

SZAKDOLGOZAT



MISKOLCI EGYETEM

Itt jelenik meg a szakdolgozat címe, akár
több sorban is

Készítette:

Ide kerül a hallgató neve

Évfolyam. szak-szak

Témavezető:

Egyik konzulens neve

Másik konzulens neve...

MISKOLC, 2016

MISKOLCI EGYETEM

Gépészszmérnöki és Informatikai Kar

Alkalmasztott 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:

Témavezető: Piller Imre (egyetemi tanársegéd)

A feladat kiadásának ideje:

.....
szakfelelős

EREDETISÉGI NYILATKOZAT

Alulírott; Neptun-kód:
a Miskolci Egyetem Gépészmeérnöki és Informatikai Karának végzős
szakos hallgatója ezennel büntetőjogi és fegyelmi felelősségem tudatában nyilatkozom
és aláírásommal igazolom, hogy
című szakdolgozatom/diplomatervem saját, önálló munkám; az abban hivatkozott szak-
irodalom felhasználása a forráskezelés szabályai szerint történt.

Tudomásul veszem, hogy szakdolgozat esetén plágiumnak számít:

- szószerinti idézet közlése idézőjel 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, év hó nap

.....

Hallgató

1.

szükséges (módosítás külön lapon)

A szakdolgozat feladat módosítása

nem szükséges

.....

dátum

témavezető(k)

2. A feladat kidolgozását ellenőriztem:

témavezető (dátum, aláírás):

.....
.....
.....

konzulens (dátum, aláírás):

.....
.....
.....

3. A szakdolgozat beadható:

.....

dátum

témavezető(k)

4. A szakdolgozat szövegoldalt

..... program protokollt (listát, felhasználói leírást)

..... elektronikus adathordozót (részletezve)

.....
.....

egyéb mellékletet (részletezve)

tartalmaz.

.....

dátum

témavezető(k)

5.

bocsátható

A szakdolgozat bírálatra

nem bocsátható

A bíráló neve:

.....

dátum

szakfelelős

6. A szakdolgozat osztályzata

a témavezető javaslata:

a bíráló javaslata:

a szakdolgozat véleges eredménye:

Miskolc,

a Záróvizsga Bizottság Elnöke

Tartalomjegyzék

1. Bevezetés	7
2. Problémakör	8
2.1. Játékmotorok	8
2.2. Miért volt szükség új játékmotor készítésére?	8
2.3. Lehetséges optimalizálási problémák	8
2.4. Az egyedi motorral megvalósított funkciók	9
2.4.1. Mesterséges intelligencia	9
2.4.2. Hangok	9
2.4.3. Magasságmező	9
2.4.4. Konkrét megjelenítés módja	10
2.4.5. A csapda mint játékelem	10
2.4.6. A lövés	10
3. Komponensek	11
3.1. Felhasznált technológiák (C++, SDL)	11
3.2. Az elképzelés, nagyobb komponensek	11
3.2.1. Modellbetöltés textúrázással	12
3.2.2. Irányítás, fizika	12
3.2.3. Hangok	13
3.2.4. Állapotok	14
3.2.5. Lövés	14
3.2.6. Mesterséges intelligencia	15
3.3. A játékindítás folyamata	15
4. Ütközésvizsgálat	16
4.1. A karakterek ütközésvizsgálata mozgás szempontjából	16
4.2. A domborzat és tereptárgyak ütközésvizsgálata a lövés szempontjából	17
4.3. A játékos, és az ellenfelek hitboxa	18
4.4. Optimalizálási módszerek	19
5. Útvonalkeresés	20
5.1. A waypoint-ok fogalma és jellemzői	20
5.2. A waypoint-hoz tartozó adatok	20
5.3. Útvonalkeresés waypoint-ok alapján	20
5.4. Az A*-algoritmus	21
5.5. A pontos megvalósítás	22
6. Animáció	25

7. Ellenfelek viselkedésének modellezése	26
8. Tesztek	27
9. Összegzés	28
Irodalomjegyzék	29

1. fejezet

Bevezetés

Egy FPS játék elkészítése különféle számítási- és optimalizálási problémát felvet.

Sorrvanni, hogy milyen jellegűek ezek, hogyan kapcsolódnak egyáltalán egy témához.

Ez a rész elég ha 1-2 oldal. Bőven elég lesz majd csak a végén összerakni.

2. fejezet

Problémakör

2.1. Játékmotorok

Új játék fejlesztésénél két fő lehetőség adott, az első, hogy választunk egy előre megírt motort, ami a játék alapját adja, illetve ha egyiket sem tartjuk megfelelőnek, akkor írunk egyet saját magunknak. Meglévő motorok például az Unreal engine, Cryengine, Quake engine (ID Tech). Az Unreal engine első változata 1998-ban jelent meg, az első játék, ami ezt használta az Unreal című játék volt. A motor jelenlegi, legújabb verziója a 4.17-es, nagyon sokat fejlődött az évek során. A Cryengine-t először a Far Cry 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 általam írt motorhoz a legközelebb a Quake engine (ID tech) 2 és 3 áll. Ezen motorok, összességében bárminelyik megoldást is választjuk, a játékunk fő komponenseinek a háttérszámításait végzik, pl.: lövés pályájának számítása, gravitáció, ütközésdetektálás, billentyűzet és/vagy egérkezelés, hangok, hálózati kommunikáció, animációk, mesterséges intelligencia. A játékmotor kiválasztása vagy megírása után, a következő lépés a játék felépítésének ki-alakítása a motor adta lehetőségekkel, eszközökkel. Úgy kell választanunk játékmotort, hogy annak funkcióival megvalósítható legyen a játék.

