

MISKOLCI EGYETEM
Gépészmeérnöki és Informatikai Kar
Alkalmazott Informatikai Intézeti Tanszék

SZAKDOLGOZAT



MISKOLCI EGYETEM

Számítási modellek bemutatása egy FPS játék készítése kapcsán

Készítette:

Árva Zoltán

Mérnökinformatikus, BSc

Témavezető:

Piller Imre, egyetemi tanársegéd

MISKOLC, 2017

MISKOLCI EGYETEM

Gépész-mérnöki és Informatikai Kar

Alkalmazott Matematikai Intézet Tanszék

Szám:

SZAKDOLGOZAT FELADAT

Árva Zoltán (B5X5NT) mérnökinformatikus jelölt részére.

A szakdolgozat tárgyköre: Számítógépi grafika, optimalizálás, C++ programozás

A szakdolgozat címe: Számítási modellek bemutatása egy FPS játék készítése kapcsán

A feladat részletezése:

A számítógépes játékok különféle komplex számítási és programozási problémákat vetnek fel. Ilyenekkel a mérnöki gyakorlatban is napi szinten találkozhatunk.

A dolgozat célja, hogy egy belső nézetes, lővöldözős játék elkészítése kapcsán bemutasson ezek közül néhányat, amelyek lehetnek például az alábbiak.

- Ütközésvizsgálat térben. Hatékonyság javítása térrészszektorral.
- Útvonalkeresési módszerek egzakt és heurisztikus algoritmusokkal.
- Interpolációs módszerek alkalmazása számítógépes játékok karaktereinek animálásához.
- Döntési módszerek vizsgálata, alkalmazása a nem játékos karakterek viselkedésének modellezéséhez.

Témavezető: Piller Imre (egyetemi tanársegéd)

A feladat kiadásának ideje:

.....
szakfelelős

EREDETISÉGI NYILATKOZAT

Alulírott Árva Zoltán; Neptun-kód: B5X5NT, a Miskolci Egyetem Gépészszmérnöki és Informatikai Karának végzõs mérnök informatikus szakos hallgatója ezennel bùntetõjogi és fegyelmi felelõsségem tudatában nyilatkozom és aláírásommal igazolom, hogy "Számítási modellek bemutatása egy FPS játék készítése kapcsán" címû szakdolgozatom/diplomatervem saját, önálló munkám; az abban hivatkozott szakirodalom felhasználása a forráskezelés szabályai szerint történt.

Tudomásul veszem, hogy szakdolgozat esetén plágiumnak számít:

- szószerinti idézet közlése idézõjel és hivatkozás megjelölése nélkül;
- tartalmi idézet hivatkozás megjelölése nélkül;
- más publikált gondolatainak saját gondolatként való feltüntetése.

Alulírott kijelentem, hogy a plágium fogalmát megismertem, és tudomásul veszem, hogy plágium esetén szakdolgozatom visszautasításra kerül.

Miskolc, 2017. év 11. hó 24. nap

.....

Hallgató

Tartalomjegyzék

1. Bevezetés	1
2. Problémakör	2
2.1. Játékmotorok	2
2.2. Egyedi igények és megoldandó problémák	3
2.3. Optimalizálási problémák	3
2.4. A játékmotor fő elemei	4
2.4.1. Mesterséges intelligencia	4
2.4.2. Hangok	4
2.4.3. Magasságmező	4
2.4.4. Modellek kezelése	4
2.4.5. A csapda, mint játékelem	4
2.4.6. A lövés	5
2.5. A játék szabályai és mechanikája	5
3. Komponensek	6
3.1. Felhasznált technológiák	6
3.2. Az alkalmazás fő részei	6
3.2.1. Magasságmező betöltése, textúrázása	7
3.2.2. A karakter irányítása és a játék fizikája	9
3.2.3. Hangok betöltése és lejátszása	9
3.2.4. Lövessel kapcsolatos számítások	9
3.2.5. Mesterséges intelligencia	10
3.3. A játékindítás folyamata	11
4. Ütközésvizsgálat	12
4.1. A karakterek ütközésvizsgálata mozgás szempontjából	12
4.2. A domborzat és tereptárgyak ütközésvizsgálata a lövés szempontjából . .	14
4.2.1. Program: Lövedék ütközésvizsgálata	14
4.3. A játékos, és az ellenfelek ütközésének vizsgálata	15
4.4. Optimalizálási módszerek	16
4.4.1. A játéktér szabályos felosztása	16

4.4.2. Négyes fa és az oktális fa	17
4.4.3. A BSP fa	17
4.4.4. Optimalizálási módszer kiválasztása	18
4.4.5. Program: Térfelosztás négyes fa segítségével	18
5. Útvonalkeresés	21
5.1. A waypoint-ok fogalma és jellemzői	21
5.2. A waypoint-hoz tartozó adatok	21
5.3. Útvonalkeresés waypoint-ok alapján	21
5.4. Az A*-algoritmus	22
5.5. Program: Útvonalkeresés A*-algoritmussal	23
6. Animáció	26
6.1. Kulcsképkocka alapú animáció	26
6.1.1. Az md2-es formátum	27
6.2. Csontváz alapú animáció	27
6.2.1. Program: Emberszerű karakter animálása	28
6.3. Interpolációs módszerek	30
6.3.1. Lineáris interpoláció	30
6.3.2. Lagrange-interpoláció	30
6.3.3. B-spline	30
6.3.4. Program: Lagrange Interpoláció	31
7. Ellenfelek viselkedésének modellezése	33
7.1. Viselkedés bemenetei és kimenetei	33
7.1.1. Bemenetek	33
7.1.2. Kimenetek	35
7.1.3. Program: Gépi ellenfél viselkedésének modellezése	35
7.2. Véletlenszerű tényezők	36
7.2.1. Konkrét megvalósítás	36
8. Tesztek	38
8.1. Kirajzolási módszerek	38
8.2. Profilozás	38
9. Összegzés	42
10. Summary	43
Irodalomjegyzék	44
Adathordozó használati útmutató	46

1. fejezet

Bevezetés

Egy FPS (*First-Person Shooter*) játék elkészítése különféle számítási- és optimalizálási problémákat vet fel. A számítógépes grafika és a játékfejlesztés segítségével lehetőség adódott a termelésinformatikához kapcsolódó optimalizálási problémák bemutatására.

Azon számítógépes játékoknak, amelyek célja a minél valószerűbb megjelenítés, számos olyan eleme van, amely optimalizálás nélkül olyan erőforrásigényes lenne, amely a manapság elérhető csúcskategóriás számítógépeken sem futna megfelelően. Ilyen elem például a karakterek mozgásterének definiálása, a lövés során eltalált objektumok metszéspontjának számítása, az ellenfelek viselkedésének szimulálása.

Megjelenítés szempontjából nagyon fontos, hogy a videokártyának megfelelő formában, egyben adjuk át a kirajzolni kívánt elemeket, mert így hatékonyabb lesz a megjelenítés, jobban ki lehet használni a rendelkezésre álló erőforrásokat, illetve magasabb másodpercenkénti képkockaszámot érhetünk el.

A problémakör egyik fontos eleme az ellenfelek bejárató területének kezelése. Túladonképpen ez is egyfajta ütközésvizsgálatot jelent a falakra, objektumokra nézve. A dolgozatban bemutatásra kerülnek azon térfelületek részletei, amelyekkel a számítások gyorsabban, hatékonyabban elvégezhetők. Az ütközésvizsgálat egy másik tipikus esete az ilyen jellegű játékokban a lövedékek útjának és metszéspontjainak kiszámítása. Ennél például jelentős optimalizációt érhetünk el, ha csak azon objektumokat vesszük figyelembe, amelyek a lövedék haladási irányába esnek.

Az ellenfelek találatainak regisztrálását az optimális teljesítmény érdekében nem közvetlenül a játékos számára látható, komplex geometriára végezzük el, hanem egy egyszerűbbre. Ezen egyszerűbb geometria részletek meghatározása nagyon fontos tényező, mert ha túl egyszerű, nem lesz pontos a találatok regisztrálása, ha túl komplex, akkor pedig túl nagy lesz az erőforrásigénye.

Az ilyen jellegű játékoknak egy fontos eleme az ellenfelek viselkedésének modellezése is. minden ellenfélnek van egy aktuális célja, amit a különböző környezeti tényezők befolyásolhatnak. Fontos, hogy a játékos úgy érezze, hogy a mesterséges intelligencia ésszerűen cselekszik, ennek megvalósítását is részletezi a későbbiekben a dolgozat.

2. fejezet

Problémakör

2.1. Játékmotorok

Új játék fejlesztésénél két fő lehetőség adott. Az első, hogy választunk egy kész, elérhető grafikus vagy játékmotort, amely majd a játék alapját adja. Amennyiben nem találunk megfelelőt, akkor írhatunk egyet saját magunknak.

Elterjedt játékmotorok például az Unreal engine [1], Cryengine [2], Quake engine (id Tech) [3].

Az Unreal engine első változata 1998-ban jelent meg, az első játék, amelyik ezt használta az Unreal nevű játék volt. A motor jelenlegi, legújabb verziója a 4.17-es. Nagyon sokat fejlődött az évek során, mind modellezés, UV, világítás, animációk, és hangok kezelése terén.

A CryEngine-t először a Far Cry (2.1. ábra) nevű játéknál használták 2004-ben, a második, illetve harmadik verziót pedig a Crysis trilógiához. Ezen játékok mindegyike a magas, korát megelőző grafikai megjelenésről lett híres, nagyon szép összképet lehet elérni ezzel a motorral. Az első változat már támogatta a pixel és vertex shaderek 3.0-ás változatát, illetve a HDR megvilágítást.

Az általam írt motorhoz a legközelebb a Quake engine (id Tech) 2-es és 3-as verziója áll. Ezen motorok, összességében bármelyik megoldást is választjuk, a játékunk fő komponenseinek a háttérszámításait végezik, például lövés pályájának számítása, játék fizikájának számításai, ütközésdetektálás, billentyűzet és egérkezelés, hangokkal kapcsolatos számítások, hálózati kommunikáció, animációk, mesterséges intelligencia. A játékmotor kiválasztása vagy megírása után, a következő lépés a játék felépítése a motor adta eszközökkel. Praktikusan arra kell törekednünk, hogy olyan játékmotort válasszunk, amelyikben minél több kész funkció elérhető számunkra.



2.1. ábra. Pillanatkép a Far Cry című játékból (Crytek)

2.2. Egyedi igények és megoldandó problémák

A saját játékomban tervezésekor megvizsgáltam, hogy milyen jellegű programra, milyen funkciókra van szükség, azt hogyan tervezem majd megvalósítani. Egy kész játékmotor használata helyett egy saját játékmotor implementálására esett a választás. A döntést részben az indokolta, hogy az egyedi funkciók implementálására egy kész játékmotor esetében is szükség lenne. A fő ok viszont az volt, hogy egy játékmotor fejlesztése ki-mondottan alkalmas különféle matematikával (geometriával, optimalizálással) és programfejlesztéssel (szoftver architektúrával és implementációval) kapcsolatos problémák vizsgálatára, szemléltetésére és megoldására.

2.3. Optimalizálási problémák

A játékban megvalósítandó elemek között szerepel a lövés, a mesterséges intelligencia, az ütközésvizsgálat, és mindenek megvannak a maga optimalizálási problémái. Ütközésvizsgálatra szükség van a játékos karakterének az ellenfeleknek és a lövéshez is a találatok regisztrálására, pontos helyének számolására. A játéktér háromszögekből áll. A lövés megvalósításához szükségünk van háromszög, és egyenes metszéspontjának számítására, hogy vizualizálható legyen a lövedék becsapódásának helye a talalon, falakon, illetve egyéb játékbeli elemeken. Ez alapjában véve egy matematikai probléma, amit ki kell számoltatni, le kell programozni. Meg kell határozni azt az egyenest, ami azt mutatja meg, éppen merre nézünk, majd vizsgálni kell, hogy az adott egyenes metszi-e a teret alkotó háromszögek egyikét. De mivel több 100000 háromszög és egyenes metszéspontját kellene alap esetben számítani, így ez több optimalizálási problémát is felvet, amelyek a későbbiekben részletezésre kerülnek.

2.4. A játékmotor fő elemei

2.4.1. Mesterséges intelligencia

A mesterséges intelligenciának elsősorban az a feladata, hogy az ellenfelek egy meghatározott útvonalon mozogjanak, ne menjenek át falakon, menjenek egymásba. A véletlenszerűen lerakott ellenfelet a legközelebbi definiált waypoint-hoz kell irányítani, definiálni kell a célt (ami elsődlegesen a játékos), és ki kell számítani a pontokon keresztül vezető legrövidebb útvonalat.

2.4.2. Hangok

Egy játékhoz hozzátaroznak a hangeffektek, zenék is, amelyek élvezetessé teszik azt. Egy jól megválasztott zene például nagyon sokat tud javítani a játékélményen. Ezek megszólalásáért az `SDL_mixer` felelős, amely lehetővé teszi azt is, hogy több hang egy időben, átfedésben megszólaljon, és ne várják meg egymást. Lehetőség van hangcsatornákat megadni, folyamatos újrajátszást, illetve a hangok egymáshoz viszonyított hangerejét beállítani.

2.4.3. Magasságmező

A játékteret adó domborzat magasságmezővel lett kialakítva. A játék végleges verziójának alap elrendezését, útvonalait határozza meg, az azokon elhelyezkedő díszletek, illetve egyéb objektumok nélkül. Fontos az is, hogy a kép, amelyből a magasságmezőt számolja a program, bárki számára módosítható legyen, képes legyen saját egyedi pályát létrehozni egy egyszerű rajzolóprogram segítségével.

2.4.4. Modellek kezelése

A játéktér domborzatának adatait egy képből, az egyéb modellekkel pedig `.obj` kiterjesztésű fájlból olvassa be a játék. De csak a beolvasás még nem jelenti azt, hogy látjuk is azokat a kijelzőn. Ehhez a VBO-s (Vertex Buffer Object) kirajzolási módszert választottam, ami a videókártyának a legoptimálisabb adatstruktúrában adja át azt a lehető leggyorsabb kirajzoláshoz.

2.4.5. A csapda, mint játékelem

Mindenképp szerettem volna olyan elemet a játékba, ami megkülönbözteti a többi, hasonló, aréna jellegű, túlélős játéktól. Az egyik ilyen elem, hogy ha a játékos egy előre meghatározott időnél több ideig nem mozdul, akkor megjelenik alatta egy csapda, amelybe leesik, és meghal. Vannak olyan játékok, ahol szimplán az ellenfelek ösztönzik a játékost a mozgásra, de a játék itt kikényszeríti azt, hogy a játékosok mozgásban

legyenek. A megállásnál a küszöbérték tapasztalati alapon beállítva 5 másodperc lett. Ezt követően megjelenik a csapda modell egy hang kíséretében, majd 2 másodpercig még a kamera irányát tudja változtatni a játékos, utána viszont már azt sem. Ezt követően az a föld felé fordul, majd értesíti a játékosokat a történtek okáról.

2.4.6. A lövés

Egy ilyen játékban alapelem az is, hogy tudunk lőni, mivel e nélkül az egész értelmét vesztené. A játéktér alapvetően egy textúrázott polygonháló. A lövés során azt kell meghatározni, hogy a lövedék melyik háromszögbe ütközne bele, ha a nézet irányába lónénk, és egyenes pályát feltételeznénk. Ez egy olyan metszéspontszámítást tesz szükségessé, amelyet a játék összes objektumára el kell végeznünk. Megjelenítés szempontjából ez tipikusan egy lövés textúra illesztését jelenti.

