Szegedi Tudományegyetem

Informatikai Tanszékcsoport

Mesterséges Intelligencia a Játékfejlesztésben

Szakdolgozat

Készítette:
Dusnoki Attila
programtervező informatikus
szakos hallgató

Témavezető:
Dr. Csirik János
egyetemi tanár

Szeged
2013

Tartalomjegyzék

Feladatkiírás

A feladat a játékfejlesztésben használt Mesterséges Intelligencia (MI) bemutatása. Ennek része a játék MI fogalom bevezetése, történelmi áttekintése, valamint áttekintése. Ezt követően egy önállóan választott játék készítése, majd ehhez egy játék MI tervezés és megvalósítás.

Tartalmi összefoglaló

A dolgozatban a játékfejlesztésben használt Mesterséges Intelligenciát kell bemutatnom, amit két részre bontottam: elméletire és gyakorlatira. Az bevezetőben a játék MI fogalmát ismertetem, majd ennek fejlődését, és egy alap struktúrát vezetek be. Az első résznél a játékfejlesztésben használt technikákat mutatom be a teljesség igénye nélkül. A második részben az általam készített játékot, és az ehhez tervezett és megvalósított játék MI-t.

Kulcsszavak: mesterséges intelligencia, játékfejlesztés, umba

Bevezetés

A dolgozatban megismerkedünk a játék MI-vel és annak implementálási technikáival. A téma elég nagy, ez a terjedelemből is látszik, pedig csak a felszint érintjük majd. Az elkészítéshez két könyvet dolgoztam fel, az első [1] könyvből elsajátított technikákról szól a Játék MI Technikák fejezet, a második [2] könyvben bemutatott játék készítési technikákat vettem alapul a Gyakorlati Megvalósítás fejezetben.

1. fejezet

MI és Játék

1.1. Mi az MI?

A számítógép képes legyen úgy gondolkozni (döntést hozni), mint ahogy az emberek vagy állatok képesek.

Már régóta tudunk úgy programozni gépeket, hogy egy emberhez képest „természet feletti” képességeik legyenek egyes problémák megoldásában: aritmetika, rendezés, keresés és így tovább. El tudjuk érni, hogy jobban játszanak táblajátékokat mint az emberek (Sakk például). Ezek a problémák mind MI problémák voltak, de mivel ezekre széles körben született már megoldás, így már nem ezekre a problémákra koncentrálnak az MI fejlesztők.

De rengeteg probléma van amiben a gépek nem túl jók, de számunkra triviális: egy ismerős arcának felismerése, saját nyelvünkön beszélni, kreatívnak lenni. Ilyen problémák megoldása a fő iránya a mai MI-nek: megpróbálni kitalálni, illetve megvalósítani, hogy milyen algoritmusok kellenek ezen tulajdonságok megvalósításához.

Játékfejlesztői szempontból elsődlegesen csak a mérnöki oldalról érdekel: olyan algoritmusok készítése, ami a játék karaktereit ember-, illetve állatszerűvé teszi.

1.2. Játék MI fejlődése

1951-ben írták meg az első dámajátékot és sakkjátékot Mark 1-re, amiben a gép ellen játszhattunk. Ezen programok az 50-es és 60-as években addig fejlődtek, hogy egy amatőr játékossal már fel bírták venni a harcot. A sakk- és dámajátékok MI-je addig fejlődött, míg 1997-ben az IBM Deep Blue gépe megverte Kasparovot, az akkori világbajnok sakkmestert.

Az első videójátékok, amelyeket a 60-as 70-es években fejlesztettek, mint a Spacewar!, vagy a Pong, két játékos harcán alapultak MI nélkül. Az egy játékos által játszható

játékokban a 70-es években jelentek meg először számítógép által vezérelt ellenségek. Az akkori időben leghíresebb játék, a Space Invaders használt egyre nehezedő pályákat, különböző mozgásmintákat, ami kihívássá és érdekessé tette a játékot, erre alapozva a többi játék is elkezdte használni a nehézségi szinteket.

Később a 90-es években kezdtek szaporodni a stratégiai játékok, melyeknek MI-je túl sok feladatot kell, hogy ellásson, így az első játékoknak sok hibája volt: a számítógép által vezérelt játékos sokszor csalt, szinte nem működő útvonalkeresést valósított meg, valamint nagyon egyszerű három állapotú állapotgépet használt az egységek irányítására.

Azóta nagyon sokat fejlődtek a játékok és a játék MI, amely hozzásegített egyéb projekteket a megvalósuláshoz, mint pl. az IBM Watson névre keresztelt Jeopardy!-t(hasonló a magyar Mindent vagy semmit! című játékhoz) játszó gép, vagy a RoboCup, ahol robotokat tanítanak focizni.

1.3. Az MI felépítése

Rengeteg technika lesz bemutatva, így könnyű lesz elveszni benne, és ezért is fontos megérteni, hogy függnek össze.

Emiatt egy alap struktúrát definiálunk, hogy könnyebb legyen megérteni a játékban használt MI-t. Természetesen nem csak egy modell létezik, de hogy egyszerűbb legyen, minden technikát besorolunk a modell egyes részeibe.

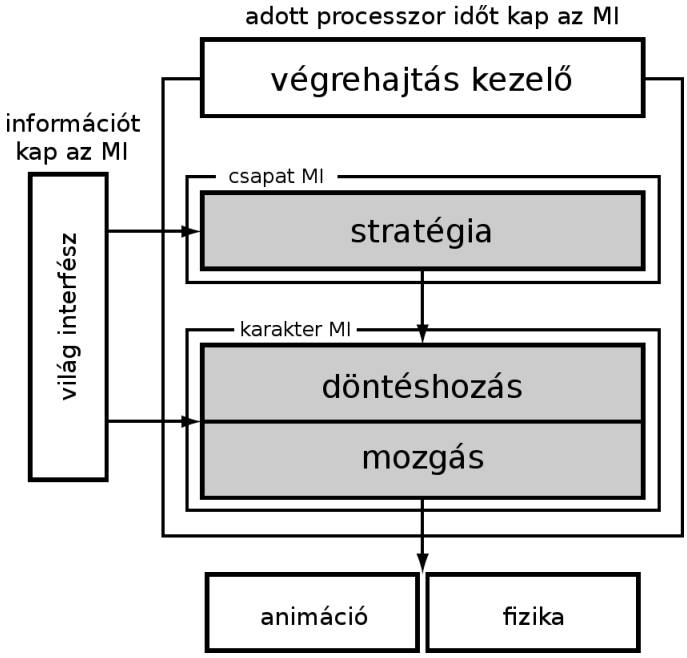
Ahogy az 1.1 ábra mutatja a három rész: mozgás, döntéshozás, stratégia.

Az első két rész olyan algoritmusokat tartalmaz, amelyek egyes karakterekre vonatkoznak, míg a harmadik egy egész csapatra vagy oldalra.

Nem minden játékban kell az összes elem. A táblajátékokban mint a sakk, csak a stratégia kell, nem hoznak saját döntéseket, és nem kell azon aggódni, hogyan mozogjanak. A másik részről viszont rengeteg játékban nincs stratégia, mint a legtöbb platform játékban.

Annak érdekében, hogy egy MI-t építsünk a játékhoz, szükségünk lesz további infrastruktúrára. A mozgás kérést egy akcióvá kell formálni animáció, fizika szimulálás, vagy egyéb formában. Valamint szüksége van információra a játékról, hogy ésszerű döntéseket hozzon. A gyakorlatban itt nem csak szimuláljuk, hogy mit láthat vagy hallhat, hanem különböző interfészeket kell létrehozni a játék világ és az MI között. Általában ez az egyik legnagyobb része az MI programozó munkájának. Végül az egész MI rendszert úgy kell működtetni, hogy megfelelő CPU időt és memóriát használjon.

Az ágens szó ritkán fog előfordulni, de a modell ami megvan adva egy ágens-alapú modell. Ebben a kontextusban az ágens-alapú MI olyan „autonóm karakter”, ami információt gyűjt a játékról és eldönti, milyen cselekvést hajtson végre az információ alapján,



1.1. ábra. MI modell struktúrája

és végül végre is hajtja azokat.

A dolgozatban az ”egység” és ”karakter” kifejezés egyaránt szerepel, ezeknek közel azonos a jelentésük. Az egységet úgy tudnám a legjobban megmagyarázni, hogy például a stratégiai játékokban mindent egységeknek (unit) neveznek: katona, tank, repülő, stb. A Warcraft 3 egy stratégiai játék és ennek egy pályáját, a Warlockot vettem alapul a saját játékom elkészítésekor, ezért számomra egyértelműnek tűnt az egység szó használata. A karakter is igazából egy egység, csak ez a játékokban egy "képzeletbeli személy", vagyis az egységet felruházzuk valamilyen tulajdonsággal, személyiséggel, kinézetettel. A könyv amiből a technikákat vettem az mindig a karakter szót használta, ezért a technikák bemutatásánál a karakter, a játék bemutatásakor az egység kifejezést használtam.

2. fejezet

Játék MI Technikák

2.1. Mozgás

Az egyik legalapvetőbb dolog amit már a legelső MI-knek is tudni kellett (pl. pong, pacman) a mozgás megvalósítása. Persze nem minden játékban kell megvalósítani (pl. sakk), mert az egységek (pl. bábúk) egyszerűen áthelyezhetők mozgatás nélkül.

Most inkább egy egyszerűbb megvalósítását nézzük a mozgásnak, mint a pálya egyik részéről a másikra eljutás, vagy egy fára felmászás.

Egy kisebb átfedés van az animáció és a mozgás megvalósításánál: az animáció is a mozgásról szól. Persze a jelenetek nem MI vezéreltek, így az ottani animációkat nem tárgyaljuk.

A következő részben megismerünk különböző MI-vezérelt mozgás algoritmusokat, az egyszerűektől az egészen bonyolultakig.

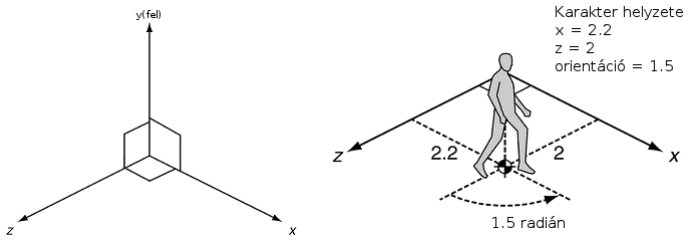
2.1.1. Alap mozgási algoritmusok

Minden mozgási algoritmusnak van egy alap formája. Kapnak egy geometrikus adatot és a saját állapotukból és a világ állapotából előállnak egy olyan geometriai reprezentációval, ahogyan szeretnének mozogni. A kimenet nagyon sok mindentől függhet.

$2D$ mozgás

Rengeteg játéknak van MI-je, ami két dimenzióban működik. Manapság nagyon ritka a két dimenzióban készített játék, a karakterek a gravitáció hatására vannak a talajon és ott mozognak két dimenzióban. Sok algoritmust meg lehet valósítani két dimenzióban, a legtöbb klasszikus algoritmus csak erre az esetre van megírva.

A karakterek két dimenzióban egy koordináta rendszerrel vannak megvalósítva. A tengelyek derékszögűek és a gravitáció merőleges az így létrejött felszínre.



2.1. ábra. Koordináta rendszer

A legtöbb játékban a geometria három dimenzióban van tárolva és megjelenítve. Ez három tengelyt jelent: x, y, z. A legtöbb esetben az y a gravitációval ellentétes irányba mutat ("fel"), és az x és z koordináták pedig a talajon fekszenek el. Ebben a fejezetben az x, z koordinátákat használjuk a két dimenziós mozgáshoz.

Egy két tengelyű koordinátánál az objektumnak van iránya (orientációja), amerre néz az objektum. Ez általában egy szögként van megvalósítva egy adott tengelyhez viszonyítva. Itt egy óramutató járásával ellentétes irányú szög radiánban, a pozitív z tengelyhez viszonyítva. Ezzel a három értékkel már egy objektumnak az állapotát meg lehet jeleníteni.

Az ezeket változtató algoritmusokat, egyenleteket statikusnak nevezzük, mert nem tartalmaznak információt a mozgásról.

Egy megvalósítása: a pozíció egy 2 dimenziós vektor, az orientáció egy lebegőpontos szám.

$2\frac{1}{2}D$ mozgás

A $3D$-s geometria matematikája helyenként elég bonyolult. Egy egyenes mozgás három dimenzióban elég egyszerű és mindössze a két dimenziós mozgásnak a kibővítése. Viszont az orientációnak a megvalósítása már trükkösebb. Ennek okán a fejlesztők gyakran használnak hibrid $2D$ és $3D$ geometriát amit $2\frac{1}{2}D$ neveznek. Ekkor $3D$-ben van megvalósítva a mozgás, és az orientáció egy változóként van megvalósítva, mint $2D$-ben.

Ez elég logikus döntés, mert a legtöbb játékban a gravitáció hatása alatt vannak a karakterek, és egy síkban mozognak. Ha csak a talajon mozognak, arra elég a $2D$ mozgás megvalósítása, bár az ugrás, létrára mászás, felvonó használata a $3D$ térben van megvalósítva. Még ha fel-le megyünk is, a karakter függőlegesen marad, és amíg így van addig csak az egyetlen dolog amire figyelnünk kell, hogy az orientáció felé legyen forgatva a karakter az y tengely mentén.

Persze ha egy repülőgép szimulátort írunk, akkor az orientáció nagyon fontos az MI-nek, ekkor $3D$-ben kell megírni. A másik véglet mikor egy teljesen sima felület a mozgástér és még ugrani sem lehet, ekkor $2D$ kell. De a legtöbb esetben a $2\frac{1}{2}D$ az optimális

megoldás.

Kinematika

Eddig minden karakternek volt két információja: a helyzete és az orientációja. Tudunk olyan algoritmust készíteni, ami ebből a két adatból ki tudja számítani a karakter sebességét, ha engedjük, hogy a kapott sebesség egyből változzon. Ez sok játéknál jó lenne, csak elég szürreálisan nézne ki. Newton törvényéből következik, hogy egy test nem tudja azonnal megváltoztatni a sebességét a valós világban. Ha egy karakter megy egy irányba, és hirtelen megváltoztatja a sebességét vagy irányát, az elég furcsán néz ki. Ahhoz, hogy egyenletesen mozogjon, olyan algoritmust kell használnunk, ami egyenletesen változtatja a sebességet, vagy a jelenlegi sebességet és gyorsulást használ. Emiatt a sebességet is tárolni kell a pozíció és az orientáció mellett.