2.2. Miért volt szükség új játékmotor készítésére?

A saját játékom tervezésekor, át kellett gondolnom, hogy milyen jellegű programot szeretnék írni, annak milyen funkciói lesznek, hogyan tervezem azt megvalósítani. Mint ahogy korábban írtam, úgy kell megválasztani a motort, hogy azzal megvalósíthatók legyenek az általunk elképzelt funkciók. Ebben az esetben egyszerűt azért volt szükség új játékmotorra, mert a tervezés során előjöttek olyan ötletek, játékelemek, amik teljesen egyediek ebben a formában. Ilyen például a csapda, ami egy bizonyos idő után jön elő, ha a játékos nem mozdul meg. Másrészt pedig ez lehetőséget adott az optimalizálási problémák vizsgálatára.

2.3. Lehetséges optimalizálási problémák

A játékban megvalósítandó elemek között szerepel a lövés, a mesterséges intelligencia, az ütközésvizsgálat, és mindenek megvannak a maga optimalizálási problémái. Ütkö-

zésvizsgálatra 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 talajon, 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, amit a későbbiekben részletezünk.

2.4. Az egyedi motorral megvalósított funkciók

2.4.1. Mesterséges intelligencia

Egyjátékos FPS játékról van szó, így mindenféleképp kellett egy, az ellenfelek mozgását irányító mesterséges intelligencia. Ez az egész az A* algoritmusra alapszik. A játék véletlenszerűen, különböző helyekre kirak x 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, ún. waypointok, amik 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 célpontja. Mivel a játékos folyamatosan mozog, mindezt másodpercenként legalább 15-30x kell kiszámolni.

2.4.2. Hangok

Egy játék alapelemeihez hozzáartoznak a hangeffektek, zenék is, amik élvezetessé teszik azt. Egy jól megválasztott zene például nagyon sokat tud javítani a játékélményen. Ezek megszólalásáért az SDL_mixer felelős, ami lehetővé teszi azt is, hogy több hang egyidőben, átfedésben megszólaljon, és ne várják meg egymást. Ezt a Sound osztályban valósítottam meg. Lehetőség van hangcsatornákat megadni, folyamatos újrajátszást, illetve a hangok egymáshoz viszonyított hangerejét beállítani.

2.4.3. Magasságmező

A játékteret adó domborzatot, a játék egy fekete-fehér, 384x384 pixeles képből számolja ki. Az adott képen minél magasabb egy pont, annál fehérebb, így a fehér adja a legmagasabb, a fekete szín a legalacsonyabb pontot. Ezekből az adatokból vissza lehet számolni a játékos függőleges pozíóját is, illetve az átmeneteket nézve, meredekségeknek megfelelően a visszacsúszást. Mindez tehát megadja a játékteret minden szempontból (vizualitás, játékos mozgástere).

2.4.4. Konkrét megjelenítés módja

A játéktér adatait tehát képből kinyertük, de ez még nem jelenti azt, hogy látjuk is a monitoron. 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 hasznló, aréna jellegű, túlélős játéktól. Az egyik ilyen elem, hogyha a játékos megáll x ideig (az x majd tesztelés során derül ki mi a legoptimálisabb), akkor megjelenik egy csapda, majd leesik, és meghal. Vannak olyan játékok, ahol szimplán az ellenfelek összetönik a játékost a mozgásra, de itt konkrétan ki is van kényszerítve. Jelenlegi beállítás: Ha megáll a játékos 5mp-ig, akkor megjelenik a csapda modell egy hang kíséretében, illetve onnantól nem lehet már pozíciót változtatni, csak az egérrel körbe nézni. Majd még 2mp leteltével már az egeret sem érzékeli, a föld felé fordul a kamera, majd leesik, ekkor ismét egy hangot lehet hallani, és értesül a játékos arról, hogy miért halt meg.

2.4.6. A lövés

Egy ilyen játékban alapelem az is, hogy tudunk lőni, mivel ez nélkül az egész értelmét veszti. A textúrázott játéktéren ugyan nem látni, de az egész több százezer háromszögből áll. Adott egy irányvektor, ami azt határozza meg, merre fordul épp a játékos, azaz mire néz. Ennek a vektornak a metszéspontját kell vizsgálni a háromszögekkel, hogy találatot meg tudjuk jeleníteni, ami jelen esetben a falon egy lövésnyom. Szóval ki számolja a program a metszéspont x, y, z koordinátáját, majd odailleszt egy lövésnyom textúrát. Az ellenfeleknél szintén metszéspontot kell számolni, csak ott hengerre. Ki lehetne számolni a modellre is közvetlen, de az nagyon lassú lenne, így egy leegyszerűsített alakzatot választottam. Ez az ellenfél ún. hitboxa.

3. fejezet

Komponensek

3.1. Felhasznált technológiák (C++, SDL)

Windows operációs rendszeren az általam legjobban megkedvelt program a Microsoft Visual Studio, aminek a 2015-ös verzióját használom. Az évek során nagyon megszoktam a kezelőfelületét, illetve az azon elhelyezkedő gombok, funkciók elérhetősége ebben a leglogikusabbak számomra. A játék megírásához számomra a C++ programozási nyelv túnt a legmegfelelőbbnek. A mai játékok jó része is ezzel a nyelvvel íródik, mivel „natív” fut a kód a gépen, sokkal jobb választás mint a C# vagy a Java. Ezalatt azt értem, hogy pl. egy Java nyelven megírt program futtatásához szükség van egy virtuális gépre, ami tudja értelmezni a fordító által lefordított, még nem gépi kódot. És ez a virtuális gép felelős azért, hogy az adott hardver gépi kódjára forduljon a program. Vannak olyan alkalmazási területek, ahol az utóbbi kettő a jobb, de egy játék esetében az a lényeg minél gyorsabban fusson egy adott gépen, minél több FPS-t (képkocka/másodperc) produkálva.