2.5. A játék szabályai és mechanikája

Egy belső nézetes lövöldözős játék esetében az adott személy szemszögéből látjuk a virtuális világot. Egy olyan, aréna jellegű, túlélő játékról van szó, amelyben a játékosok mesterséges intelligenciával rendelkező ellenfelekkel (gyakori szóhasználat szerint *botokkal*) harcolnak.

A játék során az ellenfelek hullámokban érkeznek. Egy ilyen hullám akkor ér véget, ha a játékosnak megadott számú ellenfelet sikerült eltalálnia. Közben a játékosnak lehetősége van gyógyszert és lőszert szereznie. A játékos teljesítményének függvényében a fegyver fejlesztését is lehetővé teszi a játék.

A klasszikusan felszedhető tárgyak (*powerup-ok*) mellett a játékban, az előzőekhez kinézetükben hasonló csapda jellegű elemek is vannak. A felszedhető elemek hatása így tehát lehet egyaránt pozitív vagy negatív.

A játék megvalósításának problémaköre tehát szerteágazó. Többek között meg kellett valósítani a modellbetöltést textúrázással, az irányítást, fizikát, ütközéskezelést, ellenfelek viselkedésének modellezését, fények és hangok kezelését.

3. fejezet

Komponensek

3.1. Felhasznált technológiák

Windows operációs rendszeren az egyik legnépszerűbb integrált fejlesztőkörnyezet a Microsoft Visual Studio, amelynek a 2015-ös verzióját használom. Ennek felülete a fejlesztők számára egyszerűen áttekinthető, kényelmesen elérhetővé teszi a szükséges funkciókat. A játék megírásához a C++ programozási nyelv tűnt a legmegfelelőbbnek. A mai játékok jó részét szintén ezen a nyelven írják, mivel a fordítást követően natívan fut a program. A játékok esetében általában hatékonyabb erőforrás kihasználást tesz lehetővé.

A képi világ kirajzolásához, illetve a hangok megszólaltatásához az SDL 2.0-t választottam. A [4]-es hivatkozáson található bővebb leírás, hogy mit is takar pontosan az SDL.

Ez egy olyan alkalmazás, vagy játékfejlesztési segédeszköz, amelynek komponenseit felhasználva platformfüggetlenül lefordítható a forráskód. Működik Linux-on, Windows-on, Mac OSX-en, illetve bármely egyéb operációs rendszeren.

3.2. Az alkalmazás fő részei

A fő részek, és azok kapcsolatainak szemléltetéséhez egy osztálydiagramot készítettem (3.1 ábra.). Ezen megtekinthető a játék felépítése, melyik osztálynak melyik az őse, illetve mely osztályok vannak kapcsolatban egymással. Továbbá feltüntetésre kerültek a kapcsolatokhoz szükséges adattagok, metódusok.

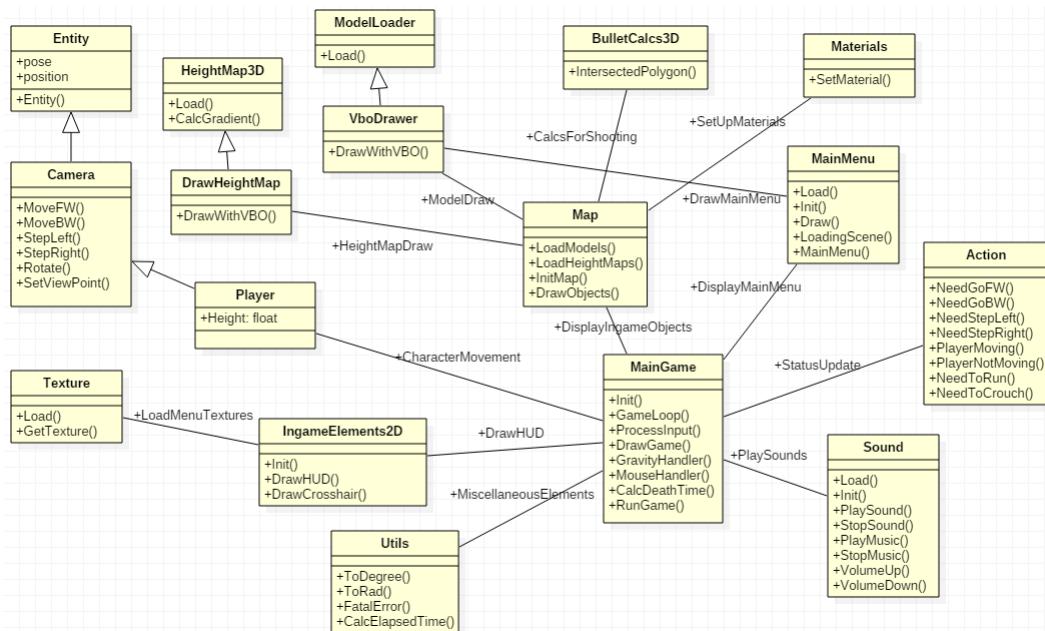
A **MainGame** a játék fő osztálya. Ehhez kapcsolódik minden egyéb komponens, és ez felel az ablak inicializálásáért, felhasználói interakciók kezeléséért. A játékbeli interakciók kezeléséért az **Action** osztály felelős, ez kezeli az állapotokat, például hogy éppen előre vagy hátra haladunk, ugrunk, vagy futunk. Ennek megfelelően az **Action** osztály kimenetei alapján a **Player** osztály végzi el a játékoshoz kapcsolódó cselekvéseket. A **MainGame**-hez kapcsolódik még közvetlen a **Map**, a **Sound**, a **MainMenu** osztály.

A Map a játéktér elemeit, és az azokhoz kapcsolódó objektumokat fogja össze. Ide kapcsolódik a HeightMapDrawer, amin keresztül betölти, kirajzolja a játék alapját képző domborzatot, és kiszámolja a karakter ütközésvizsgálatához szükséges gradienst is. A VboDrawer és őse, a ModelLoader az egyéb játékbeli objektumok (fák, tárgyak) betöltéséért, kirajzolásáért felelős. A BulletCalcs3D a lövéshez kapcsolódó háttérszámításokat végzi, és az adott háromszöggel kapcsolatos x,y,z metszéspontot adja vissza. A Materials osztály az anyagjellemzők beállításáért felelős.

A játék részét képezik a játékos informálásáért és segítéséért felelős 2D-s elemek is, amelyekért az `IngameElements2D` osztály felelős. Ez jeleníti meg a célkeresztet, rajzolja ki az életerő és a lőszer mennyiséget jelző textúrát is, amelyeket viszont a `Texture` osztály tölt be.

A `MainMenu` osztályban a főmenühöz kapcsolódó elemek vannak, amelyek a menü háttere, illetve a kijelölések lekezelése vizuálisan. Ezeket az elemeket is a `VboDrawer` és öse a `ModelLoader` tölti be, rajzolja ki.

A Sound a hangok megszólalásáért, a megfelelő zenék lejátszásáért felelős, az Utils pedig az egyéb, többi osztályba nem tartozó elemeket tartalmazza.



3.1. ábra. A játék felépítése

3.2.1. Magasságmező töltése, textúrázása

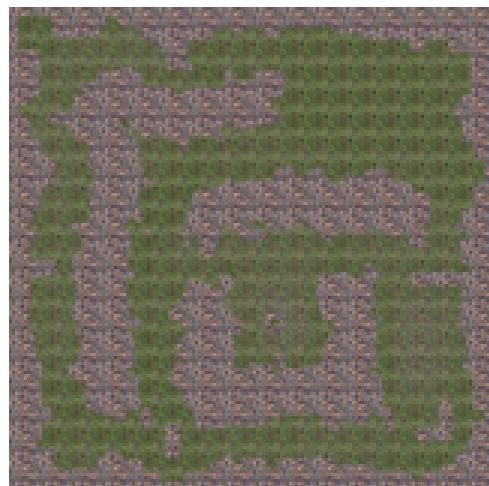
A játék .obj kiterjesztésű modelleket használ, és ezekre húzza rá a textúrát. A megjelenítéshez VBO-t, azaz *Vertex Buffer Object*-et használ, ami a modell adatait tárolja, és egyben, a legmegfelelőbb formátumban adja át a videókártyának.

Ez egy fekete fehér kép 256 színárnyalatából tud 256 különböző magasságértéket számolni. A fekete a legalacsonyabb, a fehér a legmagasabb terület, így könnyedén

lehet hegyeket illetve völgyeket kialakítani. A játék játszható területének megadásához például a 3.2 ábrán látható képet adhatjuk meg, amiből a program a magasságértékeket számolja. A magasságmező textúráját egy külön képfájlból tölthetjük be. Egy ilyen látható a 3.3 ábrán.



3.2. ábra. A magasságmező adatait tartalmazó bitmap



3.3. ábra. A magasságmező textúrája

A 3.4 képen pedig a játékban való megjelenés látható. A magasságmező használata abból a szempontból is előnyös, hogy egyszerűen megoldható vele a függőleges irányú ütközésvizsgálat, illetve az egyes részek meredekségéből közvetlenül kiszámíthatóak a bejárható területek. A magasságmező pontokból való előállításához az egyik legegyszerűbb megoldást a lineáris interpoláció adta. A textúra illesztése a nagy abszolút értékű gradiensek esetében problémás. Erre megoldást például a multitextúrázás jelenthet.



3.4. ábra. A megjelenített magasságmező

3.2.2. A karakter irányítása és a játék fizikája

A játékban egy karaktert irányíthatunk az ő szemszögéből. A karakternek van magassága, térbeli pozíciója, nézési iránya. A nézeti iránynak megfelelő térrész láthatjuk a játékos szemszögéből. A látható térrész alsó részén látszik a karakterhez tartozó fegyver, illetve annak a keze.

3.2.3. Hangok betöltése és lejátszása

Egyidejűleg több hang lejátszására is szükség lehet, amelyet valamilyen események váltanak ki. A karakter előrehaladása közben egyidejűleg ha lövünk, az már több hangcsatornát igényel. Ha csak egy csatorna lenne, akkor a lépéshang lejátszása után, vagy azt megszakítva lehetne csak lejátszani a lövés hangját. A játék alatti zenéhez szintén szükséges egy külön csatorna.

3.2.4. Lövéssel kapcsolatos számítások

A lövés háttérszámításait egy külön komponens végzi. A számítások elvégzésére minden képkocka renderelése előtt szükség van. Ennek első lépése, hogy kiszámítjuk azt a vektort, amerre a játékos néz. Mivel a világ háromszögekből épül fel, ezért ki kell számolni az egyenes háromszöggel való metszéspontját. Bemeneti adatként tehát a játékteret alkotó háromszögek vannak benne, a játékos irányvektora, kimenetként pedig azon (x, y, z) koordinátákat várjuk, amely pontban az ütközés bekövetkezett, illetve plusz információként a metszett objektumot is tudnunk kell azonosítani.



3.5. ábra. A főmenü

3.2.5. Mesterséges intelligencia

Mivel nem egy elsősorban online játszható, FPS játékról van szó, így mindenkiéppen szükséges egy, az ellenfelek mozgását irányító mesterséges intelligencia. Az ellenfelek mozgásához az útvonalak keresése A* algoritmus segítségével történik. A játék véletlenszerűen, különböző helyekre kirak adott mennyiségű ellenfelet, amiknek közeledni kell a játékos felé, különböző kritériumoknak megfelelve. Ezek a kritériumok azért kellenek, mert a pályán vannak játékelemek, amiken nem lehet átmenni, illetve az ellenfelek sem mehetnek egymásba.

Az alapötlet az, hogy a pályán le lesznek rakva pontok (úgynevezett *waypointok*), amelyek csak az ellenfelek számára lesznek láthatók. Ha kikerül egy adott ellenfél a pályára, az első feladata az lesz, hogy megkeresse a hozzá legközelebb eső waypointot. Ezután az A* algoritmus segítségével meghatározza a játékoshoz legközelebb eső waypointhoz vezető legrövidebb utat, végigmegy rajta, majd ha elérte, akkor onnantól a játékos lesz a közvetlen elsődleges célpontja.

Bemeneti adatként a waypointok állnak rendelkezésre, kimenetként pedig az útvonalat várjuk, amerre az ellenfelek mozogni szeretnének. Az ellenfél elsődleges célpontja a játékos, de adódhatnak olyan helyzetek amikor előtte egyéb dolgokat helyez előtérbe, mint például az életerő töltés, vagy lőszer felvétel. Itt fontos szerepet kap az ellenfél karakterisztikája, ami szintén bemeneti paraméter.

3.3. A játékindítás folyamata

A játék indításának folyamata a 3.6. ábrán látható.

Első lépésben a játék létrehozza az SDL ablakot, és konkrétan beállítja annak pozícióját, függőleges és vízszintes felbontását, illetve a teljes képernyős módöt.

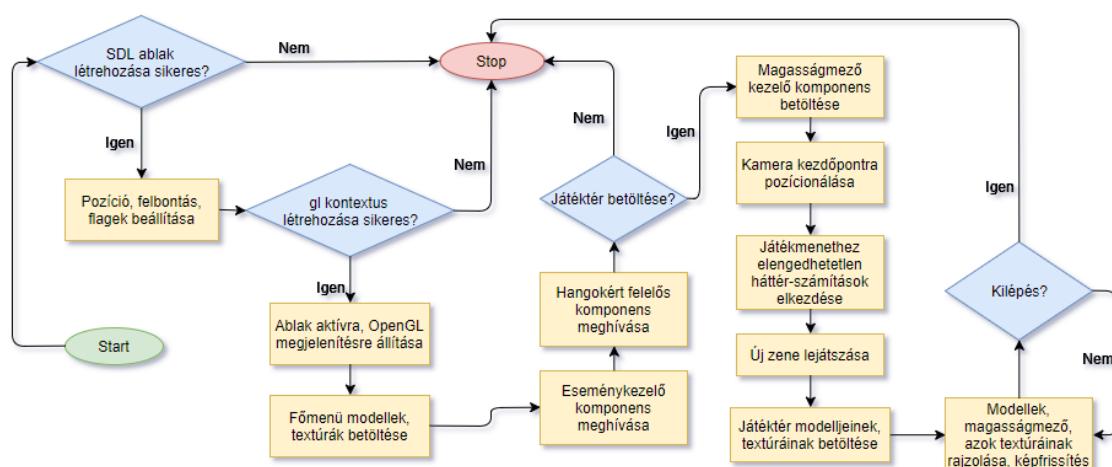
A második lépés hogy az előtte létrehozott ablakot beállítja aktívra, és OpenGL megjelenítésre. Létre kell hozni egy érvényes OpenGL kirajzolás kontextust, hogy inicializáljuk a belépési pontokat. Ez használhatóvá teszi az összes elérhető funkciót, amit az OpenGL magban definiáltak. Ezután betölti a főmenü megjelenítéséhez szükséges modellek, textúrákat.

Harmadik lépésben aktiválja a hangok megszólaltatásáért felelős komponenst, és betölti az összes hangot, zenét.

A negyedik lépésben aktiválja az eseménykezelő komponenst, ami a felhasználótól érkező interakciókat kezeli, majd meghívja a betöltött modelleket kirajzolásra.

A menübe érkezés után a felhasználónak lehetősége nyílik a játékteret betölteni, vagy kilépni.

A „Start” gombra kattintás után, betöltődik a magasságmező kezelő komponens, a kamera a kezdőpontra pozicionálódik, elkezdődnek a játékmenethez elengedhetetlen háttérszámítások és egy új zene lejátszása, betöltődnek a játéktér modelljei, elemei, átvált azok kirajzolására, illetve elrejti az egérkurzort. Hogy a játékos informálva legyen, ez idő alatt megjelenik a betöltőképernyő.