A karakternek figyelnie kell a kerületi- és a szögsebességet is. A kerületi sebességnek van x és z komponense,$2\frac{1}{2}D$ -ben x,y,z.

A szögsebesség az, hogy az orientáció milyen gyorsan változik. Ez egy szám: radián per másodperc, amivel az orientáció változik.

A következőkben a szögsebességet „forgásnak” nevezzük, mivel az utal a mozgásra. A kerületi sebességet meg egyszerűen sebességnek. Így a következőképpen tudjuk egy karakter kinematikáját ábrázolni: a pozíció egy 2 vagy 3 dimenziós vektor, az orientáció egy lebegőpontos szám, a sebesség egy 2 vagy 3 dimenziós vektor és a forgás egy lebegőpontos szám.

A kormányozó viselkedések kinematikus adatokat használnak, és egy gyorsulást adnak vissza, amivel a karakter sebességét kell változtatni azért, hogy tudjon mozogni egy adott pályán. A kapott gyorsulást a következőképpen lehet reprezentálni: a mozgási gyorsulás egy 2 vagy 3 dimenziós vektor és a forgási gyorsulás egy lebegőpontos szám.

Feltűnhet, hogy nincs semmi, ami összekötné a karakter mozgását annak nézési irányával. A karakter lehet, hogy az x tengely felé néz és a z tengely felé mozog. Sok játéknál ennek nem így kellene viselkednie, az orientációja irányába kellene mozognia.

A legtöbb kormányozó viselkedés nem törődik az orientáció irányával. Egy jó megoldás lehet a problémára, hogy folyamatosan odaforgatjuk az orientációt a mozgás irányába, akár több képkockán keresztül. Ha van fizikai szimulációs réteg, akkor az elvégzi ezt helyettünk, egyébként manuálisan kell minden frissítéskor beállítani.

2.1.2. Kinematikus mozgás algoritmusok

A kinematikus mozgási algoritmusok statikus adatot (pozíció és orientáció, nincs sebesség) használnak és a kimenet a kívánt sebesség. A kimenet általában egy be- vagy kikap-

csolás, egy célirány, max. sebességgel mozgás, vagy egy helyben állás. Nem használnak gyorsulást, habár a hirtelen változás a sebességben finomítható, ha több képkockán keresztül változtatjuk.

Sok játék még egyszerűsíti a dolgokat és az orientáció irányát a karakter mozgási irányához kényszeríti. Ha egy helyben áll, akkor az utolsó mozgás irányba néz a karakter, egyébként ha mozog, akkor a visszakapott sebességhez állítja az orientációt.

A következőkben megnézünk két kinematikus algoritmust: megközelítés és bolyongás.

Megközelítés

A kinematikus megközelítés viselkedés két inputot kap: a karakter és a célpontja statikus adatát. Kiszámolja az irányt a karaktertől a célpontig majd beállítja az orientációt, végül ad egy sebességet.

Eltávolodás. Ha azt akarjuk, hogy a célpont elöl meneküljön, akkor egyszerűen megcseréljük a két pozíciót.

Megérkezés. A fent leírt algoritmusok üldözésre használatosak, mert sosem érik el a céljukat, csak folyton mennek feléjük, illetve menekülnek előlük. Problémát jelenthet ha mindig csak maximális sebességgel a cél felé megyünk, de nem állunk meg.

A probléma elkerülésére két választásunk van: egy nagyobb sugarú értéket adunk neki, hogyha annál közelebb ér a célponthoz akkor megállhat. A másik lehetőség, hogy lelassítjuk a mozgás sebességét, minél közelebb ér a célponthoz.

A második megoldás még mindig okozhat ide-oda mozgolódást, ezért inkább a kettő kombinálását érdemes használni. Ha közelebb érünk lassítunk és ha elérünk egy bizonyos távolságot akkor megállunk.

A megközelítés algoritmuson a következőképpen változtatunk:

Megnézzük, hogy a karakter a sugáron belül van-e, ha igen, akkor odaértünk, nem kell semmilyen visszatérési érték. Ha nem, akkor próbáljuk meg elérni egy adott időn belül. A konstans idő egy egyszerű trükk, hogy lelassítsuk a karaktert, ha eléri a célpontját.

Bolyongás

A kinematikus bolyongás viselkedés mindig a karakter aktuális orientációja irányába mozog, maximális sebességgel. Ez a viselkedés folyamatosan változtatja az orientációt, ennek következtében kanyarogni fog a karakter.

2.1.3. Kormányozó viselkedések

A kormányozó viselkedések kibővítik ez előző részekben nézett algoritmusokat sebességgel és forgással. Ezek jobban elfogadottak PC és konzol játékfejlesztésben. Néhány játékfajtában (pl. autós, versenyzős játékok) ezek dominálnak, a többiben még csak most kezdik igazán komolyabban használni.

Alap viselkedések

A legtöbb kormányozó viselkedésnek hasonló a felépítése. Bemenetként a mozgó karakternek a kinematikus adatát és a célpontról korlátozott mennyiségű információt kap. A célpont információja az alkalmazástól függ. Üldözésnél vagy elkerülésnél gyakran egy másik mozgó karakter. Akadály elkerülésnél ez a világnak egy ütközési geometria reprezentációja. De akár lehet egy pont is az útvonalkövetési viselkedésnél.

Az input nem mindig érhető el MI-barátságos formában. Az ütközés elkerülésnél el kell érnie az ütközési információt a pályának. Ez elég költséges lehet.

Jó pár kormányozó viselkedés célpontok egy csoportján dolgozik. Például a híres „flocking” viselkedés a csapat átlagos pozíciója irányába mozog. Ezekhez az algoritmusokhoz kell pluszban számolás, hogy összegezze a célpontokat olyan formában, amire az algoritmus már tud reagálni.

Észre kell venni, hogy a kormányozó viselkedéseknek nem próbálnak mindent helyettünk megoldani. Nincs olyan viselkedés, hogy üldözés közben figyeljen az akadályok elkerülésére, miközben kitérőket tesz, hogy felvegyen dolgokat. Minden egyes algoritmus csak egy dolgot tesz, és csak annyi inputot kap, amennyi szükséges hozzá, hogy megtegye azt. Ahhoz, hogy komolyabb viselkedéseket kapjunk, kombinálnunk kell az eddigieket, hogy együtt tudjanak dolgozni.

Megközelítés és Eltávolodás

A megközelítés megpróbálja „összepárosítani” a karakter pozícióját a célpont pozíciójával. Pont úgy, mint a kinematikus megközelítés algoritmus, megkeresi az irányt és a lehető leggyorsabban megy arra. Mivel a kimenet most egy gyorsulás, ezért gyorsulni fog amilyen gyorsan csak tud.

Értelemszerűen, ha folyamatosan gyorsul, akkor a sebessége egyre nagyobb és nagyobb lesz. A legtöbb karakternek van végsebessége, amilyen gyorsan tud menni, emiatt nem lehet a végtelenig gyorsulni. A maximum lehet explicit, egy változóba vagy konstansba tárolva. A jelenlegi sebességet meg folyamatosan ellenőrzi, hogy túllépte-e már a végsebességet, és ha igen, akkor a végsebességre kell visszatenni.

Eltávolodás. Az eltávolodás az ellentettje a megközelítésnek, minél messzebbre megy a célponttól. Ugyanúgy, mint a kinematikus eltávolodásban, itt is csak a két pozíciót kell megcserélni.

Megérkezés

A megközelítés mindig a célja fele fog mozogni, a lehető legnagyobb gyorsulással. Ez rendben van, amíg a célpontja folyamatos mozgásban van, és teljes sebességgel kell üldöznie. Amikor eléri a célpontját, túlhalad rajta, megfordul és ismét üldözni próbálja, nagyjából csak körözni fog, de nem áll meg.

Ha elérné a karakter a célpontját, le kellene lassítania, hogy pontosan a jó helyen álljon meg, ahogyan a kinematikus megérkezés algoritmusnál láttuk.

A dinamikus megérkezés egy kicsit komplexebb mint a kinematikus megérkezés volt. Itt két kör sugár adott. Az érkezési sugár, mint azelőtt, ami olyan közel engedi a ponthoz a karaktert, amennyire csak lehet anélkül, hogy az elkezdene forgolódni. A második sugár sokkal nagyobb, amikor ezt áthaladja a karakter, akkor kezd el lassítani. Az algoritmus számolja ki az ideális sebességet a karakternek. A külső körnél ez a végsebesség, a érkezési pontnál pedig nulla (mert az akarjuk, hogy megálljon). A kettő között folyamatosan változik, a két távolságtól függően.

Elindulás. Az megérkezés ellentettje az elindulás lenne. De semmi értelme, hogy implementáljuk. Ha el akarunk hagyni egy célpontot, azt nem lassan akarjuk, hanem azonnal. Ezért nincs értelme lassan gyorsulni, hanem egyből a maximális gyorsulást használni. Technikailag a eltávolodás a megérkezés ellentettje.

Igazítás

Az igazítás megpróbálja egyeztetni a karakter orientációját a célponttal. Nem törődik a karakter sebességével vagy helyével, illetve a célpontéval. Csak a forgatással törődik, nincs hatással a gyorsulásra.

Hasonlít a megérkezésre. Megpróbálja elérni a célpont orientációját, és megpróbál nulla forgást tartani ha odaért. Ahhoz, hogy megkapjuk az aktuális forgásirányt, nem elég kivonni a karakter orientációját a célpontéból, át kell alakítanunk egy intervallumon belül az eredményt: $(-\pi, \pi)$ radián.

Az ellentét. Az igazításnak nincs olyan, hogy ellentettje. Mivel 2π-nként ismétlődik az orientáció, ezért mindig ugyanazt kapnánk, ezért a legegyszerűbb, ha hozzáadunk π-t és akkor az ellenkező irányba fogunk nézni.

Delegált viselkedések

Megnéztük a viselkedések alap építőkockáit, amik segítségével a többi viselkedést el lehet készíteni. A megközelítés, eltávolodás, érkezés, igazítás számolásait fogjuk felhasználni a többi algoritmus készítésekor.

Minden viselkedés a következő struktúrát követi: kiszámolják a célpont helyzetét, illetve orientációját, felhasználják az előbb említett viselkedéseket.

Üldözés és Elkerülés

Tehát eddig kizárólag a pozíció alapján mozogtunk. Ha egy mozgó célpontot üldözünk, a folyamatosan a jelenlegi pozíciója felé mozgás nem lesz elegendő. Mire elérjük azt, addigra az már megváltoztatta a helyét. Ez nem túl nagy probléma, ha közel vagyunk, és képkockánként újraszámoljuk a helyzetét. Egyhamar utol fogjuk érni. De ha nagyon messze van, akkor feltűnően rossz irányba fog mozogni.

Ahelyett, hogy a jelenlegi helyzetére koncentrálnánk, inkább a jövőbeli helyzetét próbáljuk meg meghatározni és oda mozogjunk. Rengetegféle algoritmussal valósíthatjuk ezt meg, de a legtöbbjük nem vezetne jó megoldásra. Többféle kutatás folyt az optimális útvonal jövendölés és optimális stratégia az üldözött karakternek kérdésében (mai napig egy aktív téma a katonaságnál pl. rakéták elkerülése). Craig Raynolds eredeti megközelítése sokkal egyszerűbb: feltesszük, hogy a célpont folytatni fogja a mozgást a jelenlegi irányába ugyanezzel a sebességgel. Ez rövid távon jól működik, és nem néz ki nagyon rosszul nagyobb távolságban sem.

Az algoritmus kiszámolja a távolságot a karakter és a célpont között és, hogy mennyi időbe fog telni, ha maximális sebességgel halad. Felhasználja a kapott időintervallumot, hogy előre gondolkozzon. Kiszámolja, hogy a célpont helyzete hol lenne, ha tovább mozogna ekkora sebességgel. A kapott eredményt felhasználja egy alap megközelítés viselkedés.

Ha a karakter lassan mozog, vagy a célpont nagyon messze van, a jövendölési idő nagyon nagy lehet. A célpont elég kis eséllyel mozog mindig ugyanúgy, ezért érdemes valamilyen korlátot szabni, hogy mennyire előre jövendöljön. Ha a jövendölési idő nagyobb ennél, akkor a maximális értéket használja.

Elkerülés. Az üldözés ellentettje az elkerülés. Hasonlóan kiszámoljuk a jövendölt pozíciót, csak megközelítés helyett eltávolodást használunk.

További problémák. Ha gyorsabb a karakter az üldözött célpontnál, akkor túl fog haladni rajta. Ezt elkerülhetjük, ha megközelítés helyett megérkezést használunk.

Nézés

A nézés algoritmus miatt néz a karakter a célpontjára. Az igazítás viselkedést használja fel, hogy megkapja a forgást, de a célpont orientációját számolja ki előbb.

Oda nézünk ahová megyünk

Eddig feltételeztük, hogy nem feltétlen kell abba az irányba néznünk, amerre megyünk. Habár sok esetbe szeretnénk, ha ez így lenne. A kinematikus mozgásnál közvetlenül állítottuk be. Az igazítás felhasználásával eltudjuk érni, hogy odanézzen. Ennek segítségével természetesebbnek tűnik a mozgás, ha mindig arra nézünk amerre megyünk, kivéve persze ha valamilyen pl. helikopterrel megyünk, mert ott az oldalazás természetes.

Ezt nagyon hasonlóan az előző nézés viselkedéshez valósíthatjuk meg. A célpont orientációja a karakter jelenlegi sebességével számolva kapható. Ha nincs sebessége, akkor a jelenlegi orientációt használjuk.

Bolyongás

A bolyongás viselkedés az, amikor a karakter céltalanul mozog.

A kinematikus bolyongásnál adott időnként változtattuk véletlenszerűen az irányt. Ez azt eredményezi, hogy a karakter megy előre, csak a kanyarodás rosszul néz ki. Ezt egy egyszerű kis trükkel meg tudjuk oldani: egy kört teszünk a karakter elé egy adott távolságra és a körvonalra teszünk véletlenszerűen célpontot.

Útvonalkövetés

Eddig olyan viselkedéseket néztünk ami egy nulla vagy egy célpontot várt. Az útvonalkövetés olyan kormányozó viselkedés, ami egy teljes útvonalat vár paraméterül. Az útvonalkövetés viselkedésnek végig kell mennie az útvonalon.

