték az éppen támadott játékos pozíciója lesz. A támadás kezdetét és végét egy *Bool* változóval figyeljük, amit pedig egy *Box Collider* ütközésével állítunk (3-21. ábra).



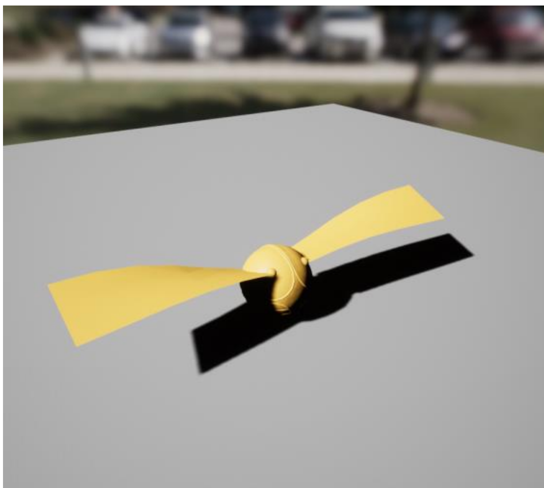
3-21. ábra - Gurkó támadása

3.3.2 Golden Snitch

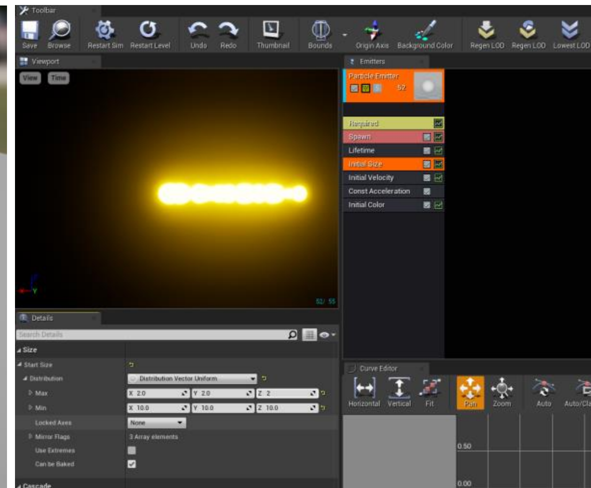
Az Aranycikesz a játék szerint a pályán cikázva nagy sebességgel és kiszámíthatatlanul repül, amíg a két fogó közül valamelyik el nem kapja. A cikeszt az *AI_Snitch* nevű osztály valósítja meg (3-22. ábra).

Az *AI_Snitch* osztály tartalmaz egy *Static Mesh* komponenst, ami a cikesz fizikai modellje, a gurkókhoz hasonlóan tartalmaz a mozgáshoz szükséges *Arrow* típusú komponensekből álló halmazt, egy *Text Renderer* objektumot és egy *Particle System* komponenst.

Az *Particle System* egy integrált részecskeeffektus-szerkesztő, ami a Cascade felület segítségével szerkeszthető (3-22. ábra). A felület valós idejű képet mutat, és moduláris effektusszerkesztést kínál, ezzel lehetővé téve összetettebb effektusok gyors és egyszerű létrehozását (3-24. ábra). Magának a részecskerendszernek az elsődleges feladata a részecskék viselkedésének szabályozása, míg a teljes rendszer egészének sajátos megjelenését saját *Material*-ok hozzáadásával szabályozhatjuk[12].



3-22. ábra - Cikesz modell



3-23. ábra - Particle System szerkesztő

3-24. ábra - Cikesz megjelenése

A cikesz mozgása hasonló a gurkóéhoz, a gurkóval ellentétben viszont nem egy meghatározott pályát követve repül, hanem folyamatosan változtatja az útvonalát. A navigációs pozíciók közül véletlenszerűen kiválaszt egyet és elindul felé, majd mikor odaért egy szintén véletlenszerű ideig nem változtat helyzet, majd elindul a következő véletlenszerűen generált pozíció felé.