3.6. ábra. A játék indításának folyamatábrája

4. fejezet

Ütközésvizsgálat

Ebben a fejezetben az ütközésvizsgálattal kapcsolatos számításokról, optimalizációról lesz szó. Ütközésvizsgálatra több szempontból is szükség van. Egyszerűen, vizsgálni kell, hogy a játékos a játszható területen belül van-e, azaz nem lehet át falakon, tereptárgyakon, nem lehet fel túl meredek emelkedőn. Másrészt, vizsgálni kell, hogy a játékos által leadott lövedék eltalálja-e a domborzatot, tereptárgyakat, jelezni kell a lövedék becsapódását. Harmadrészt regisztrálni kell az ellenfeleket eltaláló leadott lövéseket. A második és harmadik pont nagyon hasonló, de mégis célszerűnek tűnt külön kezelni. A magasságmező és az egyéb modellek ütközésvizsgálatának módját azért érdemes külön kezelní, mert a statikus magasságmező esetében másfajta optimalizálási módok állna rendelkezésre, mint az egyéb tereptárgyak, és animált karakterek esetében.

4.1. A karakterek ütközésvizsgálata mozgás szempontjából

Ez a játék szempontjából az egyik legfontosabb elem, mivel ez adja a virtuális világ egyfajta realitását. Az, hogy kirajzoltatunk valamit a képernyőre, még nem jelenti azt, hogy azon nem lehet áthaladni. Konkrétan a Z-buffer felelős azért, hogy a takarási feladatot megoldja, viszont azt úgy képes végrehajtani, hogy közben a kirajzolt objektumok ütközését a lehető legegyszerűbb módon kezeli csak.

A kirajzolás csak a vizualitást adja, a domborzat, a tereptárgyak, a karakterek kinézetét. A játékfejlesztő feladata az, hogy megírja külön az ezekhez szükséges ütközésvizsgálatot. Mivel ez két különálló dolog, egyes helyzetekben adódhatnak olyan problémák, hogy ezek nincsenek szinkronban, tehát látunk valamit amin át lehet menni, vagy nem látunk valamit és mégis megakadunk benne.

A szakdolgozat készítése közben fejlesztett játékmotorban a karakterek és a magasságmező ütközését magasságpontok és az ezek különbségeiből kapott gradiensek számításával oldottam meg. Ez azt jelenti, hogy ha két, egymás mellett lévő pont magasságérteke között túl nagy a különbség pozitív irányba, az falat, vagy túl meredek

emelkedőt jelent. Ez a különbség tehát az eredeti felület pontbeli gradiensének numerikus közelítését adja. Ha mérsékelt, vagy nagyon minimális a különbség, akkor arról vagy lecsúszik, vagy csak egyszerűen át lehet ott haladni. Mindezt a beolvasott magasságmezőből számolja, így az üközésvizsgálat minden új magasságmező esetében automatikusan elérhető.

Jelölje H azt a két változós valós függvényt, amelyet a magasságmezővel közelítünk. Legyen a \tilde{H} az az interpolációs függvény, amelyeket az interpoláció alappontjaiban felveszik a H értékét. Egy $p \in \mathbb{R}^2$ pozíció esetén a pontbeli gradienst a következő módon közelíthetjük:

$$\nabla H(p) = \left(\frac{\partial H(p)}{\partial x}, \frac{\partial H(p)}{\partial y} \right) \approx \left(\frac{\tilde{H}(p + \delta_x) - \tilde{H}(p)}{\delta_x}, \frac{\tilde{H}(p + \delta_y) - \tilde{H}(p)}{\delta_y} \right),$$

ahol δ_x és δ_y az x és y tengely szerinti tetszőlegesen kis eltolást jelöli.

A gradiens közelítésének ez az egyik legegyszerűbb módja. Előnyös, mivel az interpolált felület pontbeli értékeit egyébként ki kell számítani, illetve a számítási pontosság megfelelő a játékmotor működéséhez.

Ezen felül a gravitáció ami nagyon lényeges, mert a pályának vannak olyan magaslati pontjai, amelyekre fel lehet jutni. Ha egy ilyenre felmegyünk, és nincs semmilyen erő, ami a karaktert a föld felé húzná, akkor felfelé ugyan lekövetné a talajmagasságot, de le már nem tudna esni.

A játékmotor fizikájában egy lefelé ható erő folyamatosan hat a játékosra és ellenfelekre, aminek a küszöbértéke mindenkor az aktuális talajmagasság. Ugrás esetén a gravitációnál nagyobb, ellenirányú erőt fejt ki a karakter, amely folyamatosan csökken. Ez eredményezi azt, hogy elemelkedik a földtől, egy ponton megáll, majd vissza is esik a talajra. Ez látható a 4.1-es ábrán. A gravitációs erőt az F_g jelöli, a kifejtett erőt pedig az F_k .

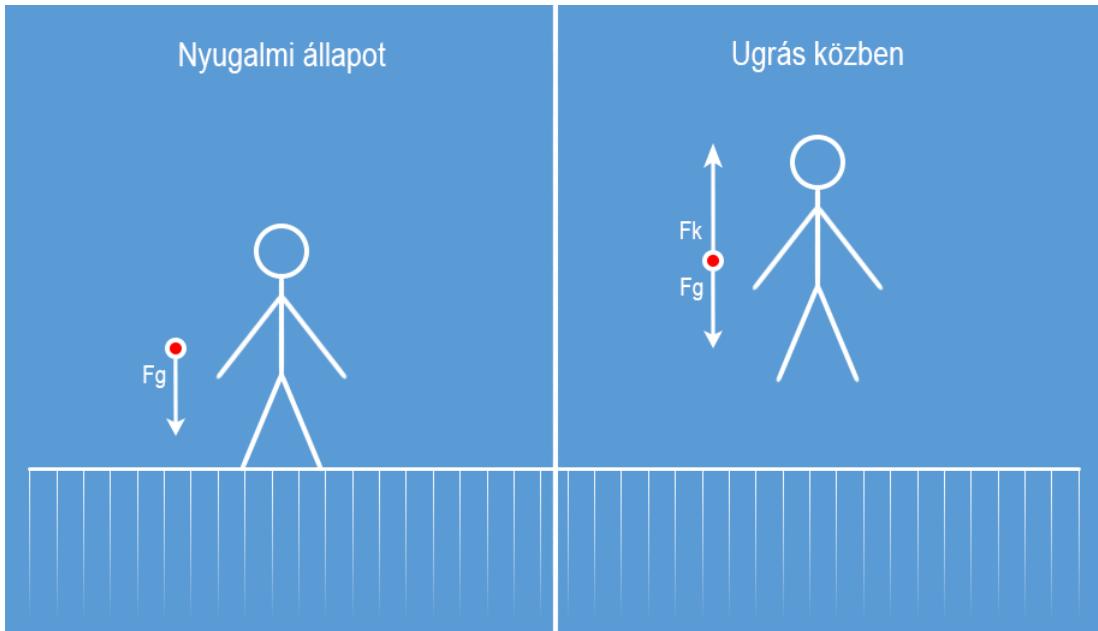
A karakter függőleges pozícióját jelöljük z -vel. Amennyiben a játékos a magasságmezőn áll, akkor teljesül, hogy $\tilde{H}(p) = z$. Az F_g gravitációs erőt konstansnak tekintjük. Ennek tipikusan csak z tengely irányú, 0-tól különböző komponense van. Amennyiben a karakter a magasságmező felett van, úgy a z és az F_k is felírható az idő függvényében t szerint. Egy Δt mintavételezési időt feltételezve (ami tipikusan a képkockák kirajzolásai között eltelt idő) a következő összefüggéseket írhatjuk fel:

$$F_k(t + \Delta t) = F_k(t) + F_g \cdot \Delta t,$$

$$z(t + \Delta t) = z(t) + F_k(t) \cdot \Delta t.$$

Amikor tehát a karakter a magasságmezőre érkezik, akkor az F_k értékét 0-nak tekintjük, a z -t pedig a magasságmező adott pontjának értékére állíthatjuk.

A koordinátarendszer és az erők mértékegységének megfelelően szükségünk lehet korrekciós szorzóra, például ha méterben és másodpercben szeretnénk számolni.



4.1. ábra. A karakterre ható erők

4.2. A domborzat és tereptárgyak ütközésvizsgálata a lövés szempontjából

A domborzattal való ütközésvizsgálat azért fontos, hogy azon keresztül ne lehessen eltalálni az ellenfeleket, illetve ezzel is realisztikusabbá tehetjük a játékot. A nagy poligonszám miatt ez is optimalizációt igényel.

A problémát az egyenes és a háromszög metszéspontjának számítására vezethetjük vissza. A domborzat háromszögeit fekete szegélyekkel láthatjuk a 4.2 ábrán.

4.2.1. Program: Lövedék ütközésvizsgálata

Az Ütközésvizsgálat nevű program mutatja be az implementációját. Első lépésként, meg kell határozni azt az egyenest, amely azt írja le, amerre néz az adott karakter. Az egyenes megadható két pontjával. Az egyik az adott karakter pozíójával egyezik meg. A másik, távoli pontot jelen esetben egy, az OpenGL-ben definiált függvényből, a GluLookAt függvényből határozzuk meg, aminek 9 paramétere van. Az első három a karakter jelenlegi pozíciója, a második három azt a pontot írja le merre nézünk (referencia pont), az utolsó három pedig azt határozza meg, hogy melyik tengelyen, és merre néz a felfelé vektor. A 9 paraméter közül a középső háromból lehet a második pontot meghatározni.

Második lépésként ki kell számolni ennek az egyenesnek a háromszögekkel vett metszéspontját. A háromszögeket a három csúcspontjukkal írjuk le. Kell lennie egy függvénynek, amelynek a háromszög három csúcspontját, és az egyenes két végpontját átadva, visszakapjuk a metszéspont (x, y, z) koordinátáját. A számítás eredményének



4.2. ábra. A térkép geometriájának határvonalai

egy szemléltetését a 4.3 ábrán láthatjuk.

A számításhoz a függvény először megvizsgálja, hogy azt a síkot, amelyen a háromszög van, metszi-e az egyenes, ha nem, nem is számol tovább. Ha metszi a síkot, akkor megnézi, pontosan hol metszi azt, melyek a metszéspont (x, y, z) koordinátái. Ez után már csak azt kell vizsgálni, hogy a metszéspont benne van-e a három csúcspontjával leírt háromszögben. Ha igen, akkor visszaadja a metszéspontot.

4.3. A játékos, és az ellenfelek ütközésének vizsgálata

Ennek a működése hasonló, mint a domborzat és tereptárgyak ütközésvizsgálatánál, azzal a különbséggel, hogy itt azt kell vizsgálni, hogy az ellenfelek körül lévő, a játékosok számára láthatatlan hengerrel van-e metszéspontja az adott egyenesnek. Ezt a hengert nevezzük jelen esetben *hitbox*-nak. Ennek a megválasztása hatással van egyrészt az ütközésvizsgálat pontosságára, másrészt pedig a számítási időre.

A hitbox arra szolgál, hogy egy közelítő becslésünk legyen az ütközésvizsgálathoz. Amennyiben eltaláltuk egy objektum *hitbox*-át, úgy meg kell vizsgálni, hogy a benne lévő modellel van-e ütközés. Amennyiben nincs, ugyanazt kell tennünk, mint ha a közelítő alakzattal sem lett volna ütközés.

Online lövöldzős játékoknál további problémát jelenthet a hálózati kommunikációból eredő hosszú válaszidő. Ebben az esetben az ellenfél játékosok nem minden esetben azon a pozícióon vannak, mint ahol látjuk őket, mivel a számunkra megjelenített helyüket a játékmotor lokálisan becsli. Amennyiben a becslés rossz volt (az ellenfél játékos más irányba mozdult el, mint ahová a kliensünk azt várta volna), az általunk leadott pontos lövést érvénytelennek kell tekinteni.



4.3. ábra. Egyenes és háromszög metszéspontja vizualizációval

4.4. Optimalizálási módszerek

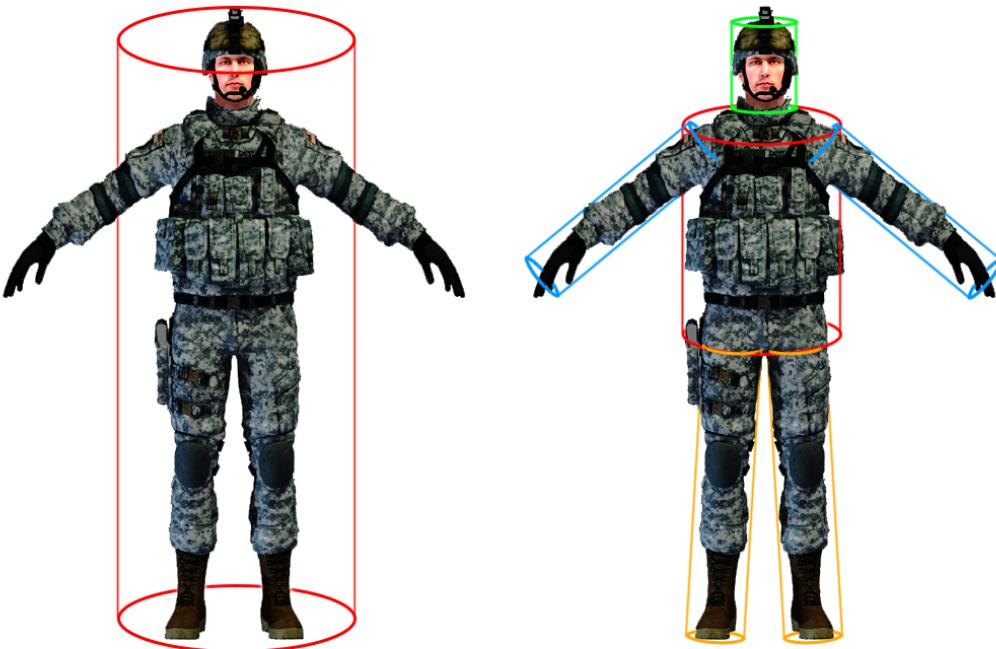
Az ütközésvizsgálatra minden képkocka kiszámítása előtt szükség van. Maga a számítás eredeti felírásában nagyon számításigényes, ezért mindenkorban szükséges, hogy ezt valahogyan optimalizáljuk. A következőkben néhány lehetséges optimalizálási módszer kerül bemutatásra.

4.4.1. A játéktér szabályos felosztása

Az optimalizálás egyik módja, hogy felosztjuk a játékteret kisebb részekre, majd ezeket olyan struktúrába rendezzük, hogy a bennük való keresés hatékonyabb legyen. Ennek egyik egyszerűbb változata, hogy egyenközű felosztást alkalmazunk. Ezt követően meg kell határoznunk, hogy melyik objektum, vagy a pályának melyik része melyik térrészben található. Egyenközű felosztásra láthatunk egy példát a 4.5. ábrán.

A karakterek ütközésvizsgálatánál elegendő, ha mindenkor csak arra a térrészre vesszük figyelembe az ütközést amelyikben épp tartózkodik az objektum, de ajánlott a körülötte levő 8 részre is. Utóbbit azért lehet szükség egyes esetekben, mert a játékos gyors mozgása esetén előfordulhat, hogy nem tölti be időben a közvetlen mellette lévő térrészt, ami problémákhöz vezethet.

A lövedék becsapódásának (x, y, z) koordinátáját is kevesebb számítással kaphatjuk meg így, mert csak azokat a térrészeket kell figyelembe venni, amelyek a játékos karakterének irányvektorával metszésben vannak. A módszer implementálása egyszerű, de mégis nagy gyorsulást tapasztalhatunk a számításokban. Hátránya viszont, hogy mindenhol egyformán osztjuk fel a teret, és lehetnek olyan részek, ahol indokolatlanul



4.4. ábra. Különböző részletességű hitbox-ok

sűrű a felosztás. Ilyen például egy nagy mező, amelyen nincsenek tereptárgyak vagy egyéb objektumok. Azért, hogy a felosztott térrészeken való keresés hatékonyabb legyen, a felosztást egy fa struktúra szerint tesszük meg. A felosztásban szereplő régiók számának logaritmusával lesz majd arányos a keresés lépéseinak a száma.