A kép világ kirajzolásához, illetve a hangok megszólaltatásához az SDL 2.0-t választottam. Az alábbi linken található bővebb leírás, hogy mit is takar pontosan az SDL:

3.2. Az elképzélés, nagyobb komponensek

Mint ahogy a bevezetésben is írtam, egy FPS (First Person Shooter = belső nézetes lövöldözős játék, az adott személy szemszögéből látjuk az eseményeket) játék elkészítését tűztem ki célul. A konkrét elképzélés az, hogy egy aréna jellegű, túlélő játék készüljön el. Nem terveztem online módot, a játékosnak egy mesterséges intelligenciával kell harcolnia. A játékmenet hullámokra lesz osztva, és ahogy halad előre a játékos, annál nehezebbé válik (több és/vagy erősebb ellenfelekkel). Egy hullám akkor ér véget, ha egy előre megadott számú ellenfelet sikerül megölni. A körök között a játékosnak lehetősége nyílik az életerőt és lőszert feltölteni, fejleszteni a fegyverét a teljesítményének megfelelően. A játékmenet nehezítve lesz azzal (azon felül hogy lőnek rá), hogy ha x másodpercig megáll a játékos, akkor azonnal meghal, így minden mozgásban kell lenni. Ennek az időtartamnak a pontos meghatározása majd a tesztelés során alakul ki. Lehetőség lesz különböző elemeket (ún. power up-okat) felszedni a körök alatt, ami lehet pozitív vagy negatív hatású is. Pl. adhat/vehet el életerőt, adhat/vehet el lőszert, adhat/vehet el fegyverfejlesztést.

A fentebb leírtak megvalósítása nagyon sok munka és idő. Meg kell valósítani a modellbetöltést textúrázással, az irányítást, fizikát, ütközéskezelést, hitboxot, mesterséges intelligenciát, fényeket, hangokat, árnyékokat.

3.2.1. Modellbetöltés textúrázással

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

Magasságmező beolvasásáért, illetve számításáért, illetve a kirajzolásért egy-egy külön osztály felelős. 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 3.1 ábrán látható képet adhatjuk meg, amiből a magasság értékek számolódnak. A magasságmező textúráját egy külön képfájlból tölthetjük be. Egy ilyen látható a 3.2 ábrán.

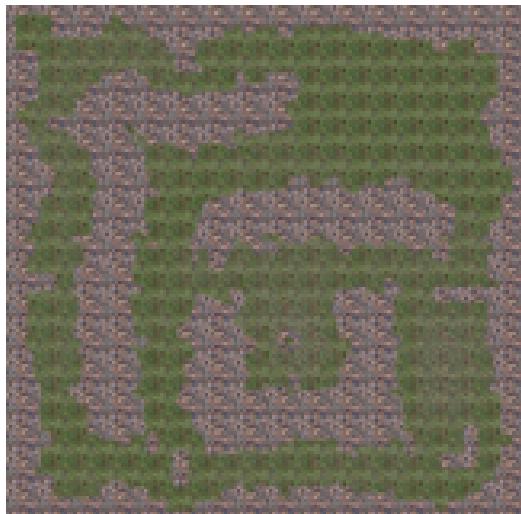


3.1. ábra. A magasságmező adatait tartalmazó bitmap

A 3.3 képen pedig a játékban való megjelenés látható. Ez egyben megoldja az ütközéskezelést is, illetve, hogy a túl meredek parton ne lehessen felmenni (lecsúszik róla). A megjelenítendő játéktér részeit egy osztály fogja össze.

3.2.2. Irányítás, fizika

A karakterrel előre, hátra, balra illetve jobbra mozogni a WSAD gombokkal, a kamera forgását az egérrel lehet vezérelni. Lehetőség van a SPACE-el ugrani, a CTRL-al guggolni, SHIFT-el gyorsabban menni, illetve a görgő fel/le mozgatásával az egér érzékenységét állítani. A kamera mozgatásához szükséges függvények is, és az állapotok



3.2. ábra. A magasságmező textúrája



3.3. ábra. A megjelenített magasságmező

kezeléséhez tartozó változók is külön osztályban vannak. minden cselekedethez tartozik egy bool (két állapotú) változó. pl. ha tegyük fel előre megyünk, akkor az ehhez tartozó változó „true (igaz)” értéket vesz fel, amihez különböző eseményeket lehet kötni, mint pl. a kamera előre mozgatását, bizonyos animációk lejátszását.