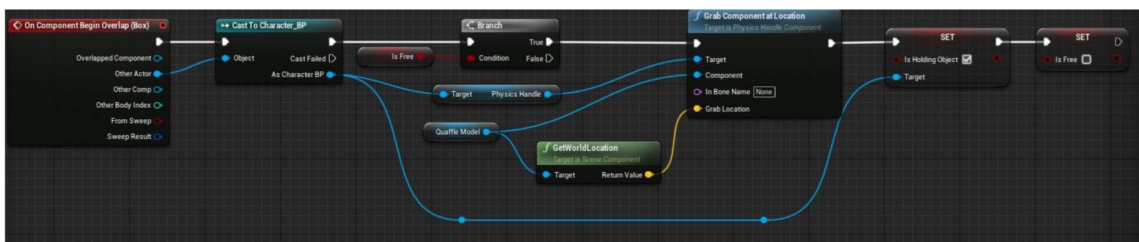
A játék végét az jelenti, ha valamelyik csapat fogójának sikerül elkapni a cikeszt. Az elkapáshoz a játékban a fogónak meg kell közelítenie a cikeszt, és ha elég közel ér hozzá, az osztály *Text Renderer* típusú *GrabText* komponense láthatóvá válik, és ebben az esetben a játékos elkaphatja a cikeszt (3-25. ábra). A *GrabText* amikor látható, minden esetben a játékos felé fordulva ajánlja fel a lehetőséget a cikesz elkapására.

3-25. ábra - Cikesz elkapása

3.3.3 Quaffle

A *Quaffle* az a labda, amivel a játékosok a legtöbbet érintkeznek. A hajtók képesek megfogni, passzolni, vagy gólt dobni vele. A hajtók vezérléséhez szükséges információ, hogy minden pillanatban meg lehessen állapítani, hogy a *Quaffle*-t éppen birtokolja-e valamelyik játékos, így ezt a *BP_Quaffle* osztály osztályváltozóként tárolja.

A labda megfogásához egy *Box Collider* komponens szükséges, ami, ha ütközést detektál egy *Character_BP* típusú objektummal, akkor a *Quaffle*-t az adott hajtó *Physics Handle* típusú komponenséhez csatolja, így onnantól kezdve a labda pozíciója az adott hajtó mozgásával együtt változik (3-26. ábra).

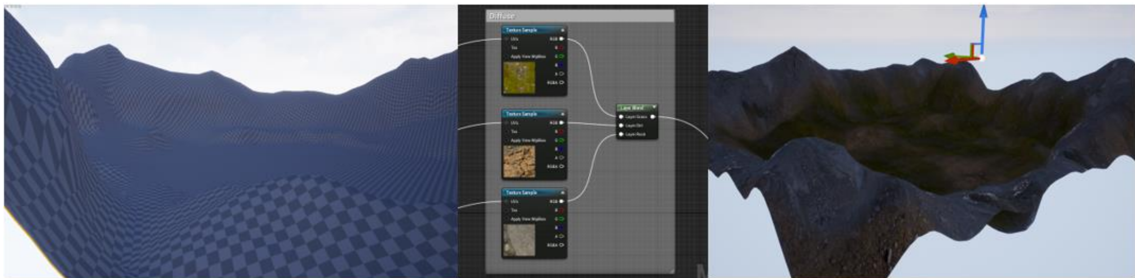


3-26. ábra - Quaffle megfogása

3.4 Egyéb Komponensek

3.4.1 Landscape

Az Unreal Engine 4 képes korlátlan domborzati világokat létrehozni a saját beépített terepszerkesztő eszközkészletével. A különböző eszközök segítségével tetszőleges domborzati formákat adhatunk hozzá a játéktérhez, vagy manipulálhatunk már meglévőket. A létrehozott táj három különböző textúrával van ellátva. Minden textúrának megvan a maga *BaseColor, Roughness* és *Normal* értéke, amikből így előállíthatunk egy saját *M_Terrain* anyagot (*material*). Ha a három textúrát egyesével csak hozzáadnánk a tájhoz, úgy egyszínű lenne a teljes felület, így viszont a saját létrehozott anyaggal tetszőlegesen lehet "kiszínezni" a tájat az élethűbb kinézet érdekében (3-27. ábra).