Az útvonalkövetés viselkedét általában úgy szokták megvalósítani, hogy felhasználják a megközelítés viselkedést. Az megérkezés nem lenne jó, mert a köztes pontokon nem akarunk megállni, csak az utolsón.

A célpont két lépésben van kiszámítva. Először a jelenlegi karakterpozícióhoz képezzük a legközelebbi útvonal pontot, másodszor a kiválasztunk egy pontot, ami tovább van az útvonalon, mint a legközelebbi, egy adott távolsággal. Ahhoz, hogy megváltoztassuk a mozgás irányát az úton, egyszerűen meg tudjuk változtatni az előjelét a távolságnak.

Elválasztás

Az elválasztás viselkedés gyakori a tömegszimulációknál, mikor a karakterek egy része nagyjából egy irányba megy. Megpróbálja a karaktereket egymástól távolabb tartani. Ez nem működik túl jól, ha a karakterek egymás útját keresztezve mozognak. Ekkor a lejjebb kifejtett ütközés elkerülés viselkedést érdemes használni.

A legtöbb esetben az algoritmusnak nincs kimenete. Nem ajánl semmilyen útvonalváltoztatást. Ha észreveszi, hogy két karakter túl közel kerül egymáshoz, hasonló módon az elkerüléshez, elszeparálja a karaktereket egymástól.

A szóródás erőssége többféle módon megadható, legelterjedtebb az egyenletes és a inverz négyzetes.

Vonzás. Ha az inverz négyzetnél negatív értéket állítunk be a konstansnak, akkor egy vonzási erőt kapunk. Egy bizonyos körön belül egymáshoz fognak vonzódni a karakterek, bár ez ritkán használatos dolog.

Függetlenség. Az elválasztásnak nincs nagy haszna magában. Egy idő után a karakterek csak forgolódni fognak, de mozogni nem. Viszont más viselkedésekkel kombinálva már működőképes lesz.

Ütközés elkerülés

Bizonyos területeken előfordul, hogy sok karakter egyszerre mozog egymás útját keresztezve. Ezeknek nem szabad egymáson átmenni, hanem el kell kerülniük egymást.

Egy egyszerű megközelítése a problémának, ha az elkerülés vagy a eltávolodás egy változatát használjuk, hogy csak akkor lépjen működésbe, ha a karakterünk előtt a célpont egy adott szögön belül van.

Ha több célpont is van előtte, mindent el kell tudnia kerülni. Sajnos annak ellenére, hogy ezt a megközelítést könnyű megvalósítani, a gyakorlatban nem fog túl jól működni.

2.2. Útvonalkeresés

A karaktereknek általában tudni kell mozogni a pálya egyik pontjából a másikba. Valamikor ez a mozgás előre definiált a programozók által, mint egy járőröző mozgás, amit egy őrnek vakon kell követnie. A fix utakat könnyű implementálni, de könnyű ezeket átverni, például ha egy objektumot teszünk az útra, akkor könnyen elakadhat benne.

Az összetettebb karakterek nem tudják előre, hogy merre mennek. Egy stratégiai játékban a játékos a pálya bármelyik részére tud egységet küldeni, egy őrnek oda kell

jutni a legközelebbi riasztóhoz, ha betörőt lát.

Minden egyes karakter MI-nek ki kell tudnia számolni egy megfelelő utat a pályán, amivel el tud jutni a céljához. Az útnak ésszerűnek és rövidnek kell lennie amennyire csak lehet.

Ez az útvonalkeresés, gyakran útvonaltervezésnek is nevezik, és ez mindenhol jelen van a játék MI-ban. A modellünkben ez a döntéshozás és mozgás közti határon van. Gyakran arra használják, hogy találjon egy utat, ami megmondja merre menjünk a célunkhoz. A cél eldöntését az MI egy másik része végzi, az útvonalkeresés csak utat adja meg. De az útvonalkeresést lehet használni vezérlésre is, döntéshozásra, hogy merre menjünk és, hogy hogyan jussunk oda.

A játékok nagy része egy algoritmusra alapoz: ez az A*. Ezt könnyen és hatékonyan lehet implementálni, de ez nem tud a pálya adatain közvetlenül dolgozni. Szüksége van egyfajta struktúrára, amin az adatokat praktikusan eléri: egy (irányított) nem-negatív súlyozott gráf. Meg lesz említve a Dijkstra algoritmus is, de azt inkább a taktikai stratégiai döntéshozásban használják.

2.2.1. Az útvonalkereső gráf

Mivel egyik algoritmus sem tud egyből a geometriai pontokon mozogni ezért egy gráfon szokás reprezentálni, ezt útvonal-keresési, vagy navigációs gráfnak nevezik. A gráf egy matematikai struktúra, ami két különböző elemet tartalmaz: a csúcspont és az élt. A csúcspontokat kötjük össze az élek segítségével. Az útvonalkeresésnél minden egyes csúcs általában egy területet jelöl a pálya szerkezetben, mint egy szoba, vagy egy platform. A köztük lévő kapcsolatok mutatják, hogy az egyikből elérhető a másik. Hogy az egyik helyről a másikba jussunk, ezeket a kapcsolatokat nézzük. Ha az egyikből közvetlen elérhető a másik, akkor egyszerű a dolog, de ha nem, akkor ezeken a kapcsolatokon keresztül kell végigmennünk, hogy elérjük a célunk.

Nem-negatív súlyozott gráf

Egy súlyozott gráf ugyanúgy épül fel mint a sima, csak az élekre úgynevezett súlyokat teszünk. Ez egy költség, ami a két út között van. Általában ez a távolságot, vagy az időt jelöli az útvonalkeresésnél. Nincs sok értelme, hogy negatív költséget használjunk. Nincs negatív távolság, vagy idő két pont között. Viszont egy súlyozott gráfban lehetnek negatív súlyok, ezért kik kell kötnünk, hogy csak nem negatív súlyokat használhatunk. A Dijkstra és A* algoritmusokat csak nemnegatív súlyokat lenne szabad használni, különben végtelen ciklusba eshet.

2.2.2. Dijkstra

A Dijkstra-algoritmus egy gráfkeresési algoritmus, amivel a legrövidebb út problémát lehet megoldani olyan gráfokban, ahol az élek nem-negatív súlyúak. Egy kezdeti pontból kiindulva a gráfban, az algoritmus megkeresi a legkisebb költségű utat ez a pont, és minden más pont között. Arra is lehet használni, hogy megkeressük a legrövidebb utat egy kezdőpont és egy végpont között a gráfban.

Az algoritmus röviden: kezdetben minden pontot nem látogatott pontnak jelöl. Választ egy pontot a legkisebb költségű távolsággal a még be nem járt pontok közül, kiszámolja a távolságot minden még be nem járt szomszédjához, és frissíti a szomszédja távolságát, ha kisebb. Látogatottként jelöli a pontot, ha végzett a szomszédjaival.

2.2.3. A*

Az A* algoritmus széles körben használt útvonalkeresésben és gráfbejárásra. Gyakran használják a legkisebb költségű utak megtalálására egy kezdeti pont és egy végpont között. Az A* a gráf bejárásánál a várható legkisebb teljes költségű vagy távolságú utat követi, egy rendezett prioritás sort használva. Különböző heurisztikát használva választja ki, melyik pontot válassza következőnek. Ha heurisztikának a 0 távolságot választjuk, akkor a Dijkstra keresést kapjuk. Gyakori heurisztikák: euklideszi távolság, Manhattan távolság.

2.3. Döntéshozás

Ha egy játékost megkérdeznél, mit gondol az döntéshozásról, azt mondaná: a karakter eldöntse, mit tegyen. Megvalósítani ezt a döntést (mozgás, animáció, stb.) magától értetődő.

A valóságban a döntéshozás egy tipikusan kis része a szükséges erőfeszítéseknek, ami egy MI-hez kell. A legtöbb játék egyszerű döntéshozást használ, mint döntési fák és állapotgépek. A szabály-alapú rendszerek ritkák, de nem elhanyagolhatóak.

Habár sokféle technika van, mind hasonlóképpen működik. A karakter feldolgozza információk egy halmazát, amiből generál tevékenységeket, amelyeket végre akar hajtani. A döntési rendszer bemenete, amit a karakter tud, és a kimenet egy tevékenység, amit végre akar hajtani. A bemeneti tudás lebontható külső és belső tudásra. A külső tudás amit a karakter tud a környezetéről: másik karakterek helyzete, a pálya kinézete, egy irány amerről hang jön, és így tovább. A belső tudás a karakter információi: az élete, a fő célja, mit tett pár másodperccel ezelőtt, és a többi.

Ennek megfelelően egy tevékenységnek két összetevője van: egy olyan művelet ami

a külső állapotát változtathatja a karakternek, vagy ami a belsőre van hatással. A belső változtatások kevésbé egyértelműek a játékoknál, de jelentősek bizonyos döntési algoritmusoknál. Ezek a karakter véleményére, céljaira lehetnek hatással.

A döntéshozásnak nagyon sokféle megvalósítása lehet: döntési fák, állapotgépek, viselkedési fák, fuzzy logika, Markov rendszer, cél-orientált viselkedés, és még sorolhatnám. Ezekből a kettőt nézünk most meg, amelyeket elég egyszerű implementálni, és mégis elég látványos eredményeket tudnak produkálni.

2.3.1. Döntési fák

A döntési fák gyorsak, könnyű megvalósítani, és egyszerű megérteni őket. Ez a legegyszerűbb döntéshozási technika, amit megnézünk. Széles körben használtosak a karakter irányításánál és egyéb játékbeli döntéshozásnál, mint például az animációknál.

A probléma

Adott információk egy halmaza, ezek segítségével kell kiválasztanunk a lehetséges tevékenységek közül a megfelelőt. A leképezés a bemenet és kimenet között elég komplex lehet. Ugyanaz a cselekvés lehet különböző inputokra, de lehet, hogy egy kis eltérés az egyik bemeneti értéken eredményezhet a viselkedésben elég nagy változást. Kell egy metódus, ami könnyedén csoportosíthat sok bemenetet egy cselekvéshez, miközben a bemeneti értékek jelentős mértékben irányítják a kimenetet.

Az algoritmus

A döntési fát döntési pontok összekötésével kapjuk. A fának van egy kezdő döntési pontja, a gyökér. Innen kezdi a döntést, és egy a lehetséges kimenetekből ki lesz választva.

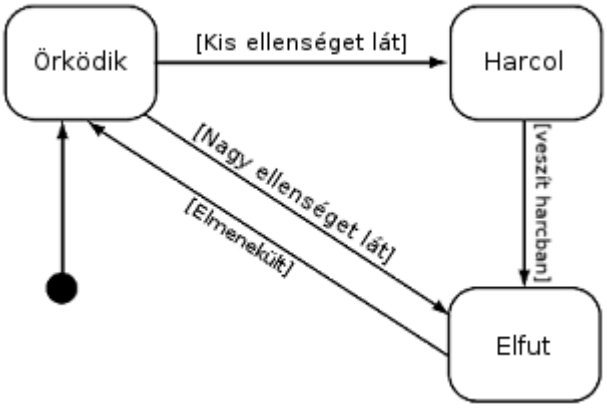
Minden döntés a karakter tudására van alapozva. Mivel a döntési fák egyszerű és gyors mechanizmusként vannak használva, ezért a karakterek a globális játékállapotot használják ahelyett, amit személyesen tudnak.

Az algoritmus addig halad a fán döntéseket hozva, amíg el nem jut egy levél pontig. Amikor elér egy tevékenységhez, azt egyből végrehajtja.

A legtöbb döntési fa egyszerű döntésekből épül fel, tipikusan két döntéssel, így egy bináris fával könnyen implementálható. Egy tevékenység több levélen is szerepelhet.

Döntések

A döntések a fán egyszerűek. Tipikusan egy értéket vizsgálnak és nem használnak semmilyen logikai kapcsolatot (logikai ÉS, VAGY, stb.).



2.2. ábra. Egy viselkedése állapotgéppel megvalósítva

Attól függően, hogy milyen adattípust használunk, különböző vizsgálatok lehetségesek. A következő táblázat egy lehetséges megvalósítást ír le.

Adattípus	Döntés
Logikai	igaz-e az érték
Felsorolás	egyezik-e az egyik értékkel
Szám (egész vagy lebegőpontos)	egy adott értéken belül van-e
2D vektor 3D vektor	A vektor hossza az adott távolságon belül van-e

2.3.2. Állapotgép

Gyakran a karakterek csak egy bizonyos módon cselekednek, ameddig valami meg nem változtatja az állapotukat. Képzeljünk el egy katonát: ő őrködni fog, és addig ugyanabban az állapotban marad, amíg meg nem lát egy ellenséget. Ha ez megtörténik, akkor megváltozik a viselkedése, egy másik állapotba lép át. Ezt a fajta döntéshozást is meglehet valósítani döntési fákkal, de egyszerűbb állapotgéppel megvalósítani.

Alap állapotgép

Minden állapotgépben a karakterek egy állapotban vannak. Alapból a viselkedések és cselekvések kapcsolatban vannak az állapotokkal. Amíg egy karakter ugyanabban az állapotban van, addig ugyanazt a tevékenységet fogja folytatni.

Az állapotok átmenetekkel vannak összekapcsolva. Minden átmenet egy állapotból bizonyos feltételekkel egy másikba állapotba megy. Ha a játék megállapítja, hogy valamilyen feltételnek megfelelt, akkor átlép abba az állapotba. (Amikor egy feltétel teljesül akkor azt mondjuk „trigger”, és mikor a feltétel átmegy az állapotba akkor „fired”.)

Véges állapotú gép

A játék MI-ben az ilyen felépítésű állapotgépeket véges állapotú gépeknek nevezzük. A következő részekben ennek implementációját mutatjuk be. A véges állapotú gépnek elég gyakori a használata, nem csak játék programozásban, ezért mikor véges állapotú gépet mondunk, mindig erre a struktúrára fogunk utalni.

A probléma

Szeretnénk egy olyan általános rendszert, ami támogatja a állapotgépet bármilyen átmeneti szabállyal. Az állapotgépünk a fent leírt struktúrának felel meg, és csak egy állapotot fogad el egy állapot egy időben.