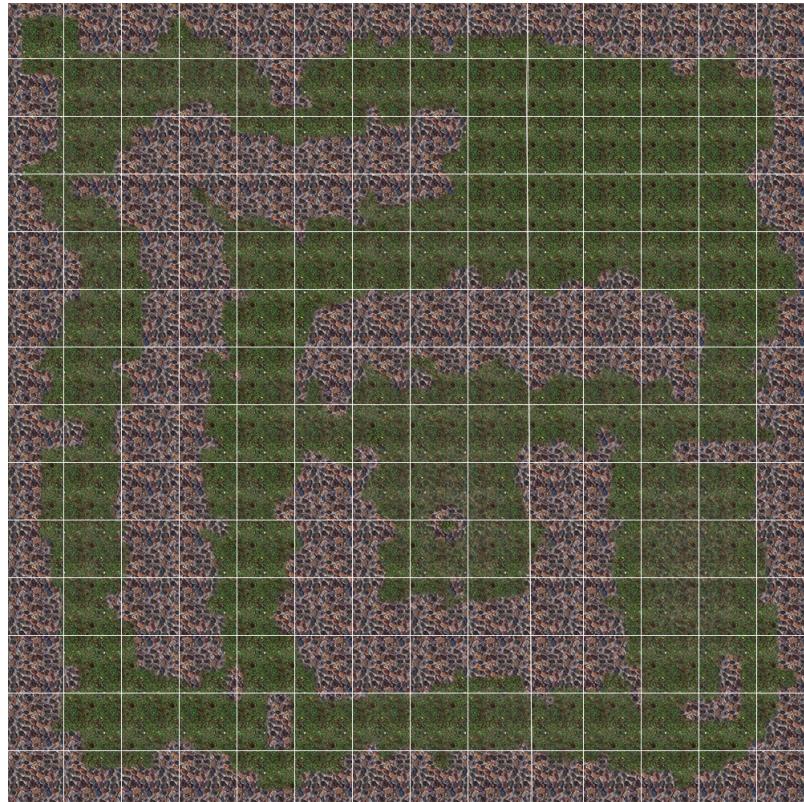
4.4.2. Négyes fa és az oktális fa

A négyes- és az oktális fa hasonló módszereket takarnak. A négyes fát síkbeli, míg az oktális fát térbeli felosztásra használják. Olyan, kivételes esetekben használható jól a négyes fa térben is, ha a játéktér nem nagy magassággal rendelkezik [5].

Használatánál az első feladatunk a fa gyökerének meghatározása, ami a teljes játékkörteret magában foglalja. Négyes fa esetén a síkot minden 4 egyenlő részre osztjuk mindaddig, amíg a fa mélysége el nem éri a maximális, előre definiált értéket, vagy az adott cellában az objektumok száma nem lesz több, mint egy előre definiált érték. Ugyanez a megközelítés érvényesül az oktális fánál, annyi különbséggel, hogy ott 8 felé osztjuk a teret. A négyes fa felosztásainak szemléltetése a 4.6. ábrán látható.

4.4.3. A BSP fa

A BSP (*Binary Space Partitioning*) egy, a tér felezésére épülő térfelosztó módszer. Tetszőleges dimenziójú térben alkalmazható. A BSP fa minden két részre osztja a teret, viszont a vágósík nem a térrész mérete alapján felezi a teret, hanem a benne ta-



4.5. ábra. A pálya felosztása egyenlő részekre

lálható objektumok száma alapján. (Páratlan elemszám esetén az egyik térrészbe egyel több elem kerül.) Az osztás irányát a fa minden mélységi szintjén változtatja, egyszer vízszintes, egyszer függőleges a vágás. Az adott objektumig úgy jutunk el, hogy minden kiválasztjuk, hogy az éppen vizsgált objektum az osztás bal, vagy jobb oldalán van. Kilépési feltételnek, szintén a négyes fához hasonlóan, megadhatjuk a fa maximális mélységét, illetve az adott cellában szereplő maximális objektumszámot. Ideális esetben a fa leveleibe egy-egy objektum kerül.

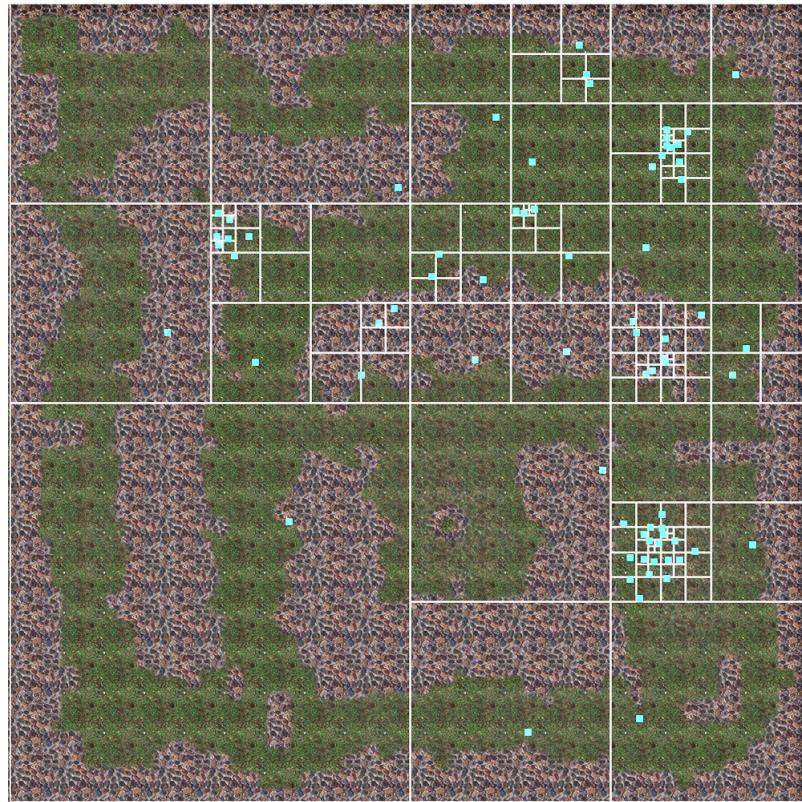
4.4.4. Optimalizálási módszer kiválasztása

A saját játékom készítése során a négyes fát alkalmaztam, hogy optimalizáljam az erőforrások felhasználását a fölösleges számítások mellőzésével. Ez a módszer felhasználható még egy raktárépület területének felosztására, hogy a keresést optimalizáljuk, így kevesebb időt vegyen igénybe.

Azért a négyes fa alkalmazására esett a választás, mert ezt egyszerűbb volt implementálni, és a térrész magassága eleve limitált, így a z összetevőnek kisebb a jelentősége.

4.4.5. Program: Térfelosztás négyes fa segítségével

A Négyesfa programon szemléltettem az eljárás implementációját. Megvalósítás során meg kell adnunk a gyökér csomópontot, ami a teljes, felosztani kívánt területet



4.6. ábra. A pálya felosztása a rajta lévő objektumok alapján

takarja. Egy csomópont x és y pozíciót, x és y hosszt, 4 darab gyerek node-ot (ha nincs, akkor az adott node értéke *nullptr*), illetve egy pont tömböt tartalmaz.

Meg kell adnunk az objektumok helyét, amelyeket figyelembe véve történik meg a felosztás. Ha az objektumok száma a meghatározott értéknél nagyobb, az adott node gyerek node-jai meghatározásra kerülnek. Ez az egész folyamat egy rekurzív algoritmus. Gyerek node területének meghatározásánál, mivel a területet 4 felé osztjuk, adott, hogy minden oldalát feleznünk kell. A for ciklusban is a terület 4 felé osztása miatt megy az i 4-ig.

Data: node; pontok;

Result: A teljes terület összes része

if *pontok* > küszöbérték **then**

for *i* := 0; *i* < 4; *i*++ **do**

 Adott node gyerek node-jának[i] := új node;

 Gyerek node területének meghatározása;

 Algoritmus rekurzív hívása;

end

end

Algorithm 1: Négyes fa területfelosztás

A program indítása után a felhasználó által megadott objektumok száma, a terület mérete, és az egy területben lévő maximális objektumszám megadása után elvégzi a felosztást. Eredményként ahogy a 4.7. ábrán is szerepel, láthatjuk a kialakult struktúrát.

```
Irja be, mennyi veletlenszeru pontot szeretne definialni!: 15
Irja be a terulet meretet!: 10
Irja be, max mennyi pont megengedett egy teruleten!: 3

Children:
    Points: 3
    Children:
        Points: 1
        Points: 1
        Children:
            Points: 1
            Points: 3
            Points: 2
            Points: 2
            Points: 1
        Points: 1
    Children:
        Points: 2
        Children:
            Points: 2
            Points: 2
            Points: 1
            Points: 2
        Points: 3
        Points: 2
    Points: 3
```

4.7. ábra. A négyes fa felbontással kialakult struktúra

5. fejezet

Útvonalkeresés

A fejezet az útvonalkeresésnél alkalmazott módszereket mutatja be.

5.1. A waypoint-ok fogalma és jellemzői

A waypoint a tér egy meghatározott pontja, amely nem látható a játékos számára. A mesterséges intelligencia ezeket használja fel az ellenfelek mozgási útvonalainak meghatározásához. Azért van rájuk szükség, mert a pályán különféle terepkadályok vannak (például falak), különböző játékelemek, amiken nem lehet átmenni, és ezekkel a pontokkal egyértelműen meg lehet határozni a bejárható helyeket.

A waypoint-okhoz rendelhetünk különféle többletinformációkat. Ilyenek lehetnek például, hogy az adott pontban a gépi játékosnak meg kell állnia, le kell guggolnia.

5.2. A waypoint-hoz tartozó adatok

A waypoint-ok pozícióját két dimenziós Descartes koordináta rendszerben (x, y) koordinátákkal adhatjuk meg. Annak ellenére, hogy térben vagyunk, nincs szükség z koordinátára, mert a talaj magassága külön kezelendő, mivel az hatással lesz az ellenfelek viselkedésére. Ennek köszönhetően minden az adott pálya talajmagasságához tud igazodni, nem szükséges azt külön megadni.

Fontos, hogy a waypoint-nak legyen egy típusa, amely meghatározza, hogy az adott pont milyen szerepet játszik. Ennek segítségével ki lehet számolni, hogy a játékban szereplő karakterek hova lehetnek.

5.3. Útvonalkeresés waypoint-ok alapján

Tegyük fel, hogy a gép létrehoz egy ellenfelet egy véletlenszerű pozícióba. (Ezt a szakzsargonban *spawn*-olásnak hívják.) Az ellenfeleknek az a fő feladatuk, hogy a lehető legrövidebb útvonalon eljussanak a játékoshoz, és megtámadják azt.

Az útvonalkeresés folyamata a következő fő lépésekkel épül fel:

- A waypoint-ok közül megkeressük az adott karakterhez vagy objektumhoz legközelebbi. Először ezt a pontot kell megközelíteni.
- A waypoint-ok közül meghatározzuk a célponthoz legközelebbi, ez lesz majd a célpont.
- A pont adott, kis sugarú környezetét elérve A*-algoritmus segítségével meghatározzuk a legrövidebb útvonalat.
- A célpontot elérve már csak az adott karakterhez vagy objektumhoz kell egyenes úton eljutni.

Data:

$S \in \mathbb{R}^2$, a kezdőpont, aktuális pozíció

$W = \{w_i | w_i \in \mathbb{R}^2\}$, a waypoint-ok halmaza

Result:

$T \in \mathbb{R}^2$, az elérődő pont,

$$T = \min_{\|w_i - S\|_2} w_i$$

Algorithm 2: A legközelebbi pont megkeresése

5.4. Az A*-algoritmus

Ezt az algoritmust a hatékonysága miatt gyakran használják gráfokban való legrövidebb útvonalak kereséséhez. Bemenetként a gráf két pontját adjuk meg, eredményül pedig az ezek között lévő legrövidebb útvonalat várjuk.

A gyakorlati problémák jelentős részében nem közvetlenül egy gráf áll a rendelkezésünkre. Ahhoz, hogy az A*-algoritmust mégis használni tudjuk, a bejárando területet (amelyben a legrövidebb útvonalat keressük) egy ráccsal közelíthetjük. Ez a rácso kialakítás több szempontból is előnyös. A pontok minden esetben egymástól egyenlő távolságra helyezkednek el, így ezeket nem kell számolni. Továbbá ha csak a bejárható helyekre helyeznénk pontokat, külön olyan algoritmusra lenne szükség, amely meghatározza azt, hogy melyik pontból melyikbe haladhatunk. A gráf összes csomópontjának és élének a tárolására szükség lenne, amely indokolatlanul sok adat kezelését jelentené. Jelen esetben egy egyenközű rácst alkalmazva a kereséshez használt gráphoz, elegendő csak megadni a rácspontokban, hogy az adott pont bejárható vagy nem [6].

Négyzetrács esetében előtörendő kérdés, hogy az átlók mentén haladhatunk-e. Jelen esetben az tűnt megfelelőbbnek, hogy az átlók mentén is haladhatunk, ugyanis a játéktéren nem csak egymásra párhuzamos, és merőleges falak vannak, a domborzat lehet bármilyen elrendezésű.

Az algoritmus végrehajtásához listát kell vezetni azon pontokról amiket még nem jártunk be, illetve azokról is, amelyeket bejártunk, ezeket hívjuk nyitott (Open) illetve

zárt (Closed) csomópont listáknak. A nyitott csomópontok listája az, amelyek vizsgálatára még a későbbiekben szükség lehet, a zárt csomópont lista pedig a már vizsgált pontok halmaza [7].

A csomópontokhoz tárolni kell az addig bezárt távolságot, illetve le kell tárolni egy prioritás változót is, amiben az eddig megtett út, és a még hátralevő út távolságainak összege található. Fontos ugyanis az algoritmus működése szempontjából az eddig megtett, és a hátralevő út, mert ezekből az adatokból számolja ki egy adott pont prioritását (a kezdőponttól való távolságának és a célponttól való távolságának összegét), és így dönti el milyen pontokat részesítsen előnyben. A prioritás minél kisebb érték, annál jobb, tehát arra kell tovább indulni [8].

5.5. Program: Útvonalkeresés A*-algoritmussal

Az Útvonalkeresés nevű program célja, hogy bemutassa az A*-algoritmus egy lehetséges alkalmazását, amely felhasználható egyéb területeken is. Áruszállítás, vagy például mobil robotok útvonaltervezése esetén meghatározhatjuk a legrövidebb útvonalat A -tól B pontig, akár mozgó B pont esetén is [14].

A program a játéktér alapját képző domborzatot egy képből olvassa be. Ez a magasságmező, ahol egy adott pont magasságát a világossága határozza meg, minél világosabb, annál magasabb pontot jelent a domborzaton, így 256 különböző magasság lehetséges, amely az aktuális változathoz megfelelő felbontásnak bizonyult.