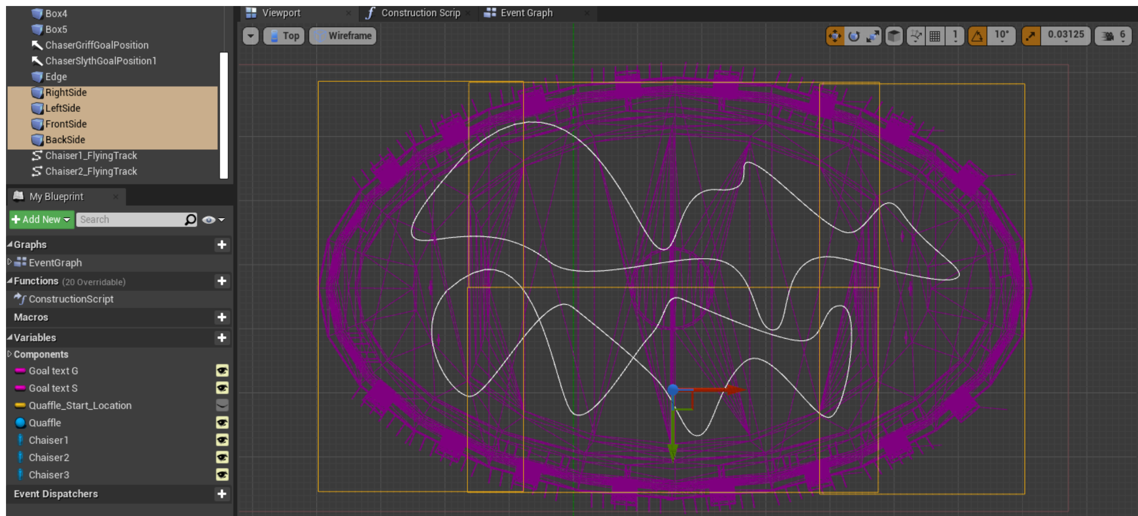


3-27. ábra - Táj szerkesztése

3.4.2 Field

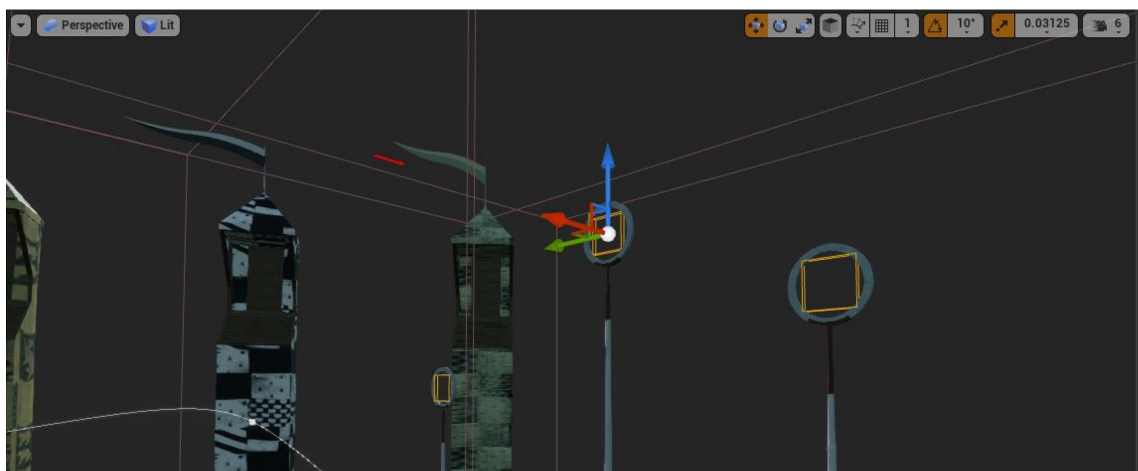
A *Field Blueprint* osztály az egyik legfontosabb komponense a játéknak, hiszen ez az objektum látja, és így összeköti a csapatokat, a labdákat és minden egyéb komponensét a játéknak. Az osztály *Static Mesh* komponense tárolja a játéktér fizikai modelljét.