Az algoritmus

Generikus állapot felületet fogunk használni, hogy bármilyen kódot tudjunk használni. Az állapotgép nyilvántartja a lehetséges állapotok halmazát, amiben a jelenlegi is benne van. Emellett minden egyes állapothoz az átmenetet is tároljuk. Minden átmenet szintén egy generikus felület, ahol a megfelelő feltételeket adhatjuk meg. Ez egyszerűen szól a állapotgépnek, ha a feltétel teljesül, vagy éppen nem.

Minden iterációban (általában képkockánként) meghívódik az állapotgép frissítés funkciója. Megnézi, hogy a jelenlegi állapotból elérhető átmenetek feltételei teljesültek-e. Az először teljesülő átmenet hajtódik végre. A metódus azután elkezdi végrehajtani az átmenethez szükséges algoritmusokat. Ha mindennel kész, akkor átvált az új állapotba.

2.4. Taktikai és Stratégiai MI

Az előző részben leírt döntéshozási technikáknak két fontos korlátja van: egy karakter használatára van kitalálva, és nem tudja a jelenlegi tudásából megjövendölni az egész helyzetet. Ezen megszorítások a taktika és stratégia témakörbe tartoznak. Ez az eddig használt MI modellünkben a harmadik nagy rész. A most következő technikák csak egy kis része a létező technikáknak.

2.4.1. Taktikai Útvonalkeresés

A taktikai útvonalkeresés egy elég felkapott téma a játékfejlesztésben. Egészen lenyűgöző eredményeket adhat: mikor a karakterek mozognak a játékban, számításba veszik a taktikai környezetüket, fedezékben maradnak, elkerülik az ellenséges támadási vonalakat és a rajtaütési pontokat.

A taktikai útvonalkeresésről gyakran úgy beszélnek, mintha jelentősen komplexebb vagy bonyolultabb lenne, mint a hagyományos útvonalkeresés. Ez sajnálatos, pedig szinte nem is különbözik a hagyományostól. Ugyanazokat az útvonalkeresési algoritmusokat használjuk ugyanolyan gráfokon. Az egyetlen változtatás a költségfüggvény, ugyanis ez kiegészítődik taktikai információval, hasonlóan mint a távolság vagy az idő.

A költségfüggvény

A mozgás költsége a pontok között a gráfban a távolságon/időn kell, hogy alapuljon, és azon, hogy taktikailag mennyire értelmes a művelet. A kiszámítása a következőképpen történhet: vesszük a nem taktikai költségét a függvénynek (távolság/idő/stb.), és hozzáadjuk az összes lehetséges taktikát, ahol ez egy taktikai minőség érték és egy súlynak a szorzata.

Az egyetlen nehézség, hogy hogyan tároljuk a taktikai információt. Általában a helyszínek alapján tároljuk. Ezt át kell alakítani, hogy a pontok közötti kapcsolatokban használhassuk. Általában a helyhez tartozó kapcsolatok számának átlagával osztjuk a taktikai minőség értékét. Az így kapott érték viszont elég gyenge eredményeket adhat. Ezért különböző súlyokat kell a kapcsolatokhoz hozzárendelni.

Ha egy taktikai érték magas, akkor azt nagy valószínűséggel a karakter elkerüli. Az ilyen pontok általában valamilyen veszélyforrást jelentenek, mint egy rajtaütési pont. Ez jól működik statikus jellemzőkkel, mint a domborzat vagy a láthatóság. Viszont nem tud számításba venni dinamikus jellemzőket: fedezékbe menni egy ismert ellenféltől. Ilyenkor mindig ki kell számolni újra a költséget az élhez. A költségszámítás elvégzése ilyenkor eléggé lelassítja az útvonalkeresést.

Ha a költségeket változtatjuk a kapcsolatok közt, akkor könnyen előfordulhat, hogy egyes heurisztikák érvénytelenek lesznek. Euklideszi távolság használatánál előfordulhat, hogy érvénytelen lesz, mivel ott csak a távolságot használjuk, a taktikai minőséget nem. Ennek egy megoldása lehet, hogyha skálázott Euklideszi távolságot használunk.

2.4.2. Koordinált Akció

Az eddigi technikák egy karakter irányítására vonatkoztak. Egyre gyakrabban látjuk játékokban, hogy van ahol a karakterek együttműködnek, hogy bizonyos feladatot el tudjanak végezni. Mostanság még az is kezd előtérbe kerülni, hogy a játékossal együttműködjenek az MI-k. Már lassan nem lesz elég, hogy csak az utasításaira reagáljanak.

A karaktereknek érzékelniük kell a játékos szándékát, és úgy viselkedniük, hogy azzal segítik a játékost. Ez sokkal nehezebb, mint csak az egyszerű együttműködés. Egy csapat MI karakter el tudja egymásnak mondani, mit akar véghez vinni, viszont a játékos csak a

cselekedetein keresztül tudja jelezni azt, és ezt az MI-nek meg kell tudnia érteni. Ez egy igen fontos téma a mai játék MI-ben.

A csapat döntéshozás nagyon hasonló az egyéni döntéshozáshoz, csak itt figyelembe kell venni különböző taktikai analíziseket, vagy taktikai útvonaltervező algoritmusokat. Ezzel lehet meghatározni, hogy hova menjenek a karakterek vagy épp mikor maradjanak fedezékben.

A csapatmozgás megoldható a már korábbiakban megnézett mozgási viselkedésekkel. Ezek képesek több karaktert egyszerre mozgatni. Egy magasabb szintű rendszer vezérli az egész csapat mozgási viselkedéseit. Alacsonyabb szinten, a karakterek külön külön mozognak, és próbálják tartani az alakzatukat, miközben elkerülik az akadályokat és számításba veszik a környezetüket. Amióta a formációs mozgások egyre jobban terjednek, egyre gyakoribb az a megvalósítás, hogy nincs semmilyen mozgási algoritmus a magasabb szinteken a hierarchiában, és a legalsóbb szintek döntései mozgási döntésekké alakulnak.

Az útvonalkeresés csoportoknak semmivel sem nehezebb, mint az egyéni karaktereknek. A legtöbb játékban figyelembe veszik azt is, hogyha sok karakter akar átjutni egy szűk helyen, akkor egyik se akadjon el.

3. fejezet

Gyakorlati megvalósítás

3.1. Bevezetés

A dolgozatomban egy játékot fogok bemutatni, amit egy játékmotor segítségével készítettem el. A játék a Warcraft 3 játék Warlock nevű pályája. A játékmotor a Unity3D. A bevezetőben ezeket ismertetem.

3.1.1. Warcraft 3 Warlock

Rövid leírása: A warlock egy gyors ütemű, mégis rendkívül stratégiai aréna pálya, fejlett fizikai motorral és rengeteg varázslattal, tárggyal és fejlődési lehetőséggel. Több mint 4 éves a játék, és még mindig elég aktívan fejleszti a két készítője, Zymoran és Demestotenes. A www.warlockbrawl.com honlapon minden információ megtalálható róla.

3.1.2. Unity3D

Rövid leírás: Egy 3D-s játékmotor, aminek van ingyenes változata. Ezt használtam én is. Nagyon aktív közösséggel rendelkezik, rengeteg anyagot lehet róla találni, és mostanság nagyon felkapott. Javascript, C# és Boo nyelven lehet benne programozni. Én a C#-ot választottam. A www.unity3d.com honlapon minden információ megtalálható róla.

3.1.3. A felépítés

Az elkészített játék elég összetett, és a megértéséhez némi bevezetőre van szükség. Először egy átlagos játék felépítését nézzünk meg. Minden játéknak van egy, az egészet összetartó része, itt én játék osztályként fogok rá hivatkozni. Ez tartja egyben a komponenseket, és folyamatosan kezeli az adatokat, interakciókat. Minden játéknak van legalább egy játékosa, aki valamilyen módon interakcióba léphet a játékkal. Nem feltétlen

kell tudnia beleszólni a dolgokba, lehet, hogy csak szemlélőként lesz jelen. A játék osztály kezeli a játékos interakcióit. Általában a játékban vannak egységek (ez lehet egy háborús játékban egy egész hadsereg, de lehet csak egy karakter, aki a játék főhőse), amikkel a játékos interakcióba léphet. Ha nem is tudja irányítani ezeket, attól még tud róluk, látja őket. Hogy ezeket az egységeket a játékba meg tudjuk jeleníteni, kell valamilyen grafikus szoftver vagy könyvtár, mint az OpenGL. Itt ezt a részt a Unity3D biztosítja. Az általában nem elég, hogy csak úgy a semmiben legyenek az egységek, ezért szükségünk lesz egy pályára, amire tehetjük őket, esetleg még mozoghatnak is rajta. A következőkben a pálya osztály lesz a felelős ezért. Tehát eddig amink van: egy pályánk, ezen egységek, és egy játék osztály, ami összetartja ezeket. Egy egyszerű játékhoz ennyi elég is lenne, csak ki kellene bővíteni a játék osztályt, hogy a játékos inputjára reagáljon. Ha viszont azt szeretnénk, hogy a gép magától is műveljen dolgokat, akkor egy mesterséges intelligenciát kell írnunk hozzá. Ennek a mesterséges intelligenciának a feladata az egységek vezérlése. Nem feltétlen kell bonyolultnak lennie, hogy működjön. Például ha az egységek mindig az egér kurzor elől próbálnának menekülni, akkor máris egy nagyon alap MI-t készítettünk. Viszont ha valami komolyabb dolgot szeretnénk, akkor már valamivel összetettebbet kell készítenünk.

Az általam készített játékban az egységeknek saját öntudatuk van, célokat tűznek ki maguk elé, és meg is valósítják őket, célpontot választanak a többi egységből, és támadják őket, az általuk jónak vélt varázslattal, és még memóriájuk is van. Ez már egy komplexebb MI, de a játék fentebb említett elemei megmaradnak, és ezekkel kibővülnek.

3.2. A játék

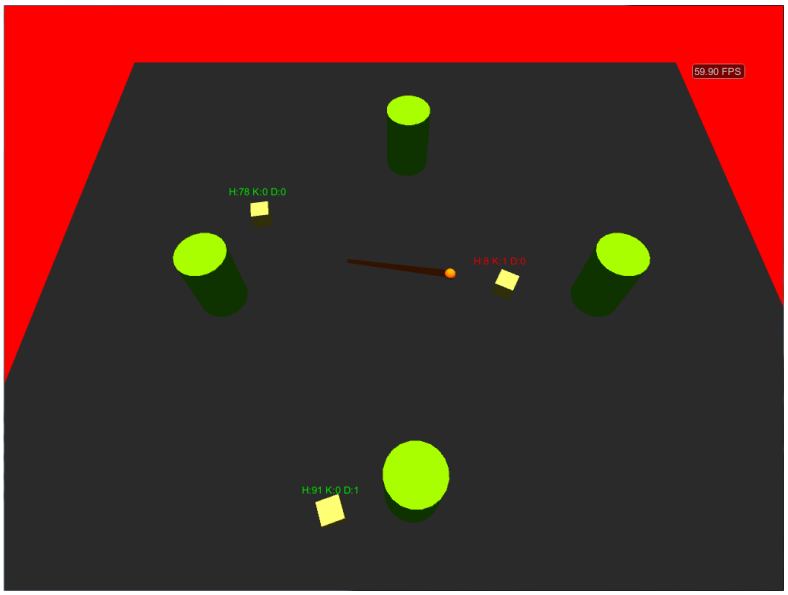
3.2.1. A játék leírása

A későbbi fejezetekben minden ki lesz fejtve részletesen, most csak a kiindulási alapot mutatom be, hogy legyen elképzelés a játékról.

Felépítés. Ez egy egyszerűbb 3D-s játék, ami elég komplex ahhoz, hogy bemutassak az előző részben említett különböző technikákat, valamint még azok kibővítését. A játék az „Umba” kitalált nevet kapta, ami a játékban lévő egységek neve.

Az alap felépítés: adott egy pálya, amin lehet mozogni, ennek a külső része körben láva, a közepe pedig normális talaj. A belső részen oszlopok vannak elhelyezve, hogy akadályozzák a mozgást.

A pályára a játékos tud adni egységeket, illetve ezeket el is tudja távolítani. Ezek a pályán mozogva egymás ellen harcolnak, a fő céljuk, hogy minél több ellenfelet semmisítsenek meg. A játékos alapból csak egy figyelő, aki belátja az egész pályát és így tudja,



3.1. ábra. A készített játék

mit művelnek az egységek. Ha úgy dönt, ő is akar játszani, akkor kiválaszt egy egységet, és akkor az nem magától fog cselekedni, hanem a játékos utasításait követi.

Varázslat. A játékban varázslatok vannak, méghozzá kettő darab, amiket az egységek tudnak használni: tűzgolyó és meteor. A tűzgolyó egy olyan varázslat, amit az egység tud varázsolni, ekkor megjelenik a pályán és ahova lövi, oda fog repülni. A meteor az előzőhöz nagyon hasonló, annyi különbséggel, hogy nem a használója pozíciójából indul, hanem jóval fölötte, az „égből", és szintén a célpontja irányába halad. Amikor az ellőtt varázslatok valaminek nekiütköznek, legyen az oszlop, vagy egység, elpusztulnak. Ha egy varázslat egy egységbe csapódik, akkor ellöki azzal az erővel, ahogy becsapódott, és meg is sebzi. Ha egy akadályba, akkor nem történik semmi más.

A 3.2 ábrán látható a játékom fontosabb osztályairól egy osztálydiagram. Ez bővül ki még az MI osztályokkal a bemutatás során.

3.2.2. Egység

A játékban lévő objektumok alaposztálya. Tartalmazza az objektum helyzetét 2 és 3 dimenziós vektorban, és a méretét. A mozgás 2 dimenzióban történik, de a forgatás, célzáshoz helyenként kell a 3 dimenzió. A méret az egység 3 dimenzióban való kiterjedése. Mivel csak pontonként van a pozíció tárolva, így ez is kell, hogy meg tudjuk mondani, mekkora területet foglal el.

A 3D-s játéktérben vannak az objektumok, és ezekhez hozzá lehet rendelni osztályokat. Unityban, hogy egy osztályt hozzá tudjunk rendelni egy objektumhoz, a MonoBe-

havior osztályból kell származnia. Így ennek az osztálynak is ez a ősosztálya. Mivel csak ennyi attribútuma van, ezért jól látható, hogy ebből az egység osztályból fog minden öröklődni. Viszont ennyitől még nem tud mozogni egy egység, ki kell terjesztenünk a mozgáshoz.