3.2.3. Hangok

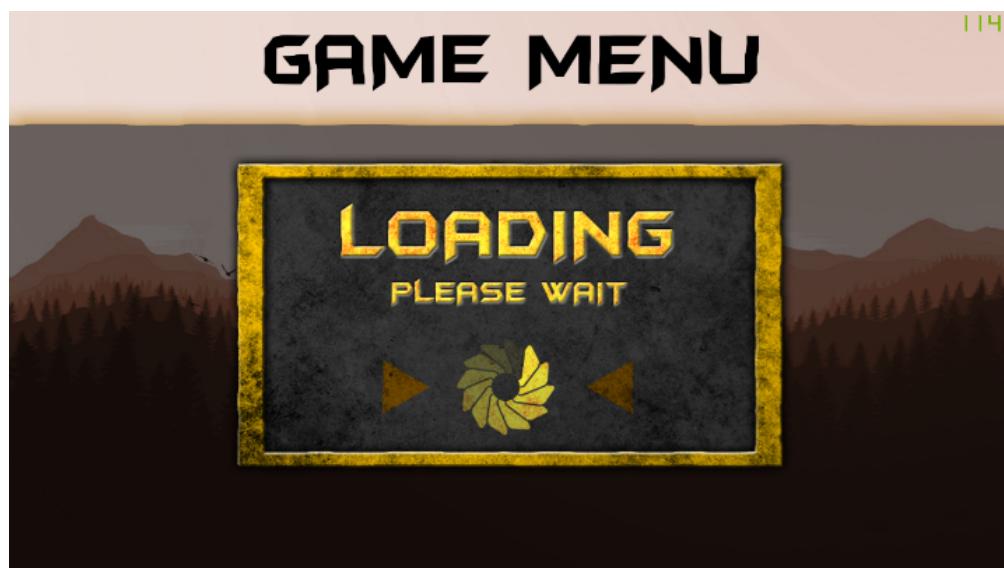
A hangok megszólaltatáshoz az `SDL_mixer`-t veszi segítségül, amiben külön vannak kezelve a háttérzenék, illetve a hangeffektek (lövés, lépés stb.). Háttérzenéhez egyszerre 1 csatorna tartozik, míg hangeffektekhez végtelen. Meg lehet adni, hogy a hang hányszor kerüljön lejátszára, illetve melyik csatornán. A `NUM +/-` gombokkal lehet a hangerőt növelni/csökkenteni, és ezek külön vannak kezelve a menüben, illetve a játékban.

3.2.4. Állapotok

A játékban jelenleg található egy kezdetleges menü, illetve az ingame rész. Ezek, mint állapotok vannak lekezelve, külön osztályokban megvalósítva. A főmenüért, az ingame objektumokért, magasságmezőért is egy-egy külön osztály felelős. A főmenü a 3.4 ábrán, a betöltőképernyő pedig a 3.5 ábrán látható.



3.4. ábra. A főmenü



3.5. ábra. Pályabetöltés közben ez látható

3.2.5. Lövés

A lövés háttérszámításai, matematikai kalkulációk is egy külön komponensnek tekinthetők. Első lépés, hogy ki kell számolni a vektort amerre a játékos néz, minden képkockára. Mivel a világ háromszögekből épül fel, ezért ki kell számolni az egyenes háromszöggel

való metszéspontját. De mivel több 100000 háromszögről beszélünk, ez további optimalizálandó problémákat vet fel, amit a későbbiekben fejtek ki.

3.2.6. Mesterséges intelligencia

Az MI osztálynak elsősorban az a feladata, hogy az ellenfelek egy meghatározott útvonalon mozognak, ne menjenek át falakon, menjenek egymásba. A véletlenszerűen lerakott ellenfelet a legközelebbi definiált waypoint-hoz kell irányítani, definiálni kell a célpontot (ami mindenkor a játékos), és ki kell számítani a pontokon keresztül vezető legrövidebb útvonalat.

3.3. A játékindítás folyamata

Első lépésben a játék létrehozza az SDL ablakot, és konkrétan beállítja annak pozícióját, függőleges és vízszintes felbontását, illetve a flageket, pl. a kurzort, teljes képernyő legyen vagy nem.

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 pedig 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, illetve átvált azok kirajzolására. Hogy a játékos informálva legyen, ez idő alatt megjelenik a betöltőképernyő.

4. fejezet

Ütközésvizsgálat

Ebben a fejezetben az ütközésvizsgálattal kapcsolatos számításokról, optimalizációkról 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, tereptárnyakon, nem lehet fel túl meredek emelkedőn. Másrészt, vizsgálni kell, hogy a játékos által leadott lövedék eltalálja-e a domborzatot, tereptárnyakat, jelezni kell a lövedék becsapódását. Harmadrészt regisztrálni kell az ellenfeleket eltaláló leadott lövéseket. A második és harmadik pont nagyon hasonló, de még is szétszedtem, mert az ellenfeleknél nem a látható geometriát találjuk el, hanem annak egy leegyszerűsített modelljét, a számítások felgyorsítása érdekében. Szükség van további optimalizációra is, mert ha ezek nem lennének, irreálisan nagy erőforrás igénye lenne a játékoknak.

4.1. A karakterek ütközésvizsgálata mozgás szempontjából

Ez a játék szempontjából az egyik legfontosabb elem. Az hogy kirajzoltatunk valamit a képernyőre, még nem jelenti azt, hogy azon nem lehet áthaladni. A kirajzolás csak a vizualitást adja, a domborzat, a tereptárnyak, a karakterek kinézetét. A játékfejlesztő feladata az, hogy megírja külön az ezekhez szükséges ütközésvizsgálatot. Mivel ez két különálló dolog, egyes helyzetekben adódhathatnak olyan problémák, hogy ezek nincsenek szinkronban, tehát látunk valamit amin át lehet menni, vagy nem látunk valamit és mégis megakadunk benne.

Ebben a játékban a karakterek ütközésvizsgálata gradiens számítással van megoldva. Ez azt jelenti, hogy ha két, egymás mellett lévő pont magasságértéke között túl nagy a különbség pozitív irányba, az falat, vagy túl meredek emelkedőt jelent. Ha mérsékelt, vagy nagyon minimális a különbség, akkor arról vagy lecsúszik, vagy csak egyszerűen át lehet ott haladni. Mindezt a beolvasott magasságmezőből számolja, ezzel dinamikussá téve az ütközésvizsgálatot. Ha ez nem lenne így, akkor elveszne a játék azon tulajdonsága, hogy dinamikusan lehet cserélni a pályákat minden további nélkül.