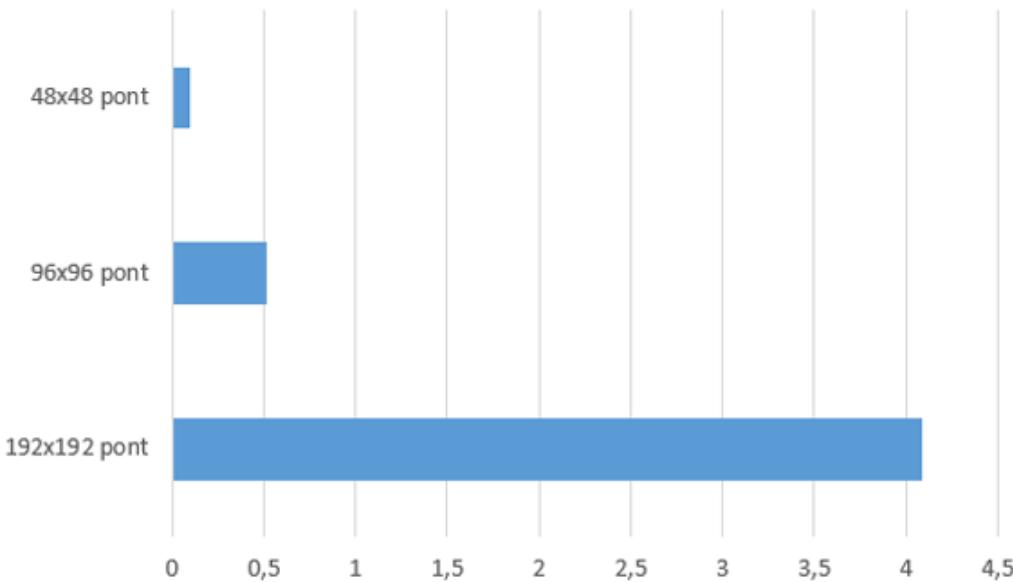
A játékteret úgy érdemes kialakítani, hogy jelentős része (körülbelül 90%) bejárható legyen.

A magasságmezőhöz tartozó kép felbontása 384×384 pixel, amely a térkép megfelelő részletességű megjelenítése miatt ennyi. Ez egy olyan felbontás, amely alatt már nagyon látszik a geometria gyenge részletessége a jelenlegi méret mellett, viszont a nagyobb felbontás már csak nagyon kis mértékben befolyásolná a kinézetet, ellenben növelné az erőforrásigényt.

A waypoint-ok meghatározására több lehetőség is van. Az egyik lehetőség, ha minden pixelhez rendelünk egy pontot. Ez olyan szempontból lehet előny, hogy pontosabban meg van határozva az összes pont amelyre léphet az ellenfél. Hátránya viszont, hogy ez nagyon sok számításigény, és fölösleges is minden pixelre egy waypoint.

A másik lehetőség, ha 8 pixelenként határozunk meg egy waypoint-ot, ami 48x48-as felbontást eredményez. Azért pont erre az értékre esett a választásom, mert ez nagyban gyorsítja a számításokat, viszont még nem megy a bejárható területek rovására, megfelelően pontos. Ez kevesebb memória igényt jelent, és mivel kevesebb adattal is kell számolni, kevésbé terheli a processzort is. Ez látható az 5.1 ábrán.

Első lépésként, létre kell hozni a waypoint hálót, amelyen a mesterséges intelligencia végig fog menni, elkerülvén hogy átmenjen a tereptárgyakon, falakon. Ezek a pontok 8 pixelenként, függetlenül az adott pixel színétől meghatározásra kerülnek, viszont a

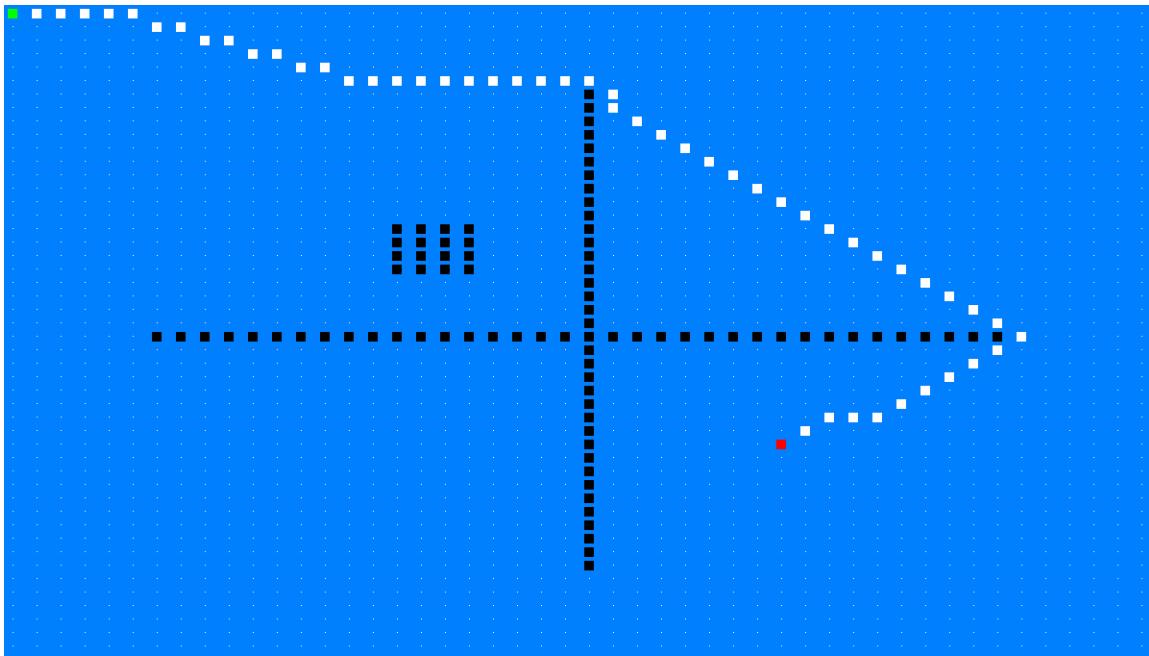


5.1. ábra. Számításigény a felbontás és idő(mp) függvényében

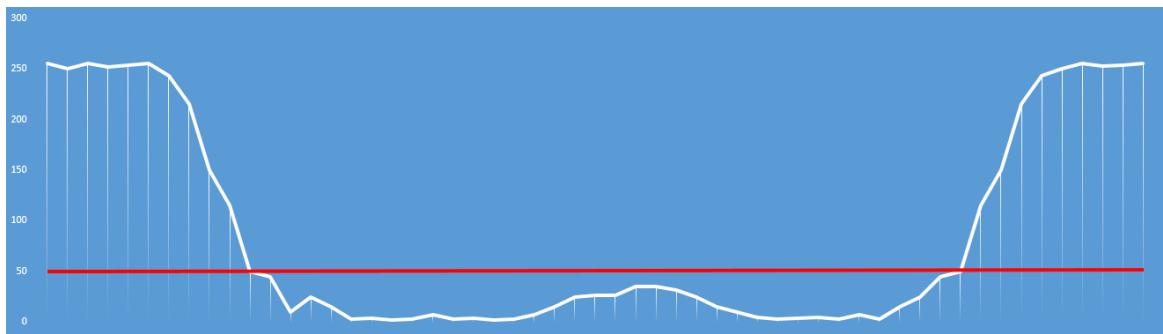
típusa az adott pixel színének megfelelően lesz beállítva. Ahogy az 5.3-as ábrán is látható, a például egy pont színéből 50-nél nagyobb érték jön ki, akkor a pont típusa zártra állítódik, így ebbe az irányba nem lesz lehetséges a továbbhaladás. Az 5.2. képen feketével a falak, zölddel a kezdőpont, pirossal a célpont, fehérrel az útvonal van jelölve.

A példaprogramban még csak statikusan vannak megadva a falak, de mint ahogy fentebb olvasható, a játékon belül a beolvastott magasságmező pontbeli értékei határozzák meg azt. Erre azért van szükség, mert így lehet megoldani azt, hogy a mesterséges intelligencia alkalmazkodjon a környezetéhez. Ha a pályától függetlenül, statikusan lenne az összes waypoint típusa meghatározva, akkor ha a játékos más pályát rajzol, a pontok típusai nem illeszkednének az elrendezésre. Ezt a módszert alkalmazva, meg lehet adni, hogy egy bizonyos magasságnál csak másszon, vagy ugorjon, vagy bármilyen más tevékenységet végezzen. Ez egy fontos tulajdonsága a játéknak, mert azt úgy terveztem, hogy a felhasználó tetszőleges rászteres rajzprogram segítségével elkészíthesse a térképet, ne legyen hozzá szükség speciális szerkesztőszközre. A program a bejárható teret ebből képes közvetlenül meghatározni, be tudja állítani, hogy a falakon a játékos ne tudjon átmenni, a meredek lejtőkről visszacsússzon.

De a küszöbérték megadása nem mindenleges, ugyanis lehetnek olyan helyek a pályán, amelyek fal magasságúak, de bejárhatók, mert alacsony meredekségű emelkedő vezet fel oda. Ezt a problémát gradiens számítással lehet megoldani, ezzel van megoldva az is, hogy a játékos ne tudjon kimenni a játéktérből. Ahol túl nagy a meredekség, ott a waypoint típusát nem bejárhatóra kell állítani.



5.2. ábra. Útvonal meghatározó algoritmus működése



5.3. ábra. Küszöbérték a még bejárható terület meghatározásához

A pontok típusának a pálya alapján legenerálása után, az A*-algoritmus segítségével meghatározza a játék a legrövidebb útvonalat a cél waypointig, természetesen a pontok típusait figyelembe véve. Ezt másodpercenként legalább 5x újra kell számolni, ugyanis a játékos mozog, és a cél waypoint mindenkor legközelebb eső pont, ami ezzel együtt változik.

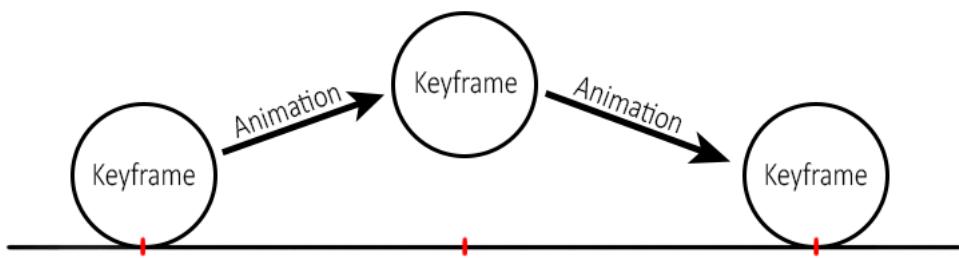
6. fejezet

Animáció

A fejezet a játékokban alkalmazott animációs módszereket, illetve azok megvalósítási módját mutatja be.

6.1. Kulcsképkocka alapú animáció

Kulcsképkocka alapú mozgatást több területen használnak, például filmek készítésénél vagy játékok fejlesztésénél. Bármilyen célra is használjuk, az elv nem változik. Meg kell adnunk olyan pontokat, amelyeken szeretnénk ha az adott tárgy, vagy objektum áthaladna. Ezzel egy előre meghatározott mozgási útvonalat adunk meg. Az egyszerűbb animációknál egy alapállapot és egy végállapot van megadva, ilyen lehet egy redőny leengedése és felhúzása, ajtó vagy kapu kinyitása, becsukása. Ilyen jellegű, alap szintű animációknál, például ha egy redőnyt le akarunk engedni, annyi a dolgunk hogy megadjuk a kezdő és végpontot, majd idő paraméter szerint interpolálni. Ahogy az a 6.1 ábrán látható, ez a kulcsképkockák közötti mozgást eredményezi.



6.1. ábra. Kulcsképkocka alapú animáció

A vertex alapú animációt a 3D játékfejlesztésben karakterek mozgatásához a 2000-es évek elejéig használták. Megvalósítása egyszerűbb, viszont sok memóriát igényel a

tárolása, és nem eredményez olyan természetes mozgást általában, mint a csontváz alapú animáció. Ezt a technológiát többek között a Quake 1, 2 és 3 alkalmazta.

6.1.1. Az md2-es formátum

Az md2-es modelltárolási formátumot az *idSoftware* fejlesztette ki 1997-ben a Quake 2 nevű játékához. A formátum kulcsképkocka animációk tárolására alkalmas. Tartalmazza az adott modell geometriáit, és a képkockánkénti animációkat.

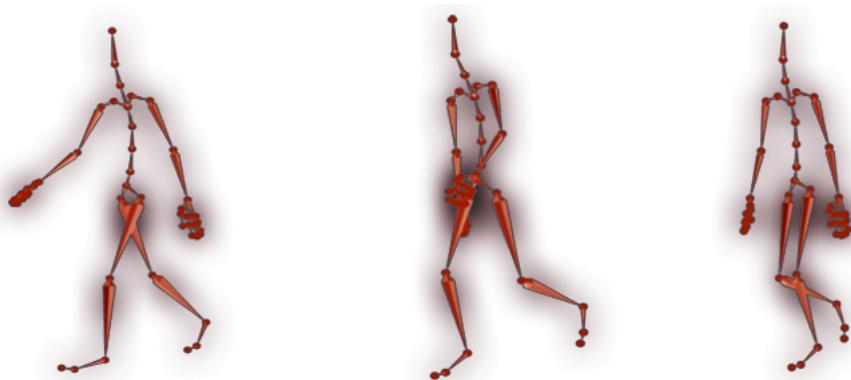
A fájl adatai olyan struktúrában helyezkednek el, hogy könnyedén kirajzoltatható legyen a GL_TRIANGLE_FAN és GL_TRIANGLE_STRIP OpenGL függvényekkel. A formátum két fő részből áll: fejlécből és adatrészvból. A fájl nem tartalmazza a modellek textúráját.

A fejléc egy struktúra, amely a fájl elején kezdődik, és tartalmazza az összes, fel-dolgozás szempontjából lényeges információt a fájl egészéről. Ilyen például a textúra szélessége, magassága, egy képkocka mérete, képkockák száma.

6.2. Csontváz alapú animáció

A legtöbb mai játék ezt a módszert alkalmazza karakterek mozgatásához. Az alapötlete az, hogy a mozgatható részeket hierarchiába szervezi. Az élőlények felépítése általában olyan, hogy ezzel a fajta animálási móddal egyszerűen modellezhetők.

A csontváz alapú animáció kapcsolódási pontokból (*joint-okból*) építkezik. Ember esetén például a kézfej kapcsolatban van az alkarral, az alkár a felkarral, ami pedig a test többi részével. minden hajlási pont egy új kapcsolódási pontot jelent. Tehát ha az ember felkarját megmozdítjuk, mozdul vele a kezének a többi része is. Ezt szemlélteti a 6.2. ábra.



6.2. ábra. Testrészek hierarchikus kapcsolódási pontjai emberen¹

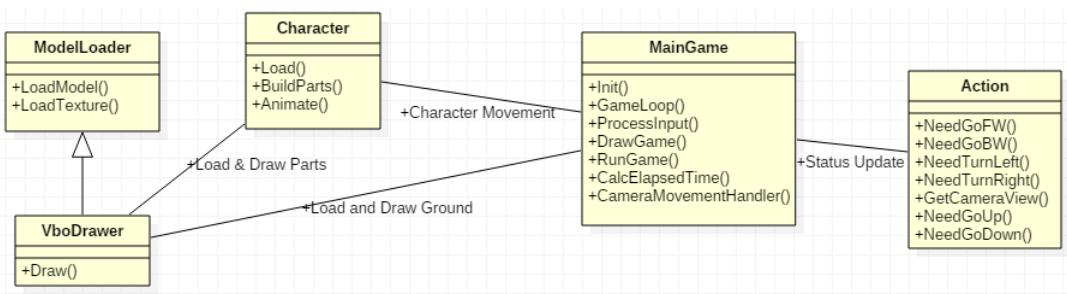
¹forrás: https://www.openflipper.org/media/plugin_images/skeletalAnimation.png

Szerencsére a módszer nem csak élőlények modellezésére alkalmazható hatékonyan. A naprendszerünk szintén modellezhető ilyen módon. Vannak a naprendszerünk bolygói, amelyek a tengelyük, és a központi csillag, a Nap körül is forognak.