3.2.3. Mozgó egység

Az egység osztály leszármazottja, ami tovább bővíti, hogy képes legyen mozogni. Ehhez szükséges a sebességet és az irányt tárolni. Ehhez még hozzájön az egység tömege, mivel a játékban a fizikának szerepe van. Azért, hogy ne tudjon azonnal bármerre fordulni, és a pálya egyik pontjából a másikba menni egy pillanat alatt, korlátoznunk kell a mozgás- és forgás sebességet. Ezt a maximális mozgási- és forgási sebesség adattagokkal tesszük meg. Mivel ez 3D térben történő forgás, így az egység orientációját is tároljuk. Ez egy Quaternion típusú változó.

3.2.4. Bot

Mi is a bot?

A játékban szereplő egységeket, amiket a számítógép irányít, botnak nevezünk. A bot elnevezést nem feltétlen egy darab egységre szokták használni, hanem a MI vezérelt játékosra. Itt az osztályt botnak nevezzük el, mivel ezeket a gép irányítja. Igaz, hogy a játékos is át tudja venni felettük az uralmat és játszhat velük, az alap elképzelés szerint egymás ellen játszanak.

Állapotok és a pontrendszer

A bot mindig egy állapotban van, ami három fajta lehet: halott, születik és él. Ha halott, akkor nincs a pályán és irányítani sem lehet játékosként, valamint semmilyen MI tevékenységet nem hajt végre. A botnak van egy életpontja, illetve egy maximális életpontja ami fölé nem mehet. A minimális a 0, ha eléri ezt, akkor az állapota a halottra vált, és az ezt követő frissítési ciklusban eltávolítódik a pályáról. Ha eltávolítottuk, akkor a születési státuszba vált az állapota. A születési állapotban lévő botokat egy előre meghatározott születési pontok halmazából kiválasztott pontba tesszük, visszaállítjuk az életpontjait maximálisra, és az eddigi céljait, célpontját töröljük. Ezeket a frissítéseket a Játék osztály végzi. Számontartjuk a pontok és halálok számát. Ha meghal a bot, akkor a halálainak számát eggyel megnöveljük, illetve ha a bot öl meg valakit, akkor a pontjainak számát növeljük.

Az ilyen típusú játékokban egy adott ideig megy a harc, majd akinek a legtöbb pontja lesz az idő leteltekor, az nyeri a játékot. Mivel itt nem fontos ki nyer, csak a botok intelligenciájának a bemutatása, így itt nincs időkorlát, a végtelenségig tudnának játszani.

A pálya körül láva van, amire ha rálép egy bot, akkor csökken az életpontja. Ekkor egy változó jelzi a botnak, hogy a lávában áll. Ha valaki eltalálja egy varázslattal a botot, akkor szintén egy változó jelzi neki, hogy megsebesült, és mivel feltehetően arrébb is lökte, ezért tervezze újra a jelenlegi döntéseit.

Az öntudat

Egy logikai változóval állíthatjuk be hogy a játékos akar-e irányítani egy botot. Ha igaz az érték, akkor nem hajtódik végre semmilyen MI tevékenység, csak a játékos által kiadott parancsok. Ha viszont nem uralja, akkor a bot különböző rendszereket használva hoz döntéseket és hajtja őket végre. Ezek a rendszerek: célválasztás, varázslatválasztás, célpontválasztás és a memóriájának a frissítése. Ezek mind külön osztályokként vannak megvalósítva. Ha ezeket minden pillanatban frissítenénk, nem csak a számítási időt terhelnénk, hanem csalnánk is, mivel egy ember nem gondolkozik ilyen gyorsan. Ezért bizonyos szabályzókkal csak adott időközönként frissítjük őket. Viszont amit minden képkockánál újra kell frissítenünk az a mozgás. Az MI megvalósítása részben lesznek az előbb említett osztályok részletesen bemutatva.

Mozgás

A mozgásért az előző fejezetben megismert kormányozó viselkedések felelnek. Ez egy külön osztályként van megvalósítva. A mozgásnál mindig lekérjük a kiszámolt erőt, amerre haladni akarunk és a mértékét. Ez egy gyorsulásvektor, amivel a sebességet növeljük. Ha az így kapott sebesség túllépné a maximális sebességet, akkor lecsökkentjük a maximális értékre. Mivel ez egy vektor érték, ezért vesszük a normalizáltját, így az iránya nem változik, csak a mérete lesz egy, majd ezt megszorozzuk a maximális sebességgel. Az így kapott értékkel növeljük a sebességet, az FPS-től (frame per second) függően. Ha csak egyszerűen hozzáadnánk, kevés FPS-nél lassan, soknál gyorsan mozogna a bot. Emiatt a két frissítés között eltelt idővel skáláznunk kell az értéket. Amikor egy zéró vektort kapnánk eredményül a mozgási rendszertől, akkor még ha nem is növeljük semennyivel a sebességet, az nem fog csökkenni. Így folyamatosan csak mozognánk. Hogy ezt elkerüljük, mikor nincs semmilyen mozgási erő, akkor egy konstans fékezési erővel csökkentjük a sebességet. Egy idő után olyan kicsire fog csökkenni a sebesség, hogy megáll a bot.

Merre is nézzünk mozgás közben?

Mivel van az egységeknek orientációja, ezért tudjuk őket forgatni. Többféleképpen meg lehetne valósítani a forgást, hogy nem feltétlen kell arra mennünk, amerre nézünk, mint például oldalazás közben, de itt az egyszerűbb irányítás érdekében mindig arra fordul a bot, amerre mozog. Ez egy metódus meghívását eredményezi, mikor mozgunk. A metódus mindig a célpont és a bot helyzetét használva kiszámolja a forgási irányt, majd egy maximális forgási sebességgel folyamatosan odaforgatja.

A környezetre reagálás

Ebben az osztályban vannak megvalósítva olyan metódusok, amik csak a botra vonatkoznak, mint hogy tud-e lépni egy adott irányban, valamint a támadás során ha megsebzünk valakit, illetve mi sebződünk meg, akkor egy-egy paraméteres üzenetküldés történik, ezeket kezeli. Ha megsebződik a bot, akkor ellökődik egy adott irányba és fogy az életpontja. Ha a kapott sérülés akkora, hogy a bot meghal, akkor egy üzenetet küld a támadójának, hogy tudjon róla, hogy jár neki a pont. Valamint a játék által küldött üzeneteket is kezeli, mint például ha a játékos eltávolít egy botot a pályáról, akkor erről üzenetet küld. Ekkor a bot kitörli az eltávolított botot a memóriájából, és ha ő a jelenlegi célpont akkor onnan is törli. Ha a lávára lép a bot akkor szintén a játék küld neki egy üzenetet, hogy tudjon róla hogy a lávára lépett, és emiatt sebződik is. A játékos is üzenetküldéssel szól a botnak, hogy hova és mit akar lőni. Illetve amikor át akarja venni az uralmat, és amikor vissza akarja adni.

3.2.5. Játék

A játék fő osztálya, ami egyben tartja a játékot. Minden egység rendelkezik a játék adattaggal, hogy elérjék a játék többi részét. Itt vannak megadva a játék kirajzolásához szükséges modellek és textúrák. A játéktérben bármilyen objektum létrehozás vagy törlés ezen osztály metódusaival valósul meg.

A létrehozható objektumok: talaj, láva, oszlop, fal (ez gátolja meg, hogy a botok kijussanak a játéktérről), varázslatok és a bot. Unityban, hogy tudjon objektumot létrehozni a játéktérbe, neki is egy objektumnak kell lennie, ezért egy üres objektumhoz van hozzárendelve, emiatt a játék osztály is a MonoBehavior-ból származik.

A pályán lévő botokat és varázslatokat számon tartja egy-egy listában. A pálya osztályt tartalmazza, a játékot indításakor inicializálja, és generálja meg a pályát, az osztály objektumlétrehozó metódusai segítségével.

A játékos kezelése

Minden frissítési ciklusban ellenőrzi, történt-e valamilyen interakció a játékos részéről, és ha igen, akkor ezt lekezeli. A játékos alapból csak egy megfigyelő, aki belátja a pályát és a kurzor nyilak segítségével tudja mozgatni a látóterét (kamerát) és a PageUp-al tud botot hozzáadni a játékhoz, PageDown-al törölni. Ha nincs bot a pályán akkor a törlésnél nem történik semmi. Még a jobb egér gombnak van hatása a játékra, mégpedig ekkor tudja „megszállni" a botot. Ekkor már csak a bot látóterületét látja, és nem tud más fele nézni. A botot az x billentyű lenyomásával tudja elhagyni, ekkor visszatér a megfigyelői állapotba.

Amíg uralja, addig a jobb egérgombbal mozgatni tudja. Ekkor a botnak parancsot küld a játék osztály, hogy mozogjon oda, ahova a játékos kattintott, és a bot végrehajtja az utasítást. Hogy pontosan hogyan is, az majd később kiderül. A játékosnak erről nem kell tudnia, ez számára egy elvárt működés.

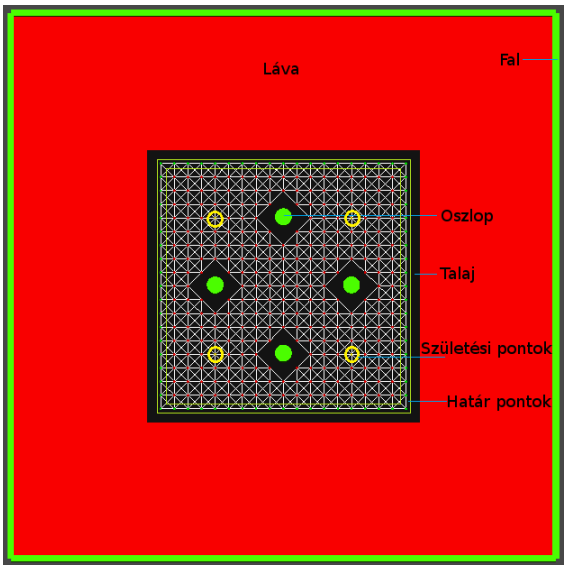
Varázsolni is tud a játékos, ekkor először ki kell választania milyen varázslatot akar használni. Ezt a Q (tűzgolyó) és W (meteor) billentyűk lenyomásával teheti meg. Ekkor a választott varázslattól függően megjelenik a választott varázslat a kurzor helyén. Ilyenkor a célzás fázisban vagyunk. Ha még egyszer megnyomnánk ugyanazt a varázslatot, visszatérnénk az alap állapotba, ha a másik varázslatot, akkor arra a varázslatra váltana a célzás. Ekkor ha megnyomjuk, a bal egérgombbal el tudjuk lőni a választott varázslatot az adott pozícióba. Ennek hatására a játék üzenetet küld a botnak, hogy mi varázsolni szeretnénk, és elküldi, hogy mit és hova. A kapott parancsot a bot végrehajtja. Ha a Shift gombot nyomva tartjuk varázslat választás közben, akkor külön célzási rész nélkül ellövi az egér pozíciójába a varázslatot.

A játék vezérlése

Amikor a bot olyan helyre akar menni, ahol útközben egy akadály van, és ki kellene kerülni, akkor egy útvonalat kell terveznie. Ekkor egy útvonal keresést indít. Az ilyen kereséseket az útvonal menedzser kezeli. A játék osztály minden frissítésnél elindít egy adott számú keresést. Hogy a keresés hogyan is zajlik, az útvonal menedzser osztály elemzésénél tudjuk meg.

Az inputkezelés és keresés elindítás után a botokat kezeli. A botokat tartalmazó listán végigmegy, és ha valaki láván áll, akkor üzenetet küld neki, hogy sebződjön. Ha egy bot születési állapotban van, akkor kiválaszt neki egy születési pontot, és újraéleszti azon a helyen. Ha egy bot halott állapotban van, akkor a születik állapotra állítja.

Ha a játékos el akar távolítani egy botot, akkor egy logikai értéket állít be. Ekkor, miután végzett a botok állapotainak frissítésével, megnézi, hogy a játékos beállította-e



3.2. ábra. A pálya felülnézetből a navigációs gráffal kiegészítve

az értéket. Ha igen, akkor szól a botoknak, hogy tudjanak a törlésről és meghívják a megfelelő metódusaikat, majd törli a játékból és a listáról.

A játék osztály tartalmaz olyan metódusokat, amik számításigényesek. A két pont között van-e akadály metódus két pontot vár a pályáról, és egy logikai értékkel tér vissza. A számoláshoz a pálya osztályban lévő akadályok listát használja, és egy kör-vonal metszést számol, mivel az akadályok kör alakúak. A másik ilyen metódus megnézi, hogy az egyik bot látja-e a másikat.

3.2.6. Pálya

Ez a pályáért felelős osztály. A pálya mérete paraméterezhető, és létrehozáskor generálódik ki: először lerakjuk a talajt, majd a lávát köré, az egész köré falakat rakunk. Véletlenszerűen lerakunk akadályokat a pályára. Ezek mind a játék osztály objektum létrehozó metódusait hívják meg. A létrejött objektumok egy-egy listában az osztályon belül el is tárolódnak. Az akadály listát használja a játék osztály ott említett metódusa. A születési pontokat a pálya négy sarkába rakja, és ezeket is egy listába tárolja, amit a játék használ az újraélesztésnél. A harmadik lista a határpontok listája, ezek a talaj szélét határoló pontok. Ezt a bot használja, mikor lávára lép, és onnan gyorsan ki akar jönni.

Ezután a járható résznek az útvonal gráfja készül el, amit a bot az útvonalkeresésnél használ. Ehhez egy gráf adatszerkezetet készítettem, amiben pontok és azokat összekötő irányítatlan élek vannak. A pontok rácsszerűen vannak összekötve. A pontban el vannak tárolva annak szomszédos pontjai, mivel a keresés ezeket használja, és nem kell külön fölösleges számításokat végeznie.

Egy metódusa van ami paraméterül egy pontot vár, és megmondja, hogy az belelóg-e egy akadályba.

3.2.7. Varázslat

A varázslatok ősosztálya. Ebből származik a tűzgolyó és a meteor is. A játékon belül meg van különböztetve a varázslat, és az abból készült lövedék. A varázslat az itt leírt osztály, ami minden botnak van. A lövedék a varázslat alapján készült objektum a játéktérben. Viszont általában mindkét helyen varázslatként hivatkozok rá a szövegben.