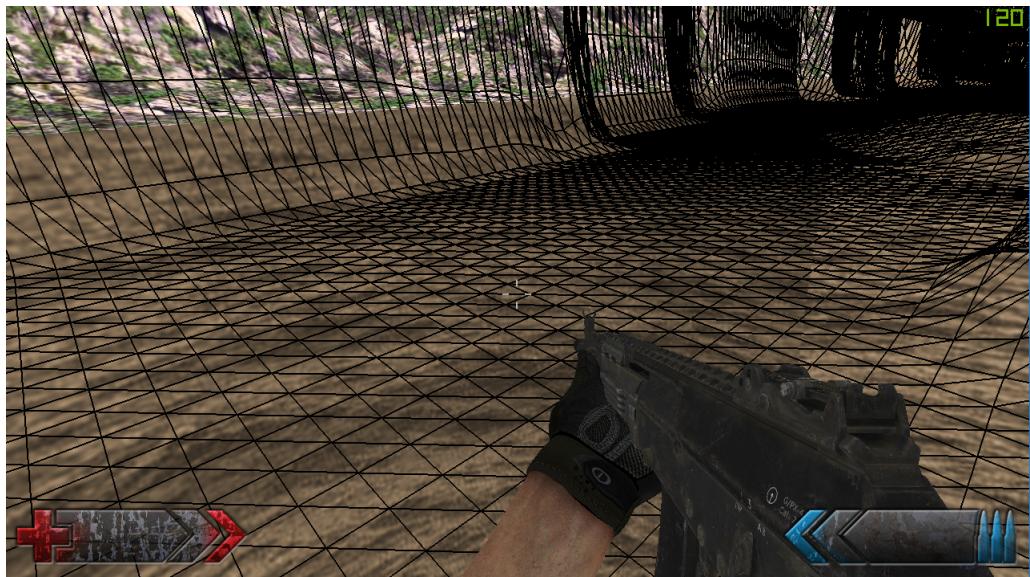
Ezen felül a gravitáció ami nagyon lényeges, mert a pályának vannak olyan magaslati pontjai, amelyekre fel lehet jutni. Ha egy ilyenre felmegyünk, és nincs semmilyen erő, ami a karaktert a föld felé húzná, akkor felfelé ugyan lekövetné a talajmagasságot, de le már nem esne alacsonyabb pontra onnan.

Ez esetben egy lefelé ható erő folyamatosan hat a játékosra és ellenfelekre, aminek a küszöbértéke mindenkor az aktuális talajmagasság. Ennek pszeudokódja:

-
1. gravitációs erő := -9,8;
 2. küszöbérték := aktuális talajmagasság;
 3. ha karakter függőleges pozíciója > küszöbérték
 4. akkor
 5. függőleges gyorsaság = függőleges gyorsaság + gravitációs erő;
 6. karakter függőleges pozíciója = karakter függőleges pozíciója
+ függőleges gyorsaság;
 6. különben
 7. karakter függőleges pozíciója = küszöbérték;

4.2. A domborzat és tereptárgyak üközésvizsgálata a lövés szempontjából

Az ilyen fajta ütközésvizsgálat számítás nem létszükséglet a játszhatóság szempontjából, de tény, hogy jóval realisztikusabbá teszi a játékot, és sok optimalizációt igényel. Ez teszi lehetővé, hogy ha lövünk egyet, és beletalál bármilyen elembe, ami a pálya része, megjeleníthessük rajta a találatot. Ennek minősége lehet többféle, csak egy textúra, vagy egyéb még realisztikusabb megoldások. Mivel a pálya háromszögekből áll, és az irány amerre nézünk egy egyenes, így itt egyenes és háromszög metszéspontját kell számolnunk térben. A pálya háromszöghálóját, geometriáját a 4.1 ábrán mutatom be.



4.1. ábra. A térkép geometriájának határ vonalai

Első lépésként, meg kell határozni azt az egyenest, amely azt írja le, amerre néz az adott karakter. Ez az egyenes két ponttal van leírva. Az egyik az adott karakter pozíójával egyezik meg. A másik, távoli pontot jelen esetben egy, az OpenGL-ben definiált függvényből, a GluLookAt-ból határozzuk meg. A GluLookAt-nek 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 túlsó pontot meghatározni. Mindegyikből ki kell vonni az adott tengelynek megfelelő

jelenlegi pozíciót, majd megszorozni egy nagy értékkal, azért, mert ez nélkül túl közel pontot kapnánk, ezáltal egy rövid egyenest.

Második lépés, ennek az egyenesnek kiszámolni a háromszögekkel való metszéspontját. A háromszögek a három csúcspontjukkal vannak leírva. Kell lennie egy függvénynek, amelynek a háromszög három csúcspontját, és az egyenes két végpontját átadva, visszaadja a metszéspont x,y,z koordinátáját. Ez működés közben a 4.2 ábrán látható.