Előnye, hogy könnyen létre lehet így hozni dinamikus animációkat, mivel az összes karakter csontjait lehet forgatni, mozgatni, továbbá megkönnyíti a bonyolultabb animációk elkészítését. Hátránya viszont, hogy komplexebbek, így több processzoridőt igényelnek.

6.2.1. Program: Emberszerű karakter animálása

Az Animáció nevű programnak az a célja, hogy bemutasson egy animációs módszert, amelyet bárki fel tud használni karakteranimációhoz játékfejlesztésnél. Az animációt megvalósító algoritmus hasznos lehet még egy automatizált gyártósor robotkarjainak mozgatásához is olyan esetekben, amikor például két vagy több karnak minden egy adott szög tartása mellett, összehangolva kell mozogniuk. A Character osztályban az összes mozgatással és azok pontos megjelenítésével kapcsolatos számítások találhatók. Az Action az állapotok kezeléséért, a ModelLoader és a VboDrawer a modellek (testrészek) betöltéséért, kirajzolásáért felelős, amelyeket a MainGame osztály fog össze. A konkrét felépítés a 6.3 ábrán látható.



6.3. ábra. Az animációt bemutató demó felépítése

A 6.4 ábrán látható, futó programban egy egyszerű robotot tudunk irányítani, illetve ha a kamerát szabad mozgás módba váltjuk, meg tudjuk tekinteni a testrészek hierarchikus mozgását több szögből is. A robot részei több .obj kiterjesztésű modellből vannak betöltve, amelyek azért vannak külön modellben tárolva, hogy egymástól függetlenül tudjuk őket mozgatni, ne csak egyben a teljes robotot. A törzshöz csatlakozik a fej, a két kéz, és a két láb. Utóbbi kettő minden 60 fokos szögkülönbséggel rendelkezik, és ha elérik a maximális, előre megadott kitérési értéket, a mozgás iránya megfordul. Az elérni kívánt mozgásmechanika implementálása után, meg kell adnunk a testrészek pozícióját. A Character osztályban lévő *BuildParts()* függvény a ModelLoader és a VboDrawer által betöltött és kirajzolt részek megfelelő pozícióba való elhelyezéséért felel, hogy azok egy egész robottá álljanak össze.

Járás közben az adott karakter előre is halad, illetve minimálisan fel és lefelé is

mozog a járásból adódóan, és a testrészek fix szögben való mozgatásán kívül definiálunk kell ezeket is. Figyelni kell, és számításba kell venni azt a jelenséget is, hogy járás közben egy élő ember földre leért lába fix, azzal lendíti előre magát, így a láb végéhez képest kell megadnunk a hierarchiában feljebb lévő testrészek mozgását is, ezt nevezzük inverz kinematikának. Az előre és a fel-le irányba való mozgást a következő összefüggéssel írhatjuk fel:

$$\Delta f = |d \cdot \sin(\alpha)|, \quad u = d \cdot \cos(\alpha),$$

ahol Δf az időegység alatt megtett előre mozgásnak a mértéke (*forward*), az u pedig a karakter fel-le mozgásának animálásához szükséges érték (*up*). A d az egységnyi idő alatt megtett távolságot jelöli (*distance*), az α pedig a jobbláb animálásánál figyelembe vett szög értéke.

Az animálásért felelős részek időfüggőek, ugyanis garantálni kell hogy a különböző teljesítményű gépeken ugyanolyan gyors legyen az animáció. A két kirajzolt képkocka közötti eltelt időt a `CalcElapsedTime` függvény számolja, amelynek a kimeneti értéke az $s = v \cdot t$ képlettel számolható, ahol s a megtett út, v a sebesség, t pedig az eltelt idő. Minél több a másodpercenként megjelenített képkockák száma, annál kisebb a köztük eltelt idő, és a mozgást annál kisebb értékkal szorozza be, így minden esetben ugyanolyan gyorsan fog mozogni az objektumunk, csak az animáció folytonosságában lesz különbség.



6.4. ábra. Pillanatkép az animációt bemutató demóból

A kamera alapesetben a robot mögött van, és annak mozgását követi, amelyet szintén a felhasználó tud irányítani. Az animáció bármikor félbeszakítható, nem áll vissza az alapértelmezett állásba, hogy megtekinthető legyen a mozgás minden fázisa.

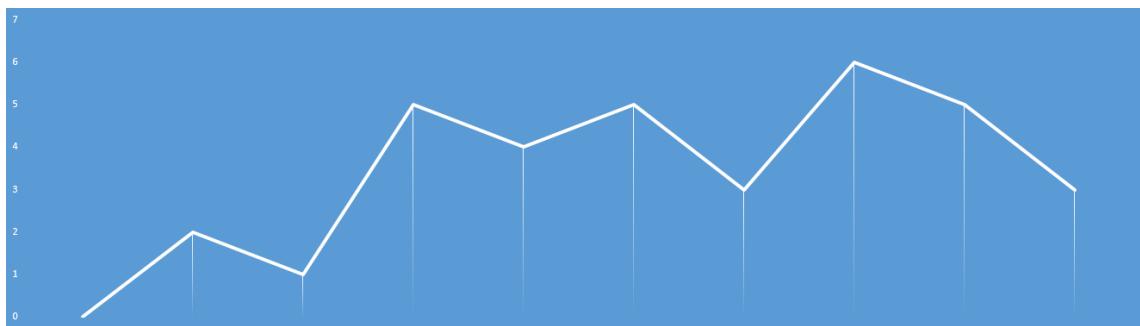
6.3. Interpolációs módszerek

Animációkészítés során többféle interpolációt használhatunk, ami azt jelenti, hogy ismert értékek alapján, a közbülső pontokra, vagyis nem ismert értékekre adunk közelítést [12].

Különféle interpolációs módszerek vannak. A választás közülük nyilván attól függ, hogy pontosan mit szeretnénk megvalósítani. Egyszerűbb esetekben még megfelelő a lineáris interpoláció, viszont általában túlságosan szögletes mozgást eredményez, ezért valamilyen simább görbével való útvonal közelítést érdemes használni.

6.3.1. Lineáris interpoláció

Ez a legegyszerűbb interpolációs módszer, minden pontot egy egyenesel kötünk össze. Nem összetett mozgásoknál ez megfelelő lehet, viszont ha több mozgás követi egymást rögtön, akkor darabos. minden előre meghatározott pont, amit érinteni kell egy törést jelent, így komplexebb, sima mozgásokhoz nem alkalmas, ez látható a 6.5. ábrán.



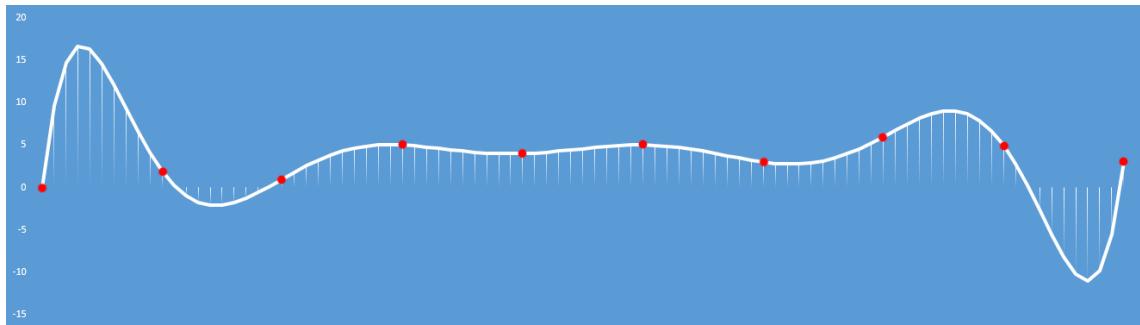
6.5. ábra. Lineáris interpoláció

6.3.2. Lagrange-interpoláció

Mivel a függvény sima, deriváltja folyontos, sokkal életszerűbb megközelítést lehet elérni vele, mint a lineáris interpolációval. Viszont nagy hátránya, hogy egyes esetekben indokolatlanul nagy hullámzások jelenhetnek meg a kirajzolt függvényen, közvetlen egymás mellett lévő két pont között, ez látható a 6.6-es ábrán. Így ez az interpoláció animációhoz nem minden esetben alkalmas.

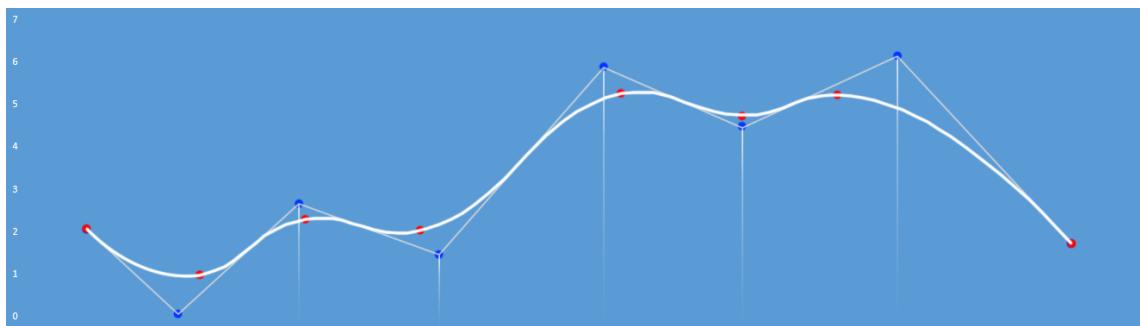
6.3.3. B-spline

Komplexebb animációknál, mint például egy élőlény mozgása, ha nem szeretnénk darabos animációt, akkor a B-spline interpolációt is alkalmazhatjuk. Ez esetben a görbe lokálisan vezérelhető, és a lineáris interpolációval ellentétben nem töredezett, így ez a legalkalmasabb folyamatos animációra. A lokális vezérelhetőség adja az előnyét a



6.6. ábra. Lagrange-interpoláció

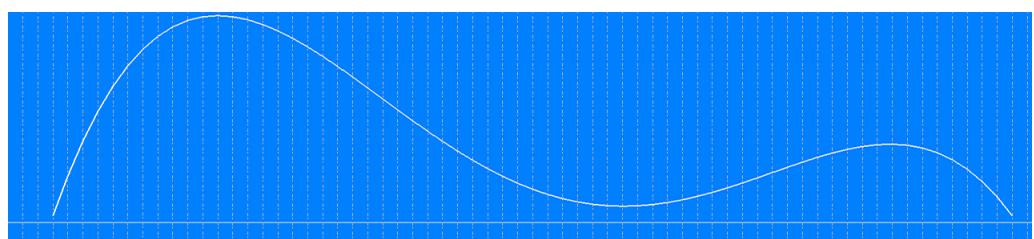
Lagrange-interpolációval szemben, mert így nem lehetnek kiugró értékek a grafikus képében. A 6.7. ábrán kék pontokkal van jelölve a lineáris interpoláció töréspontjai, és vékonyabb vonallal annak grafikus képe, hogy egyértelműen szemléltethető legyen a különbség.



6.7. ábra. B-spline interpoláció

6.3.4. Program: Lagrange Interpoláció

A felhasználónak lehetősége van megadni, hogy hány darab pontot szeretne definiálni, majd a pontok konkrét x és y koordinátáit, amelyekhez a program kiszámolja az y koordinátát Lagrange-interpolációval. Ez a program azért készült, hogy bemutassa, hogyan működik ez a fajta interpolálás. Adott két tömb, amelyek tartalmazzák a függvény megadott pontjainak x és y értékeit. A num a tömbök elemeinek számát, az input pedig a bevitt értéket takarja.



6.8. ábra. Lagrange-interpolációs demó

```

Data: x[]; y[]; num; input; s; t; eredmény;
Result: Adott x-hez tartozó y koordináta
for i := 0; i < num; i++ do
    s := 1, t := 1;
    for j := 0; j < num; j++ do
        if j != i then
            s := s * (input - x[j]);
            t := t * (x[i] - x[j]);
        end
    end
    eredmény := eredmény + ((s / t) * y[i]);
end

```

Algorithm 3: Lagrange-interpoláció implementálása

A függvény grafikus megjelenítése az OpenGL-el lett megvalósítva. Az ablak alapértelemzett felbontása 1280x720, amelyhez a bevitt értékeket a program felskálázza úgy, hogy a függvény még sehol sem lógjon ki az ablakból, de a lehető legnagyobb méretű legyen. A skálázás mértékét mind konzolban, mind grafikusan megjeleníti, utóbbit csak addig, míg a szorzó nagyobb mint 9. Ez alatt az osztás már olyan sűrű 1280x720-as felbontás mellett, hogy értelmetlen, és csúnya a grafikus kirajzolása.

A függvény kezdete a legkisebb, a vége pedig a legnagyobb x értéknél van, a köztes részen pedig minden egész x -hez kiszámolja a hozzá tartozó y -t, majd azokat vonallal összeköti. A konzol eredményei a 6.9. ábrán, a grafikus megjelenítés pedig a 6.8. ábrán látható.

```

Adja meg mennyi x es y ertekeket szeretne beirni: 5
Irja be az x es y ertekeket (x>0):
3 5 42 13 25 98 55 55 67 5
A bevitt ertekek a kovetkezok:
3      5
42     13
25     98
55     55
67     5

Ahol x = 3, a hozza tartozo y ertek: 5
Ahol x = 4, a hozza tartozo y ertek: 36.9195
:
Ahol x = 66, a hozza tartozo y ertek: 19.9641
Ahol x = 67, a hozza tartozo y ertek: 5

Az X ertekek szorzoja: 18.7224

```

6.9. ábra. Lagrange-interpolációs demó

7. fejezet

Ellenfelek viselkedésének modellezése

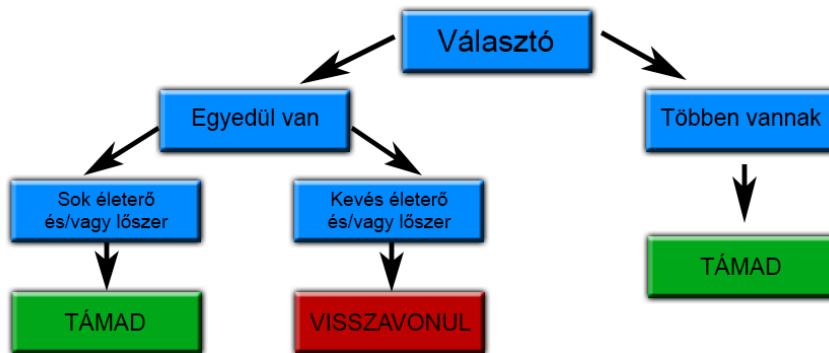
A mesterséges intelligencia útvonalkeresés részéről már szó volt a korábbiakban, viszont nagyon fontos, hogy az ellenfeleknek egyéb tulajdonságokat adjunk, ezzel érdekesebbé téve a játékot. Szüksége van minden ellenfélnek egy célra, ami alapjában véve az, hogy megtámadják a játékost, viszont bizonyos szituációkban ez megváltozik, így kapunk még élethűbb viselkedést. Ez a fejezet ezen elemek bemutatásáról szól.

7.1. Viselkedés bemenetei és kimenetei

Ahhoz hogy az ellenfelek viselkedését modellezzük, különböző bemenetekre és kiemenetekre van szükség. Ehhez referenciának nagyon jól használható a *Quake III Arena* esetében használt módszer [13].

7.1.1. Bemenetek

Bemenetként az ellenfél szemszögéből számít a játékostól való távolság, az aktuális életerő, a lőszer mennyisége, hogy a játékos benne van-e a látótérben, illetve hogy egyedül kell-e neki szembeszállni a játékossal, vagy többen vannak. Ezek a bemenetek egymástól is függhetnek. Ha a játékostól való távolság kisebb, mint egy előre definiált érték, és nem egyedül vannak, akkor közelítsenek és támadjanak, viszont ha egyedül van, akkor próbáljon menekülni. Figyelembe veheti azt is, hogy milyen hatékony fegyver van a kezében, az életereje vagy lőszere egy előre definiált érték alatt van-e, és ezek függvényében támad, keres életerőt vagy lőszert. Az ilyen típusú adatok definiálásának legjobb módja, ha az ellenféltípusokhoz hozzárendelünk különböző tulajdonságokat, így kialakíthatjuk, hogy melyik miben jó, és miben rossz. Ezt nevezzük karakterisztikának. Egy ilyen karakterisztika leírása látható a 7.2. táblázatban.