A varázslat osztályban van a varázslat típusa megadva, ami az eddig megismert tűzgolyó és meteor lehet. Ha egy varázslatot ellövünk, és a végtelenségig mehetne, az nem nézne ki túl jól, ezért egy maximálisan megtehető távolság van megadva. Hogy ne legyen minden pillanatban használható egy varázslat, ezért egy varázslat újbóli használata előtt, egy minimálisan eltelendő idő van definiálva. Ezt le lehet kérdezni egy logikai változóval visszatérő metódussal, hogy használható-e a varázslat, hogy fölöslegesen ne próbáljuk ellőni.

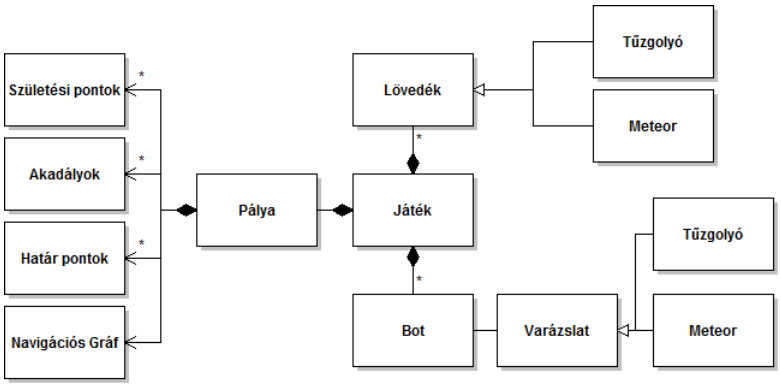
A varázslathoz tartozik még: a sebzés, hogy mennyi életpontot vonjon le a bottól, az erő, hogy milyen erővel lökje el a botot, és végül a sebesség, hogy maximálisan milyen gyorsan haladhat. Valamint tárolnunk kell a varázslat tulajdonosát is, hogyha megsebezne valakit a varázslat, tudjuk ki volt az, aki ellőtte.

Egy absztrakt metódusként definiáljuk a varázslat ellövését, mivel ez varázslatonként különbözhet. Mikor ellövünk egy varázslatot, akkor a játék osztálynak kell szólni, hogy hozzon létre objektumokat a játéktérbe. A tűzgolyót a botok maguk elől lövik el, míg a meteor az égből zuhan le. A legfőbb különbség a két varázslat között, hogy a tűzgolyó csak egy botot tud megsebezni és ellökni abba az irányba amerre haladt, ehhez képest a meteor a földbe csapódáskor egy adott körben lévő botokat mind sebzi, és a becsapódás középpontjától kifele löki el őket. A tűzgolyó ezért egy egycélpontos varázslat, míg a meteor egy területi varázslat.

3.2.8. Lövedék

A varázslatok ellövésekor keletkeznek az úgynevezett lövedékek, amik a játéktérbe bekerülnek, és mozognak a megadott irányba. Mivel ezek a játéktérben vannak, ezért, hogy egy objektumhoz hozzá lehessen rendelni, ez a MonoBehavior osztályból származik.

A játék osztály hozza őket létre, amikor a varázslatot ellövik. A megfelelő varázslatból lekéri a paramétereket, és beállítja inicializálásnál a sebzést, sebességet, erőt és, hogy ki a lövedék tulajdonosa. A lövésnél megadott célpont állítódik be célpontnak. A varázslattól függően beállítjuk a kezdő helyzetet. A megtett távolságot számon kell tartani, ezért egy



3.3. ábra. A fontosabb osztályok

távolság változó van definiálva.

A varázslat nem törölheti magát, ha becsapódik, mert lehet, hogy még hivatkozások vannak rá, ezért egy logikai változóban elmentjük az állapotát, hogy él, vagy már elpusztult. Mikor becsapódik, varázslattól függően üzenetet küldünk az érintett botnak, illetve botoknak, majd egy üzenetet a játék osztálynak is, hogy törölje a lövedéket.

Minden frissítés során a maximális sebességgel a célpont irányába mozgunk. Ezt a mozgást is a két képkocka között eltelt idővel kell skálázni, különben lassabb vagy gyorsabb lenne, mit ahogy a bot mozgásánál is említettem. Ha elérnénk a célpontot, de még nem a maximális távolságnál tartunk, akkor tovább fog menni. Ha eléri a maximális távolságot és még mindig nem ütközött semminek, akkor szól a játék osztálynak, hogy törölje, de ekkor nem sebez senkit. Ha egy akadálynak ütközne, akkor szintén elpusztul, de nem sebez senkit.

3.3. Az MI megvalósítása

3.3.1. Tervezési szempontok

Az eddigiekben ismertetett felépítés elég egy teljesen működő játékhoz, csak a lövés és mozgás parancs végrehajtást kellene valamilyen egyszerű módon implementálni. Viszont ha MI-vel szeretnénk bővíteni a játékot, akkor előtte érdemes végiggondolni, hogy milyen viselkedések kellenek, hogy megfelelően reagáljanak a botok a környezetre, majd ezeket az ötleteket valamilyen módon implementálni.

Két alapvető dolog jön szóba egyből, ami az ilyen típusú játékoknál szükséges: egy adott pontba mozgás és egy kiválasztott varázslattal célzás és lövés. Ami viszont már nem ilyen egyértelmű, hogy most támad vagy védekezik a játékos, mikor lő. Lehet, hogy épp elfutva lő, vagy oldalazva kerülget és támad. Milyen fajta mozgással kapcsolatos képességet kell tudnia az MI-nek? Egyértelmű, hogy bármely irányba képesnek kell lennie

mozogni, miközben elkerüli az akadályokat. Valamint valamilyen kereső algoritmus is kelleni fog az útvonaltervezéshez.

És a varázslatok? Milyen varázslatot válasszon? Először is ki kell választania, melyik lenne a legjobb az adott szituációban. Itt minden varázslatnak van előnye és hátránya, jobb lehet egy adott helyzetben, pl. ha több bot is van egy helyen, egy meteor célszerűbb, mert több ellenfelet is egyidőben támad, és nem kell a köztes akadályokkal foglalkozni, viszont, ha csak egy ellenfél van, akkor a tűzgolyó a jobb ötlet. A két varázslat között minimálisan eltelendő idő varázslatonként változik. Ezért ha rosszul döntünk, elég sokat kell majd várni, mire újra használható lesz. Itt előre megjegyezném, hogy mivel összesen két varázslat van, ezért nincs komolyabb varázslat-választási algoritmus. Viszont a további tervekben részben meg van említve egy megvalósítási lehetőség.

Gyakran előfordul, hogy egyszerre több ellenfelet lát, ekkor valahogy el kell döntenie, melyik legyen a célpontja. Ha egy ellenfél eltűnik egy oszlop mögött és nem látjuk, attól még tudjuk, hogy ott van, ezt is figyelembe kell venni tervezéskor. Persze nem elég véletlenszerűen rohangálni és lövöldözni egy célpontra, rengeteg dolgot figyelembe kell venni és reagálni rá, mint például, ha kevés az élet, akkor érdemes inkább elfutni.

Az előző fejezetben megismert technikákból valósítottam meg, illetve azokat ki is bővítettem. Az elsőként megismert témakör a mozgás volt. Majd ezt követte az útvonalkeresés, döntéshozás. Mivel itt csak egyéni MI-k vannak megvalósítva, és ezek nincsenek csapatban, így a csapat stratégia nem lesz megvalósítva, viszont a további tervekben meg van említve, hogyan is lehetne ezt implementálni. Amivel kibővül még a program: az a Memória, Célpont- és Varázslat választás.

3.3.2. Mozgás

A bot mozgásáért a már említett kormányozó viselkedések felelnek. Az előző részben már volt róluk szó. Két viselkedést valósítottam meg: megközelít és érkezik.

Ezekhez kell egy saját pozíció és egy cél, ahová mozogni akarunk, ezért egy célpontot változóban tárolom a mozgási cél helyzetét. Hogy most a megközelítést, vagy az érkezést használja, azt a bot döntéshozási rendszere állítja be, és a célpontot is ez választja. Mikor a játékos adja ki a mozgási parancsot, akkor is a döntéshozási rendszer kezeli ezt.

Logikai változókkal lehet beállítani, hogy megközelítünk egy pontot, vagy megérkezünk egybe. Ha a pont közel van, akkor a megérkezés lesz választva, hogy ne menjen túl rajta. Ha egy útvonalat követ, akkor csak a legutolsó ponthoz fogja a megérkezést használni, a többi ponthoz a megközelítést.

A megközelítésnél a két pontból egy vektorral tér vissza, ami az irányt és a mozgási erő nagyságát tartalmazza. A két pontból meghatározom a mozgási irányt, majd ennek méretévé a maximális mozgási sebességet teszem. Az így kapott sebességből kivonom a

jelenlegit, és ez az a kormányozó erő amit visszaad a metódus.

A megérkezés nagyon hasonlóan történik, annyi különbséggel, hogy a célponttól való távolsággal skálázom a sebességet. Így ha elég közel van a bot a célponthoz, egy zéró vektor lesz a visszatérési érték.

3.3.3. Útvonaltervezés

A botoknak szükségük van útvonal tervezésre, különben minden akadályban elakadnának. A döntéshozó rendszer, amikor parancsot kap, hogy mozogjon a pálya egyik részéről a másikba, megnézi hogy a két pont között van-e akadály. Ha nincs akkor egyszerű a dolga, csak kiadja a mozgás parancsot és a bot gond nélkül a célhoz tud menni. Ha viszont van akadály, akkor egy útvonalat kell tervezni.

Minden botnak van egy útvonaltervező osztálya. Ha egy kérés érkezik, hogy egy útvonal kell, elindít egy keresést, és ha az kész van, szól a döntéshozó rendszernek, hogy használhatja azt. A játék osztályban már volt szó ezekről a keresésekről, és ezeket az útvonalmenedzser kezeli. Most kiderül, hogy hogyan is működik ez.

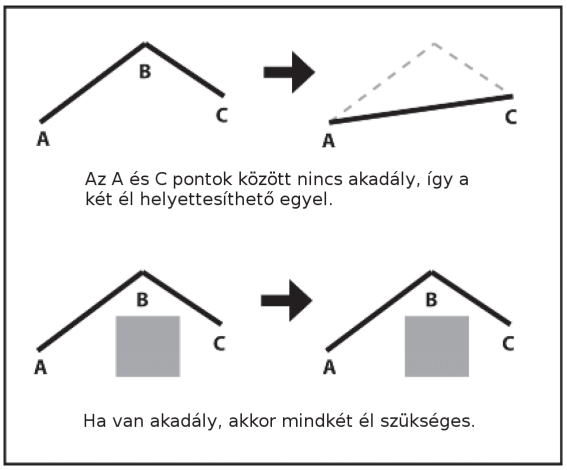
Az útvonal menedzser tartja számon a kereséseket. Ha egy keresést szeretnénk indítani akkor ide kell beregisztrálni. Ekkor egy listába bekerül az útvonal tervező. Ha úgy alakulna, hogy mielőtt a keresés véget érne, már nem lenne szükségünk az útvonalra, akkor le lehet regisztrálni a keresést, ekkor törlődik a keresési listáról.

Amikor a játék osztály elindítja a kereséseket, akkor a listán végigmegyünk, és elindítunk egy keresést. A listában az útvonal tervezők vannak, tehát a keresés ott van megvalósítva. A keresésnél a pálya navigációs gráfját használja.

Az ilyenfajta keresésnél az A* algoritmus a leghatékonyabb. Ennek egy speciális változatát implementáltam. Egy idő-szeletelt algoritmusként van megvalósítva. Ennek a lényege, hogy mikor egy keresés elindul, akkor csak egy lépést tesz meg a keresésben, egy pontot bont ki. Erre azért van szükség, mert nem feltétlen többszálú a számolási kapacitásunk, illetve ez elég költséges számolás, és minél inkább minimalizálnunk kell ezen költségeket. Ezért mikor a menedzser végigmegy a listán, a keresés mindig csak egy ciklust, egy lépést hajt végre.

Képzeljük el, hogy minden egyes keresési kérést egy külön szálon elindítva kezelnénk, ekkor ha a játékos elég gyorsan kattintgat, hogy mozogjon a bot, rengeteg keresést indítanánk el. Így viszont mindig csak felülíródik az eddigi keresés. Egy ilyen kis játékhoz, meglehet, hogy fölösleges, de ez is egyfajta technika, hogy csökkentsük a CPU időt, és be szerettem volna ezt mutatni.

Mivel mindig csak egy ciklust lépünk a keresésnél, ezért egy adott keresésszámig megyünk végig a listán újra és újra. Ha a keresés sikeresen vagy sikertelenül tér vissza egy ciklus után, kivesszük a listából és értesítjük az eredményről a döntéshozó rendszert.



3.4. ábra. Az élsimítás két esete

Ha „számol” eredményt kapunk, akkor még nem ért végig, ezért marad a listában. Ezután továbblép a következő keresésre. Ha eléri a maximális keresések számát, vagy nincs több keresés a listában, akkor véget ér a ciklus.

Élsimítás

A kapott keresési útvonalon általában lehet simítani. Az ábrán látható, hogy a felesleges pontokat ki lehet venni, ekkor az út simább lesz. Ez elég időigényes lehet ha nagyon precízen akarjuk megoldani, de meg lehet valósítani egy viszonylag egyszerű módon, bár ez nem lesz tökéletes.

Elindulunk a végponttól, és egy pontot kihagyva, megnézzük, hogy a két pont között van-e akadály, ha nincs akkor a pontot töröljük és megyünk a következőre. Ha van akkor onnantól kell nézni a pontokat. Ezt egész addig műveljük, amíg a kezdőponthoz nem érünk. Ezzel az algoritmussal lehet, hogy egy harminc pontos útvonalból három pontosat készítünk.

3.3.4. Memória

A bot hozzá tud férni az összes bot adatához, hol van, mennyi az élete, tud-e varázsolni, bárhol is van a pályán, valamint a pályán lévő varázslatokhoz is. Ez csalásnak számít, hiszen a játékos se látja be az egész pályát, illetve nem tud észben tartani ennyi dolgot. Ezért az intelligencia látszatának fenntartása érdekében minden botnak van memóriája. Mint ahogy az ember se jegyez meg mindent örökre, így ez is csak egy ideig tárol információkat, utána „elfelejti” azokat.