4.2. ábra. Egyenes és háromszög metszéspontja vizualizációval

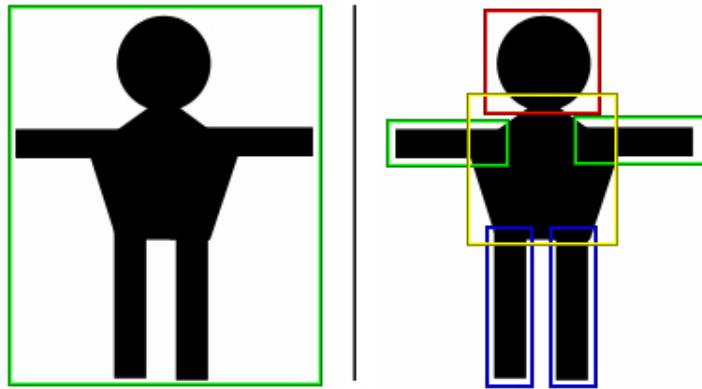
Ez a függvény először megvizsgálja, hogy azt a síkot, amelyen a háromszög van, metszi-e az egyenes, ha nem, nem is számol tovább. Ha metszi a síkot, akkor megnézi, pontosan hol metszi azt, megkapva az x,y,z koordinátákat. Ez után már csak azt kell vizsgálni, hogy a metszéspont benne van-e a három csúcspontjával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

4.3. A játékos, és az ellenfelek hitboxa

Ennek alapelve ugyanaz, mint a domborzat és tereptárgyak üközésvizsgálatánál, annyi 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, de egyéb játékokban lehet sokkal részletesebb, attól függően, mennyire szeretnénk pontosan regisztrálni az egyes találatokat.

Ebben az esetben azért nem a pontos, látható geometriára számoljuk a találatokat, mert annak nagyon nagy erőforrásigénye lenne, ellenben az lenne a leg pontosabb, tehát amit látunk, az jelenti a találatot. A találatszámításra használt leegyszerűsített geometria előnye, hogy sokkal kevesebb hátérszámításra van szükség. Viszont mivel nem azt találjuk el, amit látunk, előfordulhatnak kellemetlen, vagy érdekes szituációk. Előfordulhat, hogy úgy találjuk el az ellenfelet, hogy ott valójában nem látunk semmit, vagy ennek ellenkezője, hogy elvileg eltaláljuk, de az a rész nem tartozik a hitbox-hoz.

Online lövöldözős játéknál a külön leegyszerűsített hitbox-os megoldással lehetnek még problémák. Ilyen például az, hogy a nem megfelelő internetkapcsolat okozta nagy



4.3. ábra. Különböző részletességű hitbox-ok

válaszidő, vagy csomagvesztés miatt a hitbox elcsúszik a látható geometriától. Így akár hiába van leegyszerűsített, de még egész részletes geometria ráhúzva az adott karakterre, mégsem találjuk el az ellenfelet, mert az nem illeszkedik pontosan.

4.4. Optimalizálási módszerek

5. fejezet

Útvonalkeresés

A fejezet az útvonalkeresésnél alkalmazott módszereket mutatja be.

5.1. A waypoint-ok fogalma és jellemzői

Ez egy térben meghatározott egyszerű pont, amely nem látható a játékos számára. A mesterséges intelligencia ezeket használja fel az ellenfelek mozgatásához. Azért van rájuk szükség, mert a pályán vannak falak, különböző játékelemek, amiken nem lehet átmenni, és ezekkel a pontokkal egyértelműen meg lehet határozni, hogy melyek azok a helyek, amelyek bejárhatók.

A waypoint-okhoz rendelhetünk különféle többletinformációkat. Ilyenek lehet például, hogy az adott pontban a gépi játékosnak megállnia, 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, 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.

Az útvonalkeresés folyamata a következő fő lépésekkel épül fel.

- A waypoint-ok közül megkeressük az adott karakterhez legközelebbi. Először ezt a pontot kell megközelíteni.
- A pont adott, kis sugarú környezetét elérve A*-algoritmus segítségével meghatározzuk

•

A legközelebbi pont megkereséséhez, a következő algoritmust használom:

```
1. Waypoint vegso_pont;
2. Vec2 legkozelebbi_pont;
3. double pont_tav = 100000, temp, x_tav, y_tav;
4. float x, y;
5.
6. ciklus Waypoint pont : waypointokon
7. x = pont.x;
8. y = pont.y;
9. x_tav = abs(x - jelenlegi_xpoz);
10. y_tav = abs(y - jelenlegi_ypoz);
11. temp = sqrtf(x_tav * x_tav + y_tav * y_tav);
12. ha temp < pont_tav akkor
13.   pont_tav = temp;
14.   vegso_pont = pont;
15. legkozelebbi_pont.x = vegso_pont.x;
16. legkozelebbi_pont.y = vegso_pont.y;
```

5.4. Az A*-algoritmus

Ezt az algoritmust a legrövidebb útvonal számításra gyakran használják, mert ez az egyik leghatékonyabb. Bemeneti paraméterei a start pont X, Y, és a célpont X, Y koordinátái.

A keresést egy gráfon tudjuk végrehajtani. A gráf egy lehetséges realizációja a rács, amelyben a gráf pontjai négyzetrácsnak megfelelően helyezkednek el.

Ez a rácsos kialakítás több szempontból is előnyös. A pontok minden esetben egy-mástól egyforma 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. Pontpárok tárolására lenne szükség, amelynek a pálya elrendezéséhez való dinamikus meghatározása plusz erőforrásigény. A rácsos megoldással a pálya elrendezésétől függetlenül rakjuk le a pontokat, csak egy értéket állítunk át a pont típusának megváltoztatásakor, amely a be nem járható területeket határozza meg.