7.1. ábra. Az ellenfél egy lehetséges viselkedése

Név	Az ellenfél neve.
Támadóképesség	<p>Ellenfél képességei támadás közben.</p> <p>$> 0.0 \& < 0.15$ = Nem mozdul.</p> <p>$\geq 0.15 \& < 0.5$ = Csak előre és hátra mozog.</p> <p>$\geq 0.5 \& < 1.0$ = Körbe-körbe megy.</p> <p>$> 0.6 \& < 1.0$ = Véletlenszerű mozgás irányváltással.</p> <p>$> 0.4 \& < 1.0$ = Visszavonuláskor is fedezze magát lövessel.</p>
Agresszió	Az ellenfél agressziója.
Neme	Férfi, nő, vagy egyéb teremtmény
Célzóképesség	<p>Fegyverenként különbözőt.</p> <p>$> 0.0 \& < 0.9$ = Az ellenfél mozgása hatással van a célzásra.</p> <p>$> 0.4 \& \leq 0.8$ = Ellenfélmozgás követése.</p> <p>$> 0.8 \& \leq 1.0$ = Várható felbukkanás helyének figyelése.</p> <p>$> 0.6 \& \leq 1.0$ = Környezeti elemek felhasználása sebzésre.</p>
Guggolás	Guggolás gyakorisága.
Ugrás	Ugrás gyakorisága.
Forgás	Forgás sebessége (reflex).
Reakcióidő	Miután meglát, mennyi idő telik el az első lövésig.
LövésPontosság	0 és 1 közötti érték, a lövés szórását adja meg.
Bosszú	Az ellenfél bosszúállósága.

7.2. ábra. Egy, a Quake III esetében használt viselkedés leíráshoz hasonló szabályrendszer [13]

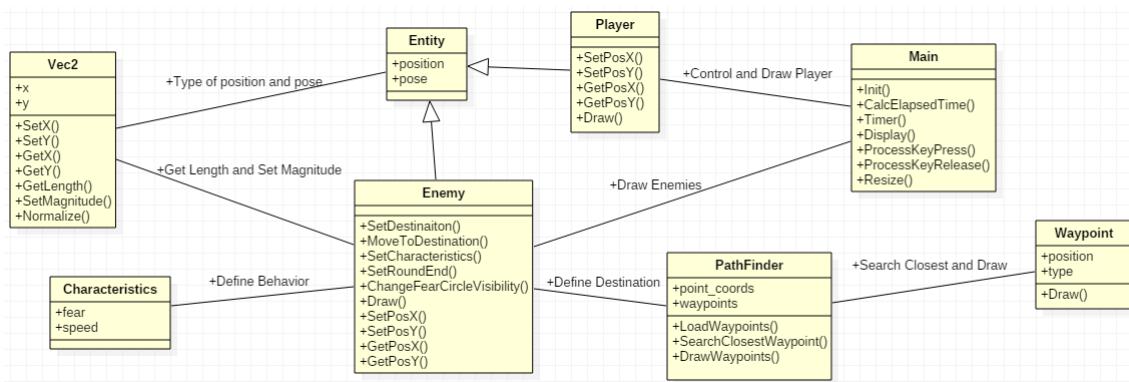
7.1.2. Kimenetek

A bemeneti adatok alapján a program egy modulja kiértékeli az eseményeket, és annak kimeneti értékei határozzák meg a következményeket. minden kimenet egy végrehajtható akciót jelöl (támadás, menekülés, életerő vagy lőszer töltés). A döntési fa az csak az állapotot határozza meg. minden iterációban ki kell értékelni, és minden lehetséges esethez meg kell írni azt a kód részt, ami az adott cselekvésfolyamatot elindítja. A 7.1-es ábrán kékkel vannak jelölve a bemeneti paraméterek, zölddel és pirossal pedig azok következményei, vagyis a kimenetek.

7.1.3. Program: Gépi ellenfél viselkedésének modellezése

A Viselkedés nevű program elkészítésével az volt a célom, hogy minél egyszerűbben, vizuálisan lehessen bemutatni az ellenfelek viselkedését. A termelésinformatikában ennek az algoritmusnak a segítségével különböző döntési tényezőket lehet definiálni, hogy a robot azoknak megfelelően cselekedjen. Jelen esetben a játékost fehér, az ellenfeleket piros, a biztonságos pontokat kék négyzet jelöli. A biztonságos pontok azok a helyek, ahova egy ellenfél vissza tud vonulni, ha a játékos megmozdul. minden ellenfél a korábban leírtaknak megfelelően rendelkezik egy karakterisztikával, amely minden esetben két tulajdonságából áll, a sebességből, és a félelemfaktorból. Utóbbi azt határozza meg, hogy ha megmozdul a játékos, mekkora legyen az a sugár, amelybe ha egy biztonságos pont beleesik, visszavonuljon. Ha több pont esik az adott sugárba, akkor azok közül is a legközelebbit választja.

A félelemfaktor függ az ellenfél sebességtől is, tehát gyors ellenfél esetén csak alacsony értékeket, lassú ellenfél esetén pedig csak magas értékeket vehet fel, ezzel életszerűbbé teszi a viselkedést. A mintaprogram felépítése a 7.3 ábrán látható.



7.3. ábra. A viselkedést bemutató program felépítése

A 7.4 ábrán látható módon, az elindítás után minden ellenfélnek véletlenszerűen generál egy karakterisztikát, olyan módon, hogy a sebességet és a félelemfaktort beállítja egy adott értékre. A felhasználónak lehetősége van ezeket újragenerálni, illetve minden ellenfélnek a félelemfaktorát vizuálisan megjeleníteni, amely a 7.5 ábrán látható.

```
NEW ROUND!
New Enemy (number 1):
    Fear: 75.000000, speed: 4.000000
New Enemy (number 2):
    Fear: 53.000000, speed: 4.000000
New Enemy (number 3):
    Fear: 34.000000, speed: 6.000000
New Enemy (number 4):
    Fear: 118.000000, speed: 3.000000
New Enemy (number 5):
    Fear: 36.000000, speed: 7.000000
NEW ROUND!
New Enemy (number 1):
    Fear: 144.000000, speed: 2.000000
New Enemy (number 2):
    Fear: 55.000000, speed: 5.000000
New Enemy (number 3):
    Fear: 84.000000, speed: 5.000000
New Enemy (number 4):
    Fear: 48.000000, speed: 7.000000
New Enemy (number 5):
    Fear: 43.000000, speed: 7.000000
```

7.4. ábra. Az ellenfelek félelemfaktora

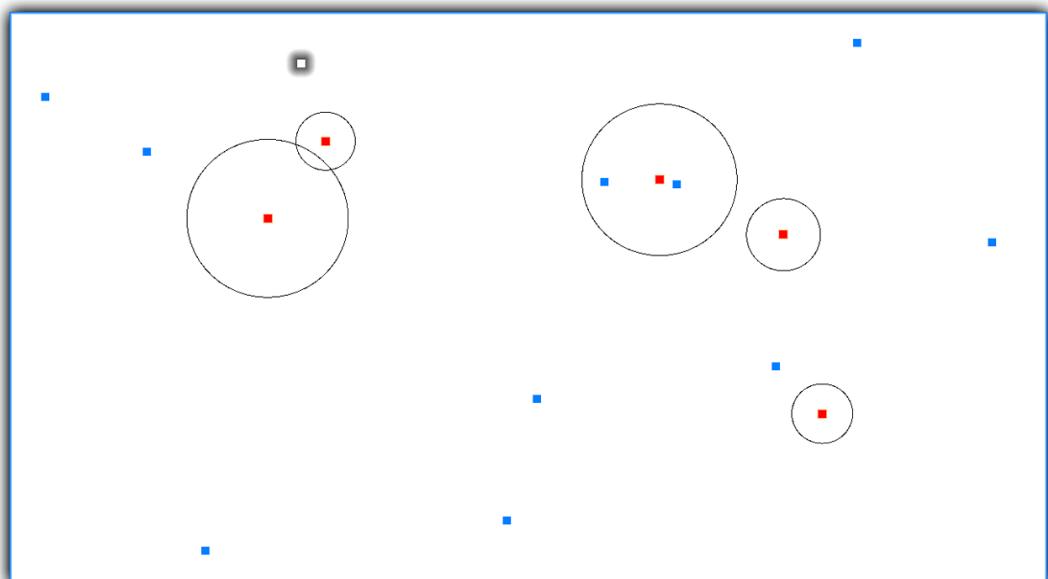
7.2. Véletlenszerű tényezők

A játékban el vannak helyezve különböző használható elemek, amelyek kihatással vannak a játékmenetre. Viszont az, hogy annak milyen mértékben van hatása pozitív irányban, csak akkor derül ki, ha már felvettük azt. Különböző típusú elemek léteznek, amiket a mesterséges intelligencia által vezérelt ellenfelek is tudnak hasznosítani. A játéktéren találhatunk életerőre, lőszerre és csapda megjelenési idejére hatással levő elemeket.

7.2.1. Konkrét megvalósítás

Az életerő és a lőszer megvalósítás szempontjából egy egész (integer) típusú, a csapda megjelenési ideje pedig egy tört (float) szám. A felvehető elemek ezeket az értékeket módosítják különböző mértékben. Ezek egyikének megvalósítása a 7.6 ábrán látható.

- Életerőt véletlenszerűen 10, 25 vagy 50%-kal növelheti.
- Lőszerhez véletlenszerűen adhat 15, 30, vagy 50 darabot.
- Csapda megjelenési idejét véletlenszerűen növelheti 25, 50 vagy 100%-kal.



7.5. ábra. Az ellenfelek félelemfaktora



7.6. ábra. Egy felvehető elem a játékban

8. fejezet

Tesztek

Ez a fejezet az általam írt játékmotor, illetve az azzal megvalósított funkciók különböző megvalósításainak tesztjeiről szól. Különböző elemeket, különféle módszerrel lehet implementálni, többféleképp lehet optimalizálni, amelyek kirajzolási idejében különbségek vannak. Léteznek kisebb és nagyobb hatékonyúságú algoritmusok a különböző problémák megoldására.

8.1. Kirajzolási módszerek

Alapvetően két fő kirajzolási módszer létezik az OpenGL-ben. Az egyik a régebbi glBegin-glEnd blokkos, a másik a modernebb Vertex Buffer Object-es (VBO). A kettő között a legfőbb különbség a gyorsaság. VBO-s megjelenítési mód esetén, mint ahogy a nevében is benne van, egy buffer-be (tárolóba) betöltjük az összes kirajzolni kívánt elemet, és azt egyben adjuk át a videokártyának, a legmegfelelőbb formátumban. Ez sokkal gyorsabb, mivel a régebbi egyesével adja át a pontokat kirajzolásra. A 8.1-es ábrán a régebbi, a 8.2-es ábrán pedig a VBO-s megjelenítési mód látható. Ugyanazoknak az objektumoknak a kirajzolása sokkal gyorsabb VBO-val, ez látható minden képen a bal felső sarokban. Sokkal jobb a hardver kihasználtsága is, és az FPS (képkocka/másodperc) is kicsivel több, mint 8x magasabb.

8.2. Profilozás

A játék egyes elemeinek betöltéséről, kirajzolásáról diagramok készültek. minden ábrán, a függőleges tengelyen az adott objektum, vagy elem neve látható, a vízszintes tengelyen pedig az idő van feltüntetve milliszekundumban.

A 8.3-as ábra a játékindítás folyamatának időigényét részletezi, hogy az egyes elemek létrehozása, betöltése hogyan oszlik el a teljes időhöz képest. Mint látható, az időtartam nagy részét a képi elemek betöltése teszi ki, a további négy elem szinte elhanyagolható ilyen szempontból.



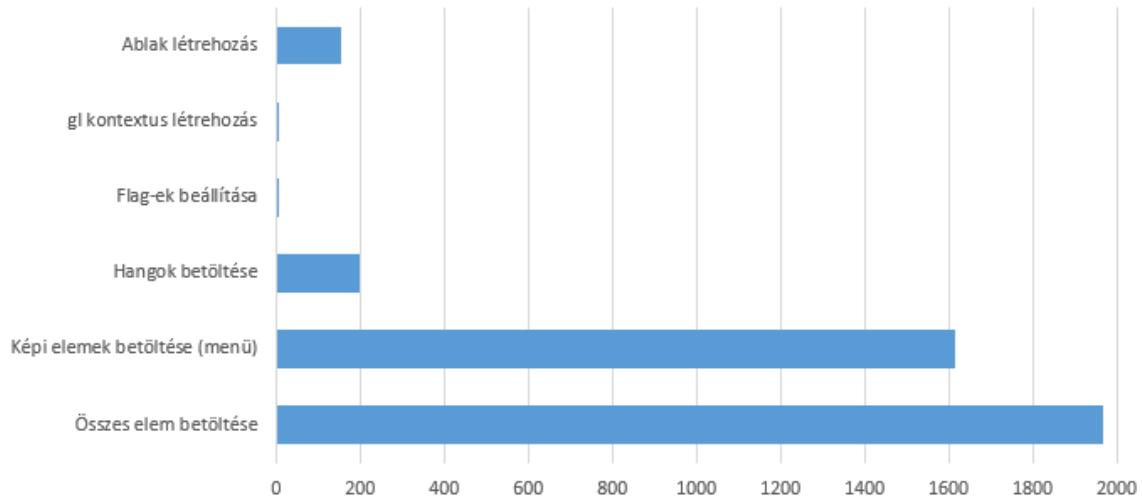
8.1. ábra. A terep kirajzoltatása a régebbi módszerrel



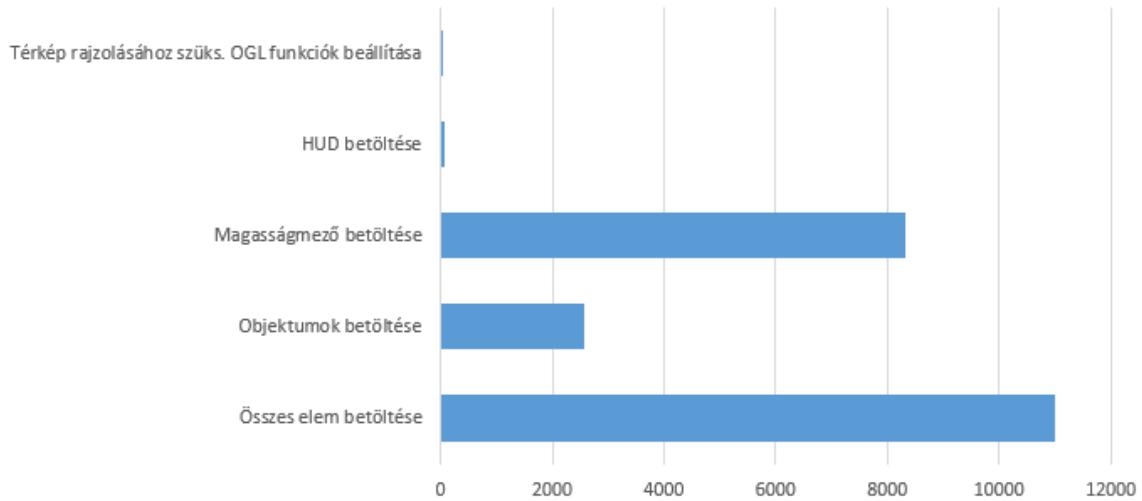
8.2. ábra. A terep kirajzoltatása az újabb, VBO-s módszerrel

A 8.4-es ábrán a teljes játéktér betöltésének, illetve az egyes részek betöltésének ideje látható.