A pálya fel van osztva négy területre, amik alapján minden játékos meg tudja kapni a saját „területét", amin belül mozoghat (3-28. ábra). A játékosok elhagyhatják a saját területeiket amikor meg akarják szerezni a labdát. A pálya ezen kívül tartalmaz a hajtók számára egy alapértelmezett útvonalat, amit az *Idle_Flying* eseményre elkezdenek követni a játékosok.

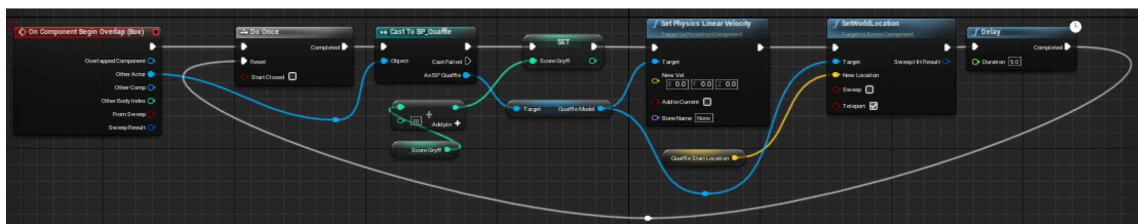


3-28. ábra - Pálya területei

Ezen felül a pálya tartalmazza a játékosok góldobó pozícióját, a csapatok pontszámait és a góldetektáláshoz szükséges collidereket. A póznákban lévő *Box Colliderek* ütközés esetén megvizsgálják, hogy mi ment át rajtuk, és ha a *Quaffle*, akkor növeli az adott csapat pontszámát, illetve visszahelyezi a *Quaffle*-t a kezdőpozíciójába (3-29. ábra, 3-30. ábra).