Listát kell vezetni azon pontokról amiket még nem jártunk be, illetve azokról is, amelyeket bejártunk, ezeket hívjuk nyitott (Open) illetve zárt(Closed) node listáknak. A nyitott node lista másképpen fogalmazva a hátralévő, vizsgálatlan pontok, a zárt node lista pedig a már vizsgált pontok halmaza. Kell lennie egy, vagy több feltételnek, hogy az adott pontból merre haladhatunk tovább, ilyen például az, hogy csak vízszintesen és függőlegesen haladhatunk a pontról, vagy átlósan is. Az a pont, amin éppen tartózkodunk az a jelenlegi node (current node), amelyekre léphetünk, azok pedig a gyerek node-ok (successor node).

Egy node-hoz a következők tartoznak: X és Y pozíció, ezen felül kell lennie egy változónak, amiben az adott pontig bejárt távolság található. 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 hátralévő út, ugyanis ezekből az adatokból számolja ki egy adott pont

prioritását (kezdőponttól való táv + célponttól való táv), és így dönti el milyen pontokat részesítsen előnyben. A prioritás minél kisebb érték, annál jobb, szóval arra kell elindulni. Ha egy idő után falba ütközik, elkezdi vizsgálni minden a lehető legkisebb prioritással rendelkező, még szabad pontokat, egészen addig, míg utat nem talál valahol magának.

Az A* algoritmus leírása pszeudokóddal: A célpontot jelöljük goal_node-dal, a kezdőpontot pedig start_node-dal.

1. Helyezük a start_node-ot az Open listába. $f(start_node) = h(start_node)$ ■
2. Amíg az Open lista nem üres
3. Vegyük a az Open listából azt a node-ot, amely a legkisebb költségű
4. $f(current_node) = g(current_node) + h(current_node)$
5. ha current_node a célpont, megtaláltuk a megoldást; break
6. Generálunk minden állapothoz successor_node-ot, ami a current_node-ból származik
7. ciklus minden current_node successor_node-ján
8. Állítsuk be a successor_current_költséget, amely $g(current_node) + w(current_node)$
9. ha successor_node az Open listában van
10. ha $g(successor_node) \leq successor_current_költség$ folytatás (19. sorral) ■
11. különben ha successor_node a Closed listában van
12. ha $g(successor_node) \leq successor_current_költség$ folytatás (19. sorral) ■
13. Mozgassuk successor_node-ot a Closed listából az Open listába
14. különben
15. Adjuk hozzá successor_node-ot az Open listához
16. Állítsuk a $h(successor_node)$ -ot hogy megkapjuk a heurisztikus távolságot a célihoz
17. Legyen egyenlő $g(successor_node)$ a successor_current_költséggel
18. Állítsuk be node_successor szülőjét current_node-nak
19. Adjuk hozzá a current_node-ot a Closed listához
20. ha($current_node \neq goal_node$) lépjünk ki a következővel (az Open lista üres) ■

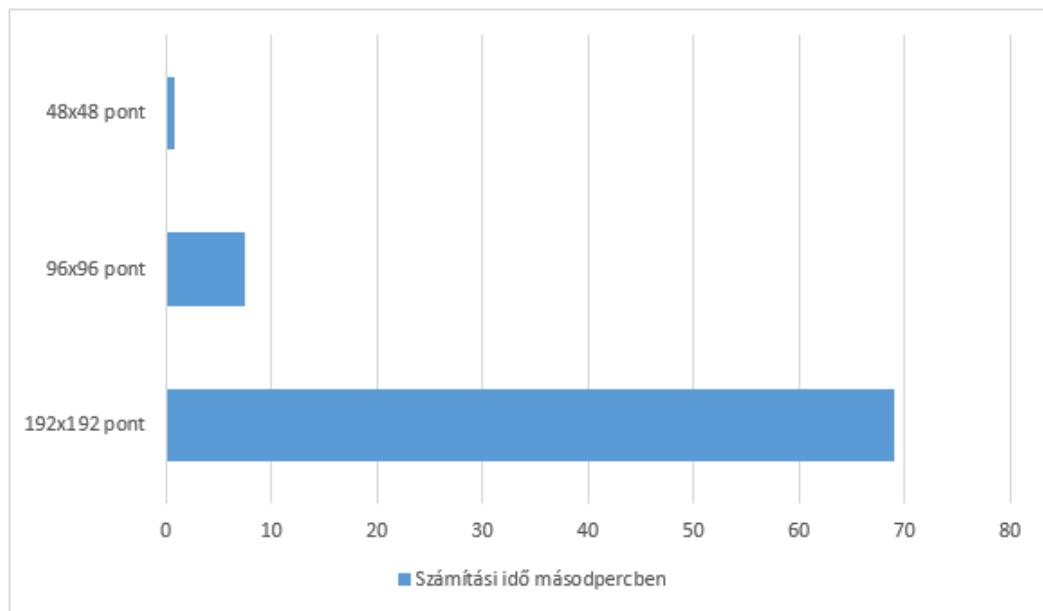
5.5. A pontos megvalósítás

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 féle magasság lehetséges, ami az aktuális változathoz megfelelő felbontásnak bizonyult.

A játéktér jelentős része (körülbelül 95%) bejárható. A magasságmezőhöz tartozó kép felbontása 384×384 pixel, amely a térkép megfelelő részletességgű megjelenítése miatt ennyi. A waypoint-ok meghatározására több lehetőség is van.