Mivel a magasságmező betöltésének ideje nagyon kiugró, külön megvizsgáltam (8.5-ös ábra), hogy azon belül melyik rész okozza ezt. Feltűnő, hogy a magas betöltési időt a textúra betöltése okozza, mert nagy a felbontása. Ezen javítani úgy lehet, hogy kisebb felbontású textúrát választunk, és azt többször helyezzük a magasságmezőre. Ha ezt a megoldást válasszuk, akkor a jelenleg is alkalmazott módszert, miszerint a pálya elrendezésének megfelelően rajzoljuk be a magas pontok textúráit, nem lehet megvalósítani, mivel a többszörös ismétlés miatt nem fog illeszkedni.

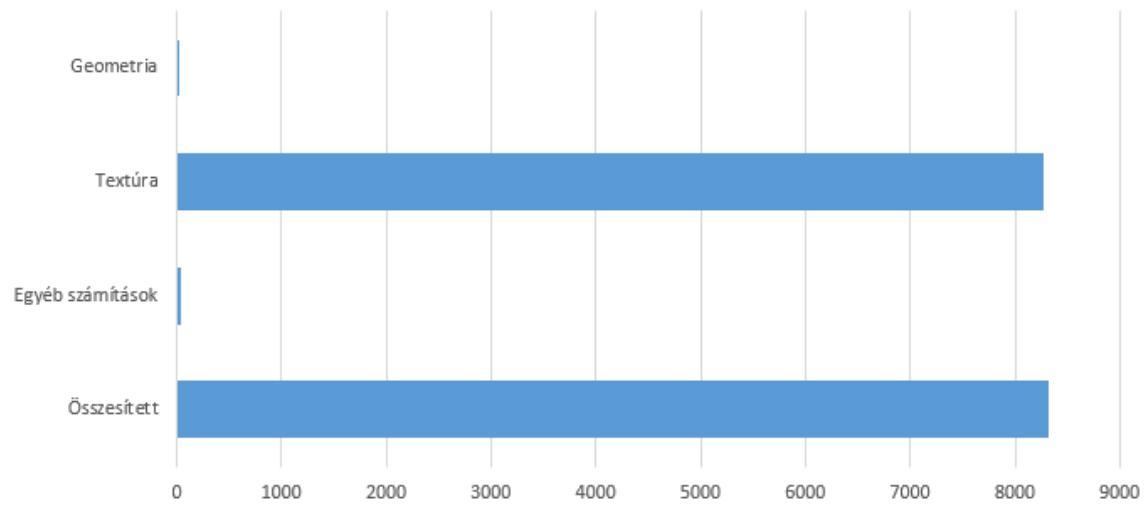


8.3. ábra. Játékindítás teljes ideje a menübe való eljutásig

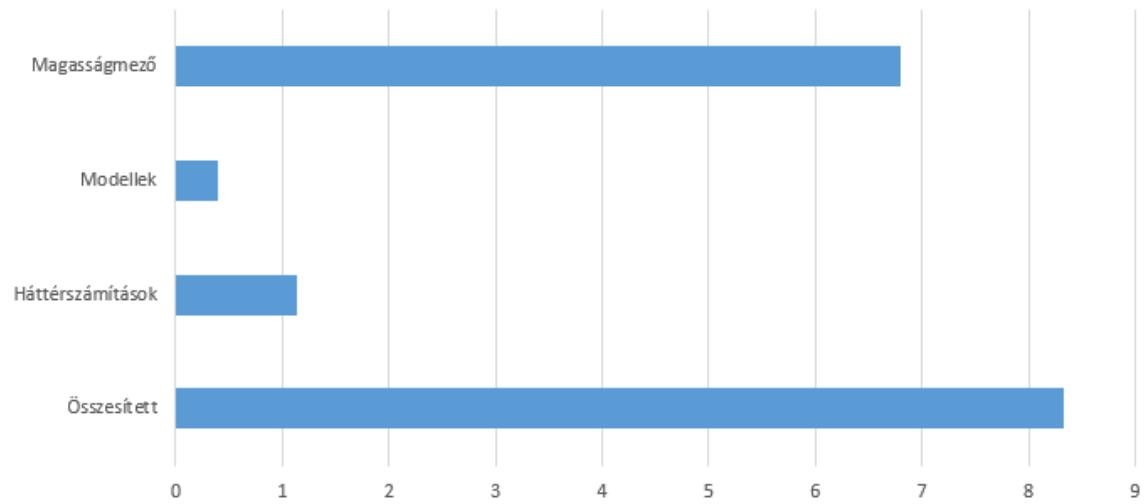


8.4. ábra. A játéktér betöltésének ideje

Végző soron megvizsgáltam, hogy egy képkocka elkészülése során, melyik játékkomponens megjelenítése vagy számítása hogyan oszlik el a teljes képkocka idejének elkészüléséhez képest. Ez fix 120 másodpercenkénti képkockaszám (FPS) mellett lett kiszámolva. Tudjuk, hogy 120FPS esetén a képkockák kirajzolása között eltelt időtartam 8,33ms, így elég volt csak a modellek, illetve a magasságmező kirajzolásának idejét mérni, a maradék értelemszerűen a további háttérszámításokat teszi ki. A másik ok, ami miatt fix FPS-t választottam a méréshez, az a túl magas másodpercenkénti képkockaszám (átlag 2000FPS) korlátozás nélkül. Ilyen esetben a képkockák kirajzolása között eltelt időtartam annyira kicsi volt, hogy még milliszekundumban is mérhetetlen volt.



8.5. ábra. A domborzat betöltésének ideje részletezve



8.6. ábra. Egy képkocka kirajzolása (120FPS esetén)

9. fejezet

Összegzés

A dolgozatban olyan optimalizálási módszerek kerületek bemutatásra, amelyek nagyban csökkentik egy adott játék, (vagy általában egy alkalmazás) erőforrásigényét. A bemutatott eszközök gyakran szükségesek ahhoz, hogy egy komplex grafikus alkalmazás optimalizációját a fejlesztők el tudják végezni.

A játékfejlesztés egy nagyon időigényes munka, ugyanis az esetek jelentős részében szükség van a kód optimalizálására. E nélkül az alkalmazások nem lennének képesek a virtuális valóság valós idejű megjelenítésére, vagy csak jóval nagyobb hardverigény mellett. Egy mai modern játékot olyan hatalmas bezárható terület, valószerű fizika, illetve szép, szinte teljesen valósághű grafika jellemzi, amely akár még fél évtizede is elköpzelhetetlen lett volna.

A megjelenítéshez az OpenGL grafikus függvénykönyvár került felhasználásra. Ahhoz, hogy a megjelenítés sebessége elfogadható legyen, vertex buffer objektumok (VBO-k) használatára volt szükség.

A dolgozat bemutatja a négyes fákat és az oktális fákat, amelyek a nagy játéktérben való keresés optimalizálásában segítenek. A karakter ütközésvizsgálatának szempontjából a probléma annyival komplexebb, hogy ott mozgó karakterről van szó, amelynél az ütközésvizsgálatot a karakter több pontjára is el kell végeznünk.

A lövések esetében a találatok regisztrálását egy egyszerűsített geometriára szokták elvégezni. Ez szintén egy olyan optimalizálási módszer, ami ahhoz szükséges, hogy a játék futási ideje elfogadható legyen, még nagyobb méretű térképek esetében is.

Mivel egy modern játék esetében az egyik fő cél az, hogy minél valóságszerűbb legyen a játékmenet, ezért a karakterek mozgatására is kiemelt figyelmet kell fordítani. A dolgozat bemutatja az ezek mozgatásához gyakran használt csontváz alapú karakteranimálási módot és ehhez kapcsolódóan az elterjedt interpolációs görbéket.

10. fejezet

Summary

The thesis presents some of the well-known optimization methods, which are able to decrease the resource requirements of a computer game or an application. The mentioned methods are essential for developers for creating complex graphical applications.

The game development is a time consuming task, because we have to optimize the resulted source code for achieving acceptable performance. Without these steps, the applications were not able to render the frames of the virtual reality in real-time. The modern computer games have large, explorable areas, realistic physics and photorealistic graphics, which was unimaginable a half decade ago.

The application uses OpenGL for rendering. For the acceptable performance we have to use vertex buffer objects (VBO).

This work explains the concept and implementation of quadtrees and octrees, which makes the searching in large game space efficient. The collision detection of characters is much more complex, because the characters can move and they have more points.

In the case of bullets, the registration of the hits is usually applied to a simplified geometry. It is also an optimization method which necessary for real-time games in the case of large maps.

The realistic game mechanics is also one of the main goal of modern games. Therefore, we have to consider the details of character animations. This work presents the frequently used skeleton-based animation methods and the corresponding interpolation curves.

Irodalomjegyzék

- [1] Epic Games, *Unreal Engine*, hivatalos dokumentáció,
<https://docs.unrealengine.com>, 2017.11.20
- [2] CRYTEK, *CryENGINE*, hivatalos weboldal,
<http://www.crytek.com/cryengine>, 2017.11.20
- [3] id Software, *idTech Engine*,
https://en.wikipedia.org/wiki/Id_Tech, 2017.11.20
- [4] *SDL library*, hivatalos weboldal:
<https://www.libsdl.org/>, 2017.11.20
- [5] Samet, H.: *The quadtree and related hierarchical data structures*. ACM Computing Surveys (CSUR), 16(2), 187-260., 1984.
- [6] *Informált (heurisztikus) keresési stratégiák*,
Mesterséges Intelligencia Elektronikus Almanach,
http://project.mit.bme.hu/mi_almanach/books/aima/ch04s01, 2017.11.20
- [7] Lluís Alseda, *A* Algorithm pseudocode*,
<http://mat.uab.cat/~alseda/MasterOpt/AStar-Algorithm.pdf>, 2017.11.20
- [8] Dudás László: *Mesterséges Intelligencia alapjai*, előadás diasor 31-34. oldal,
<http://ait.iit.uni-miskolc.hu/~dudas/MIEAok/MIea7.PDF>
- [9] Szirmay-Kalos László, Antal György, Csonka Ferenc: *Háromdimenziós Grafika, animáció és játékfejlesztés*, ComputerBooks, 2003.
- [10] ActiveState Code Recipes: *A-Star Shortest Path Algorithm*,
<http://code.activestate.com/recipes/577457-a-star-shortest-path-algorithm/>, 2017.11.20

-
- [11] GameTutorials: *OpenGL tutorial*, GitHub repository,
<https://github.com/gametutorials/tutorials/tree/master/OpenGL>,
2017.11.20
 - [12] Juhász Imre: *Görbék és felületek modellezése*, Digitális Tankönyvtár,
http://www.tankonyvtar.hu/hu/tartalom/tamop425/0046_gorbek_es_feluletek_modellezese, 2017.11.20
 - [13] J. M. P. van Wareren: *The Quake III Bot*, University of Technology, Delft,
<http://fd.fabiensanglard.net/quake3/The-Quake-III-Arena-Bot.pdf>,
2001.
 - [14] Zhang, Z., & Zhao, Z.: *A multiple mobile robots path planning algorithm based on A-star and Dijkstra algorithm*. International Journal of Smart Home, 8(3), 75-86.
2014.

Adathordozó használati útmutató

A szakdolgozathoz elkészült 8 példaprogram, forráskódjával együtt a "Demók" mapában található, amelyben almappák vannak minden demónak. minden program mapájában található egy "src" mappa, amelyben a forráskód fájljai vannak, illetve egy futtatható állomány. A "Játék demó" mappában található egy nagyobb, komplexebb program, amelyhez szintén tartozik forráskód és futtatható állomány is. A "Dolgozat forráskód" mappában kapott helyet a dolgozat forráskódja, illetve a "Dolgozat" mapában a magyar és angol nyelvű összefoglaló.

Egyes programokhoz szükség van a Microsoft Runtime Library 2015 x86-ra, amely a következő weboldalról tölthető le:

<https://www.microsoft.com/en-us/download/details.aspx?id=48145>

Animációs demó

Indítás után lehetőség van a W-vel előrefelé, S-el hátrafelé haladni, az A-val és D-vel jobbra és balra forogni. A kamerát a C gombbal tudjuk függetleníteni a robottól, így nem fogja követni azt. Ez esetben is a mozgás a WSAD gombokkal lehetséges, viszont a SPACE-el, illetve az X-el tudjuk emelni, és leengedni a kamerát. A C gomb újból lenyomásával vissza tudjuk helyezni a kamerát a robot mögé.

Viselkedés demó

A képernyőn látható pontok közül a pirosak az ellenfelek, a kékek a biztonsági pontok, ahova az ellenfelek visszavonulnak. Fehér ponttal van ábrázolva a játékos, amely a WSAD gombokkal irányítható. Az R gombbal lehetőség van újragenerálni az ellenfeleket új tulajdonságokkal, az E gombbal pedig megjeleníthető az a kör, amely a félelmüket jelöli. Ha megmozdul a játékos, a körön belül lévő pontok közül a legközelebbihez vonul vissza.

Ütközésvizsgálat demó

Lehetőség van a kamera pozíciójának megváltoztatására a WSAD gombokkal, illetve a kamera forgatására a (NUM)8546 gombokkal. Alapértelmezetten két darab piros háromszög jelenik meg, amelyek zöldre váltanak, ha a kamera irányvektora metszi azokat. Konzolban közben látható a pontos metszéspont.

Magasságmező demó

Ez a program csak a magasságmező betöltését, és VBO-s kirajzolását szemlélteti, indítása után felhasználói beavatkozásra nincs lehetőség.

Modellbetöltés demó

Ez a program csak a .obj kiterjesztésű modell betöltését, és VBO-s kirajzolását szemlélteti, indítása után felhasználói beavatkozásra nincs lehetőség.

Lagrange-interpolációs demó

A konzolos felületen meg kell adni, hogy hány pontot szeretnénk definiálni. Ennek beírása után a pontok koordinátáját kell megadnunk sorban, x és y értékeket felváltva, az x értékek csak pozitívak lehetnek. Ezután konzolban az egyes x pontokhoz tartozó y értékek és a skálázás mértéke, a grafikus ablakban pedig a függvény képe látható. További beavatkozásra nincs lehetőség.

Útvonalkeresés demó

A képen látható pontok közül feketék a falak, zöld a kezdőpont, piros a célpont, illetve fehérek a legrövidebb útvonal egyes részei. A WSAD gombokkal lehetőség van a piros, azaz a célpont mozgatására, ez esetben a legrövidebb útvonalat minden újra-számolja.

Négyes fa demó

Indítás után meg kell adni hogy mennyi véletlenszerű pontot szeretnénk definiálni, egy terület méretét, illetve hogy mekkora legyen az a küszöbérték, amelyet meghaladó objektumszám felett már felossza a területet a program. Ezen értékek megadása után a konzolos ablakban megjelenik a felosztás fája, megjelenítve a területeken lévő pontok számát.

A játék demó

A játék indítása után a főmenü jelenik meg, ahol a játékosnak lehetősége van a "Start" gombra kattintva betölteni a játékteret, illetve a "Quit" gombra kattintva kilépni. A játéktér betöltése után a WSAD gombokkal lehet mozogni, az egérrel körülnézni, a SPACE-el ugrani, a CTRL-al guggolni, a görgővel állítani az egér érzékenységét, az ESC-el pedig kilépni a főmenübe. Ha túl sokáig nem mozdul meg a játékos karaktere, megjelenik a csapda, majd leesik a mélybe. Mind a menüben, mind játék közben lehetőség van a + és - gombokkal a hangerőt állítani.