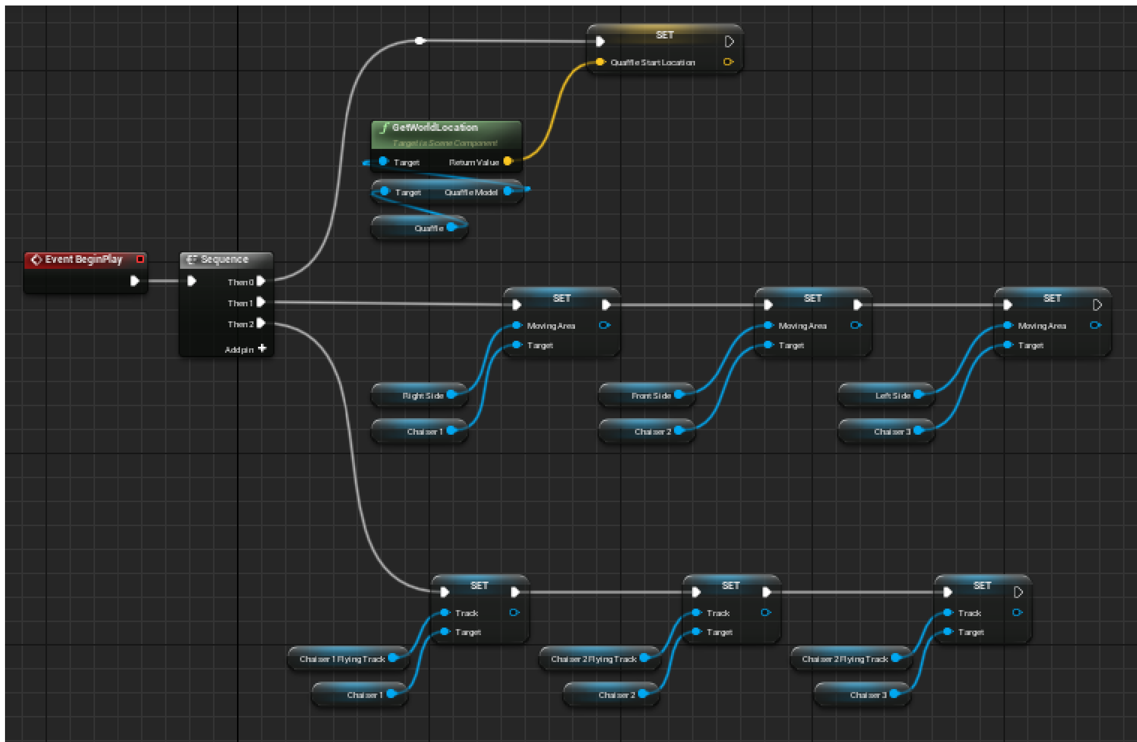
3-29. ábra - Gól detektálása/1



3-30. ábra - Gól detektálása/2

A játék elején szintén a *Field* állítja be a játékhoz szükséges kezdő paramétereket, mint például a labdák kezdő pozíciója, a játékosok alapértelmezett útvonala vagy a területe (3-31. ábra).

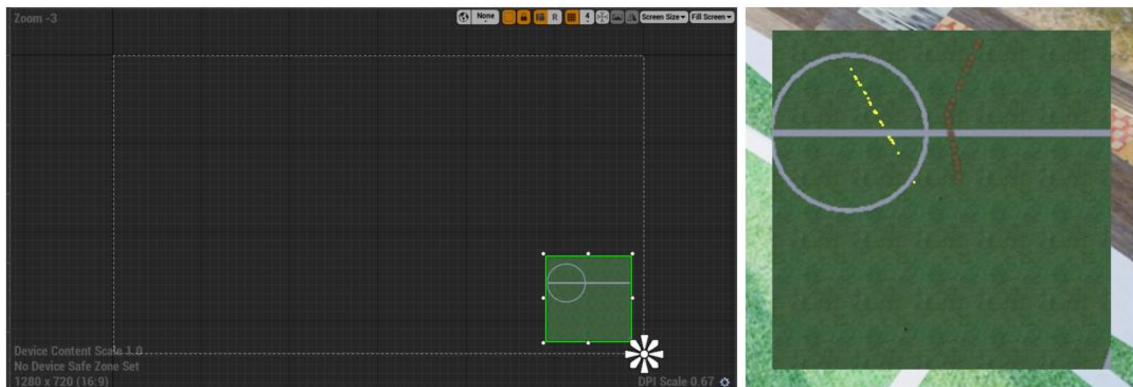
A játék során a *Field* minden időpillanatban megvizsgálja, hogy melyik *Chaser_BP* típusú játékos van legközelebb a *Quaffle*-hoz, majd az aktuális hajtónál beállítja az *ActiveChaser* változót *True* értékre, amiből így a karakter *AI Controllere* változtathatja a hajtó viselkedését.