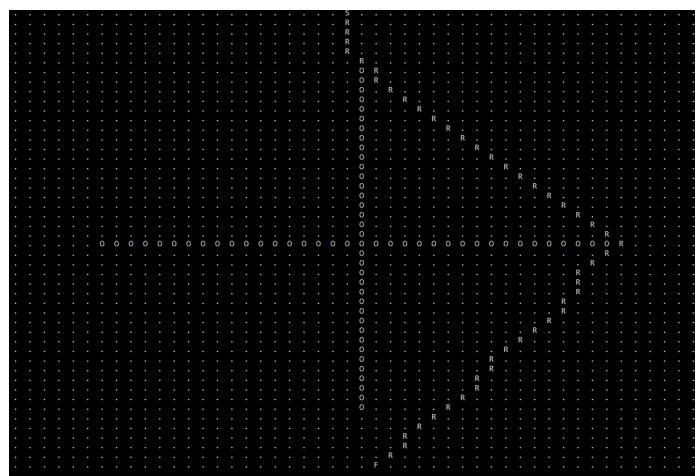
Az egyik lehetőség, ha minden pixelhez rendelünk egy pontot. Ez olyan szempontból lehet előny, hogy pontosabban meg van határozva az összes pont amelyre léphet az ellenfél. Hátránya viszont, hogy ez nagyon sok számításigény, és fölösleges is minden pixelre egy waypoint.

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.



5.1. ábra. Számításigény a felbontás és idő függvényében

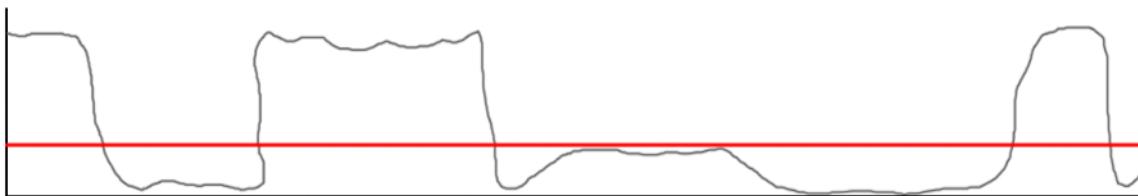
Első lépésként, létre kell hozni a waypoint hálót, amiken a mesterséges intelligencia végig fog menni, elkerülvén hogy átmenjen a tereptárgyakon, falakon. Ezek a pontok 8 pixelenként, függetlenül az adott pixel színétől meghatározásra kerülnek, viszont a típusa az adott pixel színének megfelelően lesz beállítva. Ahogy az 5.3-as ábrán is látható, aa 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 „.”-al a bejárható terület, „X”-el a falak, „S”-el az indulópont, „F”-el a célpont, „R”-el a legrövidebb útvonal van jelölve.



5.2. ábra. Útvonal meghatározó algoritmus működése

A képen még csak statikusan vannak megadva a falak, de mint ahogy fentebb olvasható, a játékon belül a beolvasott magasságmező színe határozza ezt meg. 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, stb. Ez egy fontos tulajdon-sága a játéknak, mert azt úgy terveztem, hogy ha a felhasználó akár paint-ben rajzol egy pályát, akkor a teret azonnal kialakítsa, beállítsa hogy a falakon a játékos ne tudjon átmenni, a meredek lejtőkről visszacsússzon, illetve a mesterséges intelligencia is alkalmazkodjon.



5.3. ábra. Küszöbérték a még bejárható terület meghatározásához

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.

A pontok típusának a pálya alapján legenerálása után, az A*-algoritmus segítségével meghatározza a program a legrövidebb útvonalat a cél waypointig, természetesen a pontok típusait figyelembe véve. Ezt másodpercenként legalább 15x újra kell számolni, ugyanis a játékos mozog, és a cél waypoint mindenkor legközelebb eső pont, ami ezzel együtt változik.

6. fejezet

Animáció

A csontváz kezelésének és számítógépen való ábrázolásának bemutatása

- Kulcsképkocka animáció - Csontváz alapú animáció - Blender-es animációs formátum - Egyszerűbb görbék bemutatása, amelyeket animációkhöz használnak (interpolációs módszerek)

A mesterséges intelligencia és az evolúciós algoritmusok szerepe a probléma megoldásában

- Megnézni azt a cikket, amihez az animációs videó készült. - Felvetni néhány módszert, például ANN vagy evolúciós algoritmus. (Itt még nem kell majd implementálni, egyelőre csak legyen összegyűjtve.)

7. fejezet

Ellenfelek viselkedésének modellezése

Ágens alapú viselkedés

- Leírni, hogy milyen bemenetek és kimenetek szükségesek ahhoz, hogy az ellenfelek viselkedését modellezzük.

Klasszikus, állapotgép alapú modell bemutatása Véletlenszerű tényezők a játék érdekesebbé tételehez

8. fejezet

Tesztek

A játékmotor vizsgálata egy az egyben Review jellegű észrevételek Profilozás

9. fejezet

Összegzés

A bevezetéshez hasonlóan, csak itt már múltidőben.

Irodalomjegyzék

- [1] Bujdosó Gyöngyi, Fazekas Attila: *TEX kezdől lépések*, Tertia Kiadó, Budapest, 1997.
- [2] Házy Attila: *Lineáris függvényegyenletek megoldása számítógéppel*, Doktoranduszok fóruma 2005, Miskolc, 2005. november 9., Gépészszmérnöki Kar szekciókiadványa, Miskolc, ME ITTC, 2006., 108–113.
- [3] Hettl, Mayer, Szabó: *E^ATEX kézikönyv*, Panem Könyvkiadó, Budapest, 2004.
- [4] M. E. Hohmeyer, B. A. Barsky: Rational continuity: parametric, geometric and Freudenthal frame continuity of rational curves, *ACM Transactions on Graphics*, **8** (1989), 335–359.
- [5] TEX Catalogue, www.ctan.org/tex-archive/help/Catalogue/catalogue.html