Hogy ez hogyan is valósul meg, arról a szenzormemória osztályt kell megnéznünk. Minden bot rendelkezik ezzel. A memóriába minden észlelt bothoz tartozik egy memória

rekord. Egy memória rekord a botról tartalmaz információkat: mikor érzékelte utoljára, mikor vált számára láthatóvá, mikor látta utoljára, hol látta utoljára, látótávolságon belül van-e, támadható-e. Ezen szempontok alapján hoz a bot döntéseket a célzásnál is.

Amikor a játékos töröl egy botot a játékból, a játék osztály üzenetet küld a botnak, hogy törölje azt a memóriájából. Ekkor a memória kitörli a bothoz tartozó memória rekordot. Ha egy új bot kerül be a játékba, és most először érzékeli a bot, akkor egy új memória rekordot hoz létre neki.

Csak adott időközönként frissíti a memóriáját a bot, hogy valósághűbb legyen. Egy ilyen frissítéskor a játék osztály botlistáján végig megy, és minden botról frissíti az információit. Ha saját magához ér, akkor azt kihagyja. Először megnézi, hogy van-e köztük akadály, és az eredménytől függően beállítja, hogy támadható-e. Majd megnézi, hogy látótávolságon belül van-e. Ha nincs, akkor hamis értéket állít be és nem is vizsgálódik tovább. Ha viszont azon belül van, akkor frissíti a a többi információt: a mikor érzékelte utoljára értéke a jelenlegi idő lesz, a hol érzékelte utoljára a bot jelenlegi helyzete, a mikor látta utoljára szintén a jelenlegi idő. Attól függően hogy az előző érzékelésnél a bot a látótávolságon belül volt-e, alakul a mikor vált láthatóvá ideje, ha eddig is belül volt, akkor nem változik, ellenkező esetben a jelenlegi idő lesz az értéke, és a látótávolságon belül van-e értéke is igazra vált.

A memória nem csak a botokat tartja számon, hanem a varázslatokat is. A botokhoz hasonlóan frissíti az információkat. Minden a memóriában lévő adatot le tud kérdezni a bot. A memória a döntéshozás és célpontválasztásban játszik nagy szerepet.

3.3.5. Célpont választás

A bot minden frissítésnél megpróbál magának egy célpontot választani, a célpont mindig egy másik bot. Ha nincs bot a pályán, vagy üres a memóriája (ez akkor lehet, ha létrehozás óta még nem találkozott egy bottal sem), akkor nem tud választani.

A memóriából lekéri az észlelt botok listáját, ebben csak olyan botok tartoznak bele, amiket egy adott időn belül észlelt, ami ezen kívül esik, azt már „elfelejtette”. A célpont választás a megmaradt botok közül a legközelebbi botot választja. A további tervekben leírtam, hogy tervezem a funkció változtatását.

Nem csak egy botot választ célpontnak, hanem a varázslatok közül is választhat egyet. Ilyenkor ezt a varázslatot veszélyesnek ítéli, és úgy gondolja, erre valahogyan reagálni kell, különben lehet el fogja találni a botot. Csak egy adott közelségben lévő, a memóriájából lekért, őt támadó varázslatokat vizsgálja, és ebből a lehető legközelebbit választja. A választott célpontot és veszélyes varázslatot a döntéshozási rendszer használja.

3.3.6. Varázslatválasztás

A varázslás rendszer felel a megfelelő varázslat kiválasztásáért, és ennek a varázslatnak az ellövéséért is. A rendszer inicializáláskor állítja be a bot pontosságát, reakcióidejét és adja hozzá a használható varázslatokat a bot választható varázslatai közé.

A varázslat választásnál csak kettő közül választhat: tűzgolyó és meteor. Alapból a tűzgolyó van kiválasztva. A választás nagyon egyszerű: amelyik varázslatot lehet használni, azt választja. A varázslat használata után el kell, hogy teljen egy minimális időnek, mire újra tudja azt használni. Ha mindkettő lehet, akkor amelyiket előbb lehet használni. A további tervekben leírtam, hogyan tervezem a funkció bővítését.

A választott varázslatnak nem kell lövésre készen állnia, csak akkor, mikor a lövés parancs ki lesz adva, a rendszer nem fogja ellőni a varázslatot. A játékos is ezt a rendszert használja, amikor a varázslatokat használja. Az ellövési metódusok is itt vannak megvalósítva.

Amikor valakire lőni akarunk, és távol van, akkor annak elég nagy a valószínűsége, hogy elmozdul arról a pozícióról, mire a lövedék odaér. Ezért a célpont jövőbeli helyzetét érdemes megjövendölni. Ezt a botok meg is teszik a lövéskor. Az algoritmus a következő: megnézzük milyen messze van tőlünk az ellenfél, majd ezt elosztjuk a varázslat és a célpont maximális sebességének összegével. Az így kapott eredménnyel megszorozzuk a célpont sebességét, és ezzel az értékekkel eltoljuk a pozícióját. Ekkor a kapott pozíciót használjuk a lövésnél használt pozíciónak.

Fontos, hogy azon felül, hogy a varázslatot használhatjuk, el kell telnie egy bizonyos időnek, a reakció időnek, különben túl jók lennének a botok. Valamint a célzott pozícióhoz valamennyi zajt is kell adni, a lövési pontosságtól függően, különben mindig célba találnának. Ha igazán nehéz ellenfelet szeretnénk készíteni, akkor ezeket az értékeket minimalizálni kellene.

3.3.7. Döntéshozás

Az előző részekben már sokszor meg lett említve a döntéshozás, most végre kiderül, hogyan is néz ez ki. A botnak egy célorientált döntéshozási rendszert készítettem. Először nézzük meg, hogy mi is ez, majd azt követően, hogy hogyan is lehet ezt megvalósítani.

Alapvetően kétfajta cél létezik: egyszerű és összetett. Egyszerű cél lehet: fordulj meg, vagy ugorj fel, összetett, ha azt mondjuk valakinek: menj el a boltba, ekkor tudni kell, hogy milyen útvonalon kell mennünk, azon végig is kell haladni. Mindkét típusú célnak lehet ellenőrizni az állapotát, hogy sikerült-e vagy nem, és ha nem, akkor újratervezni.

A hierarchikus célrendszer nagyon hasonló az emberi gondolkodáshoz, mert ott is egy magas szintű absztrakt célt választunk, és azt lebontjuk kisebb részfeladatokra egészen

addig, míg elég egyszerű feladatokat kapunk. A botok is így fognak gondolkodni: ha a pálya egyik végéből a másikba akar eljutni, akkor előbb az utat meg kell terveznie, majd azon végig is kell haladnia.

Fontos megjegyezni, hogy egy célt nem csak egyféleképpen lehet megvalósítani, mint ahogy az életben sem. Ha el akarunk jutni például a moziba, akkor ezt megtehetjük kocsival, busszal, gyalog, és hogy melyiket választjuk, ez nagyon sok mindentől függhet.

A botoknak a játékban kell egy fő cél, általában, hogy megnyerjék a játékot. Itt az a fő cél, hogy a másik botokat minél többször megölje, és a lehető legtovább életben maradjon. És hogy ezt hogyan valósítják meg, az nemsokára kiderül.

Cél felépítés

A célrendszer készítéséhez Composite tervezési mintát használtam. A döntéshozás alapeleme a cél, ezért egy absztrakt cél osztályt készítetem, amiből az egyszerű és összetett cél is öröklődik. Itt vannak definiálva a cél típusok és a cél állapotok. A cél tulajdonosa is tárolva van. A céltípusokról a későbbiekben lesz szó.

A cél állapot négyféle lehet: aktív, inaktív, elkészült és sikertelen. Az inaktív cél arra vár, hogy aktiválódjon, az aktív cél már aktiválva van és végrehajtódik minden frissítésnél, az elkészült cél elkészült, és a következő frissítéskor törlődik, és végül a sikertelen cél valamiért nem sikerült, és vagy újraterveződik vagy törlődik a következő frissítéskor.

Minden célnak három főbb metódusa van: Aktiválás, Feldolgozás, Megszüntetés. Nagyon hasonló, mint a folyamatoknál a belépés, végrehajtás és a kilépés. Az Aktiválás a cél inicializálásakor és újratervezésekor hajtódik végre. A Feldolgozást hajtja végre minden frissítési lépésben és a cél állapotát adja visszatérési értékként. A Megszüntetés takarít fel, mielőtt a cél kilépne, a cél megszüntetésekor hívódik meg.

Az egyszerű célok a cél osztályból származnak, és csak az absztrakt metódusokat definiálják, semmi többet. Az összetett célok viszont az összetett cél osztályból származnak. Az összetett cél osztálya a cél osztályt bővíti a következő metódusokkal: alcél hozzáadása, alcélok végrehajtása és alcélok törlése. Az alcélokat egy listában tárolja, az alcél hozzáadásakor ezt a listát bővíti. Az alcélok törlése metódus végigmegy az alcél listán, és meghívja a cél Megszüntetés metódusát. Az alcélok végrehajtása metódus az elején az elkészült és sikertelen célokat eltávolítja a listáról. Ha ezután az alcél lista üres lenne, akkor az elkészült állapottal tér vissza. Ha nem üres a lista, akkor a lista végén lévő célnak meghívja a Feldolgozás metódusát. Ilyenkor egy speciális esetet kell megvizsgálni, mikor a cél elkészült állapottal tér vissza, de még van cél az alcéllistában, ekkor a metódusnak aktív állapottal kell visszatérnie, hogy a többi cél is fel legyen dolgozva. Egyébként pedig a cél visszatérési értékét adjuk vissza.

Célok

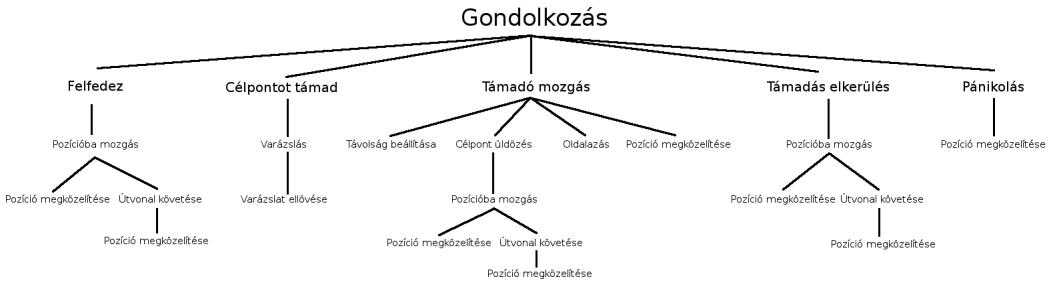
A célrendszert most már ismerjük, jöhetnek a célok. 14 célt készítettem: Gondolkozik, Felfedez, Célpontot támad, Támadó mozgás, Varázslás, Támadás elkerülés, Célpont üldözés, Pozícióba mozgás, Pánikolás, Útvonal követése célok összetettek és a Távolság beállítása, Oldalazás,Varázslat ellövése, Pozíció megközelítése célok egyszerűek. Mivel az ilyenfajta játékokban az ölések száma a fontos, a haláloké nem, ezért nincsen semmilyen fedezékbefutás vagy menekülés cél. A további tervek részben kifejtem, hogy ezt hogyan is lehetne bővíteni.

Elbírálás. A legfelsőbb szintű cél a Gondolkozás. Minden botnak van ebből az osztályból egy példánya, és addig tart, amíg a bot ki nem törlődik. Ez a cél választ ki egyet a stratégiai célok közül, ami legjobban illik a játék jelenlegi állapotához. A stratégia célok: Felfedez, ez választ egy pontot a pályán és odamegy, Támadó mozgás, a lehető legjobb pozícióba mozog a célpontjához képest, Célpont támadás, a jelenlegi célpontra varázsol, Támadás elkerülés, a veszélyesnek ítélt varázslatot megpróbálja elkerülni, és a Pánikolás, mikor a lávára kerül, és a lehető leggyorsabban megpróbál kijutni onnan.

Ezeket a játék jelenlegi állása alapján elbírálja. A bírálat egy szám érték, és a legnagyobb számértékkel rendelkező célt választja. Ha a jelenlegi célja megegyezik a választott céllal, akkor nem tesz semmit, viszont ha nem akkor a jelenlegi célt megszünteti és törli, helyére a választott cél kerül és ezt hajtja végre.

Egy cél elbírálása minden botnál egyforma értéket adna, így a Gondolkozás cél létrehozásakor az elbírálási osztályokhoz egy véletlenszerű számot definiálunk, amivel a bírálati értéket megszorozzuk, hogy minden botnak egyedi legyen célválasztása. Ezzel úgymond egy személyiséget adunk nekik. Így valamelyik jobban szeret támadni, míg a másik inkább védekezőbben játszik.

A Felfedezés cél elbírálásakor egy konstans alacsony értéket adunk vissza, hogyha nincs más ötlete, akkor ezt a célt válassza. A Támadó mozgás csak akkor jöhet szóba, ha van célpontunk, különben nulla lesz az értéke. Ha van, akkor a bot életpontja és a varázslatok erősségének a szorzata. Ha kevés az életpont, akkor kisebb a valószínűsége, hogy ezt válassza. A varázslat erőssége a varázslatainak használhatóságától függ, ha nagyon sokat kell arra várni, hogy használhassa őket, akkor kicsi az érték és egyre növekedik, amíg használhatóvá válnak. A Célpont támadás csak akkor értékelődik ki, ha van célpontunk, az a látótávolságon belül van, és lőhető, tehát nincs akadály a bot és közte. Ekkor attól függően kapunk egy értéket, hogy a varázslat mikor lesz használható, ha nagyon sokat kell rá várni, akkor kicsi az érték, ha már használható, akkor magas az érték. A Támadás elkerülés csak akkor értékelődik ki, ha van veszélyesnek ítélt varázslat. Ekkor egy konstans közepesen nagy értékkel tér vissza. A Pánikolás cél akkor értékelődik ki, ha



3.5. ábra. A célhierarchia

a bot lávára lép, ekkor egy magas konstans értékkel tér vissza.

Most, hogy már tudjuk, hogyan értékelődnek ki a stratégiai célok, ideje megtudni, hogyan is lehet ezeket implementálni. Minden cél létrehozáskor inaktív állapotba van, és a Feldolgozás metódusban mindig először megnézzük, hogy a cél inaktív állapotban van-e, és ha igen akkor aktiváljuk. Ekkor inicializálódik a cél.

Gondolkozás. Aktiválás: ha a játékos nem uralja a botot, akkor elbírálásra kerülnek a célok. Majd ezt követően az állapot aktívra vált.