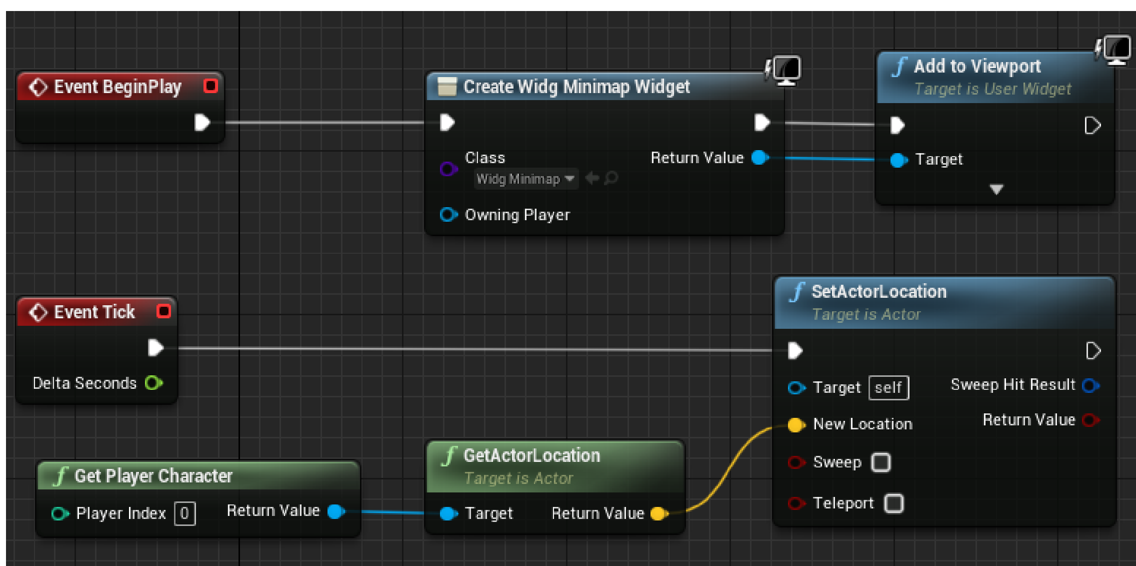
3-31. ábra - Kezdeti beállítások

3.4.3 Minimap

A képernyő jobb alsó sarkában található egy térkép (*minimap*) ami a valós időben mutatja a játékos helyzetét a pályán (3-32. ábra, 3-33. ábra). A térképet egy *Render Target* és egy *Widget Blueprint* típusú komponens valósítja meg, illetve egy kamera, ami a játékossal mozog és felülnézetből ortografikus módon mutatja a képet.



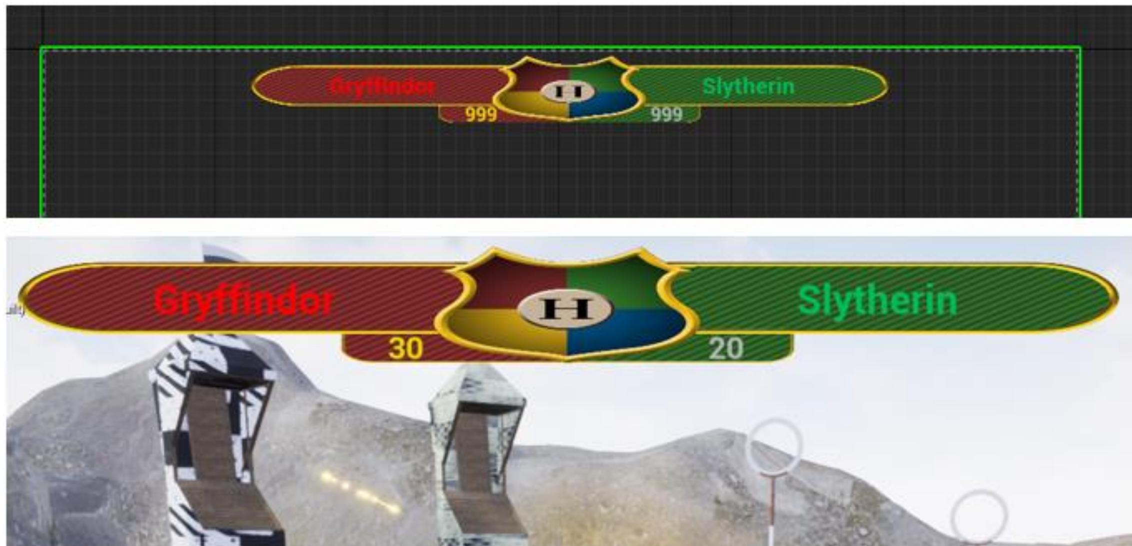
3-32. ábra - Minimap



3-33. ábra – Minimap hozzáadása a képernyőhöz

3.4.4 Pontrendszer

A csapatok pontszámai egy külön (képernyőhöz rögzített) *Widget*-ben jelennek meg (3-34. ábra). A *Widget* tárolja a csapatok nevét, illetve megjeleníti a pontszámot, amit a *Field* osztály tárol és módosít.



3-34. ábra - ScoreBoard

3.4.5 Audio

A hangeffektusok és háttérzenék létfontosságú elemek az élvezhető játék-környezet létrehozásához. Az Unreal Engine Audio Engine rendszere olyan eszközöket kínál, amelyek lehetőséget nyújtanak szinte bámilyen típusú hangeffektus létrehozására. Egy adott hang eredeti változata előállítható egy külső alkalmazásban, importálható, majd a motoron belül manipulálható.

A játékban van egy konstans háttérzene, ami egy *BP_Background_Music* osztállyal valósul meg. A színtérhez hozzá van adva az osztály példánya, ami a játék indulásakor el is kezdi lejátszani a kapott audio fájlt ismétlődően. Hasonlóan van megvalósítva a pontszerzések hangja, mely esetben a póznákban lévő *Box Colliderek* alapján játszódik le a hangeffektus

4 Értékelés

4.1 Játékélmény

A program készítése során igyekeztem a megfelelő játékélmény elérésére, és az eredeti Quidditch játék érzetének megteremtésére. Ezt nagyon sok tesztelés és próbálkozás árán értem el, vizsgálva a játékosok repülésének folyékonyságát, a méretek és sebességek arányosságát, illetve a mesterséges intelligenciák „okosságát”. A játék nehézségi szintje jelenleg felhasználóbarát, azonban ez a későbbi fejlesztések során finomítható, fokozható. A játékélmény érdekében csak a felhasználó tudja befejezni a játékot (azaz elkapni a cikeszt), pontot szerezni viszont az MI játékosok is tudnak, így abban az esetben veszíthet a felhasználó, ha akkor kapja el a cikeszt, amikor az ellenfél csapatának több mint 150 pont előnye van a saját csapatával szemben (mivel a cikesz 150 pontot ér és a játék véget ér, ha elkapják). A játékélményhez nagyban hozzájárulnak a különböző hangeffektusok. A folyamatos háttérzene fokozza az izgalmat, a gól szerzések hangja, illetve egyéb hangeffektusok pedig a pillanatnyi élményeket növelik.

4.2 Mesterséges intelligencia minősége

4.2.1 Cikesz

A cikesz mesterséges intelligenciája viszonylag egyszerű, mégis tökéletesen illeszkedik a játékba, hiszen a tesztelések alapján nem egyszerű feladat elkapni. A cikesz navigációs pontok által vezérelt útkeresést használ, ami a videojátékok mesterséges intelligenciájához használt két legnépszerűbb módszer egyike. Az elkapásához szükséges megközelítési távolság, a kiszámíthatatlan várakozás, illetve pozíciók kellően megnehezítik a felhasználó feladatát, így növelve a játékélményt.

4.2.2 Gurkók

A gurkók az eredeti sporthoz hasonlóan képesek megtámadni a játékosokat, illetve folyamatosan keresik a megtámadható karaktereket. A pályájuk kiszámítható, viszont kellően nagy a veszélyzónájuk (amin belül a labda rátámad a játékosra), így nehéz elkerülni őket. A játékélmény érdekében viszonylag könnyű elmenekülni egy támadás elől, mivel a labdák sebessége jelentősen kisebb, mint a játékosoké. Későbbi fejlesztési lehetőségként, a gurkó támadási sebességét lehetne paraméterhez kötni, így nehézségi

szinteket váltani. A cikeszhez hasonlóan az útkeresése rögzített navigációs pontok alapján működik.

4.2.3 Fogó

A fogó jelenlegi mesterséges intelligenciája a játék menetéhez nem tesz hozzá, viszont kellő ösztönzést tud adni a felhasználónak, ha az látszik, hogy az ellenfél fogója lassan elkapja a cikeszt. Természetesen az MI-vel vezérelt játékos nem tudja befejezni a játékot, bár a későbbi fejlesztések során be lehet állítani egy időzítőt, ami alatt, ha a felhasználó nem fogja meg a cikeszt, már az MI is be tudja fejezni a játékot, így motiválva a felhasználót.