Feldolgozás: meghívódik a alcélok feldolgozása metódus, majd ennek a visszatérési értékét vizsgáljuk. Ha ez az elkészült állapotban van vagy sikertelen állapotban és a játékos nem uralja, akkor az állapot inaktívra vált, majd visszatérünk az állapottal. Ennek a következménye, hogy következő Feldolgozáskor, mivel inaktív, ezért újra aktiválódik, és ekkor újra kiértékelődnek a célok, és egy új alcél lesz választva.

Megszüntetés: üres, mivel ez a fő cél, és csak akkor szűnik meg, amikor a botot töröljük a játékból.

Felfedez. Aktiválás: az állapotot aktívra állítjuk, töröljük az összes alcélt, és ha nincsen cél beállítva, akkor választunk egy pontot a pálya osztály navigációs gráfjából véletlenszerűen, és a Pozícióba mozgás célt választjuk alcélul.

Feldolgoz: ha inaktív az állapota, akkor aktiváljuk(meghívjuk az Aktivál metódust). Az alcélok feldolgozása metódus hívódik meg, és ennek az eredményét adjuk visszatérési értékként vissza.

Megszüntetés: az alcélokat töröljük, és a cél állapotát elkészültre állítjuk.

Célpontot támad. Aktiválás: a cél állapotát aktívra állítjuk és töröljük az alcéljait, majd ha nincs a célpont kiválasztva a célzási rendszerben, akkor a cél állapotát elkészültre állítjuk és nem megyünk tovább. Ez az eset csak akkor következhet be, ha a cél kiértékelésénél még volt, de a cél végrehajtásakor már nincs célunk. Nagy eséllyel a bot meghalt vagy törlődött és még nem frissült a célzó rendszer. Ha van célpont, akkor tovább vizsgáló-

dunk, ha a célpont lőhető és látótávolságon belül van, akkor a Varázsol cél választjuk alcélnak. Ellenkező esetben a cél állapotát elkészültre állítjuk.

Feldolgozás: ha inaktív, akkor aktiváljuk, az alcélokat feldolgozzuk, majd ennek eredményét adjuk vissza.

Megszüntetés: az alcélokat töröljük és az állapotot elkészültre állítjuk.

Támadó mozgás. Aktivál: az állapotot aktívra állítjuk, töröljük az alcélokat. Ha nincs célpontunk, akkor az állapotot elkészültre állítjuk és nem megyünk tovább. Ha van, akkor megnézzük, hogy lőhető-e, ha nem, akkor a Célpont üldözés célt választjuk alcélul. Ha lőhető, akkor ha nincs látótávolságban, akkor a Távolság beállítása célt választjuk alcélnak. Ha látótávolságon belül van, akkor megnézzük, hogy tudunk-e oldalazni, ha van elég hely hozzá, akkor az Oldalaz célt választjuk, egyébként a Pozíció megközelítése célt.

Feldolgozás: ha inaktív az állapot, akkor aktiváljuk, az alcélt feldolgozzuk és ha a kapott állapot sikertelen, akkor inaktívra állítjuk, ekkor újraterveződik a következő feldolgozáskor. Ha a botot meglőtték, akkor szintén inaktívra állítjuk az állapotot. Végül az állapotot visszaadjuk.

Megszüntetés: töröljük az alcélokat és az állapotot elkészültre állítjuk.

Támadás elkerülés. Aktiválás: beállítjuk a cél állapotát aktívra. Ezután megnézzük, hogy van-e veszélyes varázslat. Ha nincs, akkor az állapotot elkészültre állítjuk és nem folytatjuk tovább. Ha van, akkor annak típusától függően kiszámoljuk, hogy hova kellene mozogni, hogy elkerüljük, majd a Pozícióba mozgást adjuk alcélul.

Feldolgozás: ha inaktív, akkor aktiváljuk, az alcélokat feldolgozzuk. Ha eltűnne a veszélyesnek ítélt varázslat a célzórendszerből, akkor az állapotot elkészültre állítjuk és végül visszatérünk az állapottal.

Megszüntetés: töröljük az alcélokat és az állapotot elkészültre állítjuk.

Pánikolás. Aktiválás: aktívra állítjuk az állapotot és töröljük az alcélokat. Megkeressük a pálya osztályban lévő határpontok közül a legközelebbit és a Pozíció megközelítése alcélt választjuk és a választott pontot paraméterül adjuk.

Feldolgozás: ha az állapot inaktív, akkor aktiváljuk, az alcélokat feldolgozzuk és megnézzük, hogy még mindig láván állunk-e. Ha nem, akkor elkészültre állítjuk az állapotot. Ha még mindig láván állunk, akkor attól függően, hogy eltelt-e egy adott idő, inaktívra állítjuk az állapotot (a legközelebbi frissítéskor az Aktiválás újra fog futni), mert feltehetően mikor először állítottuk be a célpontot, még mozogtunk. Vagy az is lehet, hogy azóta megint eltaláltak és azért lökődtünk el, és mindig a legrövidebb úton akarunk kimenni, mert elég rosszul nézne az ki, hogy a pálya másik végébe megyünk, hogy kikerüljünk a lávából, mikor csak egy picit előrébb kellene mennünk. Ezután visszaadjuk az állapotot.

Megszüntetés: töröljük az alcélok listáját és az állapotot elkészültre állítjuk.

Pozícióba mozgás. Aktiválás: beállítjuk a cél állapotát aktívra és töröljük az alcélokat. Küldünk egy kérést az útvonal tervezőnek a jelenlegi célunkkal, majd egy logikai értéket beállít, hogy várjunk-e útvonalra ami hamis, ha nincs akadály az úton és igaz, ha van akadály. A Pozíció megközelítése célt választunk alcélnak és átadjuk a célpontot paraméterül.

Feldolgozás: ha inaktív, akkor aktiváljuk, feldolgozzuk az alcélokat és a kapott eredményt beállítjuk az állapotunknak. Ha az állapot sikertelenre vált, akkor újra aktiváljuk. Majd visszatérünk az állapottal.

Megszüntetés: ha várunk útvonalat, akkor szólunk az útvonal menedzsernek, hogy a keresésünket törölje. Töröljük az alcél listát és az állapotot elkészültre állítjuk.

Varázslás. Aktiválás: beállítjuk a cél állapotát aktívra és töröljük az alcélokat. A varázslat elsütése célt adjuk alcélul és a kapott célpontot adjuk paraméterül.

Feldolgozás: ha inaktív, akkor aktiváljuk, feldolgozzuk az alcélokat és a kapott eredményt beállítjuk az állapotunknak, majd visszatérünk ezzel.

Megszüntetés: töröljük az alcél listát és az állapotot elkészültre állítjuk.

Célpont üldözés. Aktiválás: beállítjuk a cél állapotát aktívra és töröljük az alcélokat. Ha nincsen célpont kiválasztva, akkor az állapotot elkészültre állítjuk és nem teszünk mást. Egyébként megnézzük, hogy a célpont lőhető-e.Ha igen, akkor az állapotot elkészültre állítjuk és nem teszünk mást. Ha nem lőhető, akkor a legutolsó észlelt pozíciójába akarunk eljutni. Előtte meg kell vizsgálni, hogy van-e ilyen pozíció, vagy, hogy ott vagyunk-e már. Ha az előző feltételek közül valamelyik igaz, akkor nem bírjuk üldözni, ezért egy Felfedez célt állítunk be alcélnak. Ha nem teljesülnek, akkor a legutóbb érzékelt pontba kell jutnunk, ezért egy Pozícióba mozgás célt választunk és ezt a pozíciót adjuk paraméterül.

Feldolgozás: ha inaktív a cél állapota, akkor aktiváljuk, feldolgozzuk az alcélokat és a kapott eredményt beállítjuk az állapotunknak. Ha a célpontunk megszűnne, akkor a cél elkészültre vált. Ha van célpontunk és lőtávolságon belül van, akkor is elkészültre vált a cél. Végül visszatérünk az állapottal.

Megszüntetés: töröljük az alcél listát és az állapotot elkészültre állítjuk.

Útvonalkövetés. Aktiválás: beállítjuk a cél állapotát aktívra. A paraméterül kapott útvonal listából kivesszük a legutolsó elemet és ezt adjuk paraméterül a Pozíció megközelítése célnak, ha az utolsó pont az útvonalba, akkor jelezzük a célnak.

Feldolgozás: ha inaktív, akkor aktiváljuk, feldolgozzuk az alcélokat és a kapott eredményt beállítjuk az állapotunknak. Ha az állapot elkészült, de még van elem az útvonal listában, akkor meghívjuk az Aktiválás metódust. Végül visszatérünk az állapottal.

Megszüntetés: töröljük az alcél listát és az állapotot elkészültre állítjuk.

Pozíció megközelítés. Aktiválás: beállítjuk a cél állapotát aktívra. Megjegyezzük egy változóban, hogy mikor aktiváltuk és kiszámoljuk, hogy mennyi idő alatt kellene a pozícióba érnie a botnak. Majd ezt egy hibázási értékkel megnöveljük. A bot mozgási rendszerének célpontul adjuk a paraméterben kapott pozíciót attól függően, hogy utolsó pont-e az útvonalba, bekapcsoljuk az érkezés, vagy megközelítés viselkedést.

Feldolgozás: ha inaktív, akkor aktiváljuk. Megnézzük, hogy mennyi idő telt el az aktiválás óta és ha több mint amennyi idő alatt oda kellene érnünk, akkor elakadtunk. Ilyenkor a cél állapotát sikertelenre állítjuk. Ha időben vagyunk és a célpozícióba vagyunk, akkor a célt elkészültre állítjuk. Végül visszatérünk az állapottal.

Megszüntetés: kikapcsoljuk a mozgási rendszerben a megközelítés és megérkezés viselkedéseket és a cél állapotát elkészültre állítjuk.

Varázslat ellövése. Aktiválás: beállítjuk a cél állapotát aktívra. Megjegyezzük egy változóban, hogy mikor aktiváltuk. Ha kiválasztott varázslat még nem lőhető el. akkor a cél állapotát sikertelenre állítjuk.

Feldolgozás: ha inaktív, akkor aktiváljuk. Megnézzük, hogy mennyi idő telt el az aktiválás óta és ha több, mint amennyi a egy teljes forgáshoz kellene, akkor valami nem sikerült, ezért a cél állapotát sikertelenre állítjuk. Ha a jelenlegi varázslat ellőhető, akkor megnézzük, hogy a botot uralja-e a játékos. Ha igen, akkor a célzott helyhez fordulunk és ellőjük a választott varázslatot. Ha nem uralja, akkor a varázslás rendszer célzás és lövés metódusát használjuk. Ez megjövendöli a célpont jövőbeli helyét és odafordul, majd ellövi a varázslatot. Ha a varázslatot még nem lehet ellőni, akkor a cél sikertelenre vált. Végül visszatérünk az állapottal.

Megszüntetés: beállítjuk a cél állapotát elkészültre.

Oldalazás. Aktiválás: beállítjuk a cél állapotát aktívra. A bot mozgási rendszerében bekapcsoljuk a megközelítés viselkedést. A cél létrehozásakor beállítottunk egy logikai értéket véletlenszerűen. Ennek az értékét megnézve vagy jobbra, vagy balra próbálunk menni. Ha tudunk az adott helyre lépni, akkor a mozgási rendszernek célul adjuk azt a pozíciót, ahova akarunk lépni. Ha nem tudunk lépni, inaktívra állítjuk a cél állapotát.

Feldolgozás: ha inaktív, akkor aktiváljuk. Ha nincs célpont választva, vagy az nincs látótávolságban, akkor a célt elkészültre állítjuk. Egyébként megnézzük, hogy az adott

pozícióba értünk-e már, ha igen, akkor a cél állapotát inaktívra állítjuk. Végül visszatérünk az állapottal.

Megszüntetés: kikapcsoljuk a mozgási rendszerben a megközelítés viselkedéseket és a cél állapotát elkészültre állítjuk.

További Tervek

Egy játék sosincs kész, csak legfeljebb már nem folytatják. Itt még lenne mivel bővíteni, mind tartalmilag, mind kinézetileg. Mivel a dolgozat célja egy játék bemutatása volt, nem az, hogy hogyan néz ki, így a megjelenítés rész eléggé el volt hanyagolva. Ezen a jövőben tervezek változtatni, hogy egy elfogadható kinézete legyen a játéknak.

Nincs a játékban semmilyen menekülési, vagy fedezékbe bújási cél, ezt pótolni szeretném a stratégiai részben megismert taktikai útvonalkeresés technikával.

Mivel a célpont és varázslat választás igen egyszerű, a varázslatnál a fő ok, hogy csak kettő van, ezért elsősorban a varázslatokat bővíteném és egy fuzzy logikával oldanám meg mindkét választást. Ez egy viszonylag elég gyakran használt technika a játékfejlesztésben a döntéshozásnál.

Csak egy fajta játékmód van, a Deathmatch, ezt terveztem kibővíteni a csapatjátékkal, mikor is két vagy több csapatban lennének a botok. Ekkor már lehetne bővíteni különböző taktikákkal és stratégiákkal az MI-t. Valamint új pályákkal bővíteni a játékot.

Nyilatkozat

Alulírott Dusnoki Attila programtervező informatikus BSc szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Tanszékcsoport Számítógépes Algoritmusok és Mesterséges Intelligencia Tanszékén készítettem, programtervező informatikus BSc diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat / diplomamunkámat a Szegedi Tudományegyetem Informatikai Tanszékcsoport könyvtárában, a helyben olvasható könyvek között helyezik el.

Szeged, 2013. május 16. .....

aláírás

Köszönetnyilvánítás

Először is szeretném megköszönni a témavezetőmnek, **Dr. Csirik Jánosnak**, a támogatásáért és hasznos útmutatásáért. A tudása és tanácsai segítetek abban, hogy zavartalanul tudjak haladni. Hálával tartozom az évfolyamtársaimnak, akik hozzájárultak ahhoz, hogy a tanulás élvezhető élmény legyen. Végül köszönetet szeretnék mondani a családomnak és a közeli barátaimnak, akik lelkesedése, érdeklődése és támogatása segített abban, hogy elérjem a céljaimat.

Irodalomjegyzék

[1] Millington, I., *Artificial Intelligence for Games*, Morgan Kaufmann, San Francisco, 2005.

[2] Buckland, M., *Programming Game AI by Example*, Wordware Publishing, Plano, 2004.