4.2.4 Hajtók

A hajtók vezérlése az összes objektum közül a legösszetettebb. Az intelligencia szintje majdnem összehasonlítható egy valódi játékoséval. A játékban előforduló összes eseményre van releváns válaszlépése, az egymással végzett interakciójuk pedig hasonlít egy átlagos sportban a játékosok viselkedéséhez (passzolás, góldobás, labdakeresés). A stratégiai gondolkodásuk egyszerű, mégis megfelelő játékélményt tud nyújtani amikor a játékos például el akarja fogni az ellenfél passzát, vagy kivédeni a góldobást.

5 Összefoglalás

5.1 Quidditch Game

Összességében a játék megvalósítja mind a feladatleírásban és specifikációban leírt követelményeket, mind a Harry Potter filmvilágban bemutatott Quidditch sport szabályait és menetét. A fejlesztés során megismertem többek között az Unreal Engine játékmotort, illetve mélyebben elmerültem a különböző mesterséges intelligenciák tanulmányozásában. Az elkészült program jelenleg közel sem végleges, a későbbiekben folytatom még a játék fejlesztését, hiszen nagyon sok lehetőség rejlik még benne.

5.2 Fejlesztési lehetőségek

A további fejlesztésekre még rengeteg lehetőség van, beleértve a játékélményt növelő funkciók bevezetését, vagy a kimaradt játékosok (terelő, őrző) megvalósítását, de akár hajtók stratégiai gondolkodásmódjának további fejlesztése is nagyban növelheti a játékélményt.

A kész játék érdekében szükség lehet egy megfelelő menü rendszer létrehozására, a karakterek további animálására, illetve további különböző karakter modellek hozzáadására. A játékélmény fokozása érdekében egy hálózatról kapcsolódó második, vagy harmadik felhasználó csatlakozási lehetőségének megvalósítása is előnyös lehet.

Irodalomjegyzék

[1] Mesterséges Intelligencia videojátékokban: https://en.wikipedia.org/wiki/Artificial_intelligence_in_video_games

Wikipédia, [Hozzáférés dátuma: 2021, november 15]

[2] Behavior Tree: https://www.orfeasel.com/creating-a-basic-patrol-ai/

Unreal Engine, [Hozzáférés dátuma: 2021, november 15]

[3] Mesterséges Intelligencia: https://en.wikipedia.org/wiki/Artificial_intelligence

Wikipédia, [Hozzáférés dátuma: 2021, november 29]

[4] Viselkedésfák: https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/ArtificialIntelligence/AIPerception/

Unreal Engine, [Hozzáférés dátuma: 2021, december 01]

[5] Viselkedésfák: https://en.wikipedia.org/wiki/Behavior_tree_(artificial_intelligence,_robotics_and_control)

Wikipédia, [Hozzáférés dátuma: 2021, december 03]

[6] Állapotgépek: https://en.wikipedia.org/wiki/Finite-state_machine

Unreal Engine, [Hozzáférés dátuma: 2021, december 05]

[7] Karakter modell: http://www.mediafire.com/file/tx212ntv48mr4z2/Harry_Potter_Half_Blood_Prince.zip/file

[Hozzáférés dátuma: 2021, augusztus 29]

[8] Unreal Bemenetek: https://www.unrealengine.com/en-US/blog/input-action-and-axis-mappings-in-ue4?sessionInvalidated=true

Unreal Engine, [Hozzáférés dátuma: 2021, december 03]

[9] Physics Handle: https://docs.unrealengine.com/4.27/en-US/Basics/Components/Physics/

Unreal Engine, [Hozzáférés dátuma: 2021, szeptember 20]

[10] AI Controller: https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/Framework/Controller/AIController/

Unreal Engine, [Hozzáférés dátuma: 2021, október 22]

[11] Timeline: https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/UserGuide/Timelines/

Unreal Engine, [Hozzáférés dátuma: 2021, november 26]

[12] Particle System: https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/ParticleSystems/UserGuide/

Unreal Engine, [Hozzáférés dátuma: 2021, november 16]