A képen szöveg látható

Automatikusan generált leírás

Csúcsforgalom játékfejlesztés

Rush hour game development

Szabó Dániel

Z3IK64

Dr. Bolla Kálmán Milán, főiskolai docens

2024

Tartalomjegyzék

[1. Bevezetés 8](#_Toc181897190)

[1.1. A szakdolgozat célja 8](#_Toc181897191)

[1.2. Felhasznált algoritmusok és eszközök 8](#_Toc181897192)

[1.3. A szakdolgozat felépítése 8](#_Toc181897193)

[2. A feladat elemzése, a probléma érintő bemutatása 9](#_Toc181897194)

[2.1. A játék történelme 9](#_Toc181897195)

[2.2. A játék szabályai 9](#_Toc181897196)

[2.3. A probléma leírása 9](#_Toc181897197)

[2.4. A megoldandó problémák 9](#_Toc181897198)

[2.4.1. Generáló algoritmus 9](#_Toc181897199)

[2.4.2. Megoldó algoritmus 10](#_Toc181897200)

[2.5. Technológiák 10](#_Toc181897201)

[2.5.1. Unity 10](#_Toc181897202)

[2.5.2. C# programozási nyelv 10](#_Toc181897203)

[2.5.3. Adatbázis 10](#_Toc181897204)

[3. Szoftver tervezése 11](#_Toc181897205)

[3.1. Funkciók 11](#_Toc181897206)

[3.1.1. Játék nehézségének kiválasztása 11](#_Toc181897207)

[3.1.2. Játék ciklus 12](#_Toc181897208)

[3.1.3. Regisztráció 12](#_Toc181897209)

[3.1.4. Bejelentkezés 13](#_Toc181897210)

[3.1.5. Ranglista 14](#_Toc181897211)

[3.1.6. Beállítások 15](#_Toc181897212)

[3.1.7. Környezeti elemek 15](#_Toc181897213)

[3.2. Algoritmusok működése 16](#_Toc181897214)

[3.2.1. Generáló algoritmus 16](#_Toc181897215)

[3.2.2. Megoldó algoritmus 16](#_Toc181897216)

[3.2.3. Járművek és pálya megjelenítése 17](#_Toc181897217)

[3.3. Adatbázis 17](#_Toc181897218)

[3.3.1. Adatbázisterv 17](#_Toc181897219)

[4. Szoftver kialakítása 19](#_Toc181897220)

[4.1. Unity 19](#_Toc181897221)

[4.1.1. Történelem 19](#_Toc181897222)

[4.1.2. Hasznos funkciók 19](#_Toc181897223)

[4.2. Menürendszer 20](#_Toc181897224)

[4.2.1. Állapotok kezelése 20](#_Toc181897225)

[4.2.2. Navigáció 20](#_Toc181897226)

[4.3. Menü elemek 21](#_Toc181897227)

[4.4. Adattárolás 21](#_Toc181897228)

[4.4.1. Adatbázis létrehozása 21](#_Toc181897229)

[4.4.2. Modell osztályok 21](#_Toc181897230)

[4.4.3. Kezelő osztályok 21](#_Toc181897231)

[4.5. Feladvány létrehozása 25](#_Toc181897232)

[4.5.1. Nehézség és pályaméret kezelése 25](#_Toc181897233)

[4.5.2. Generálás folyamata 25](#_Toc181897234)

[4.5.3. Megoldhatóság ellenőrzése 27](#_Toc181897235)

[4.6. Játékmenet 30](#_Toc181897236)

[4.6.1. Jármű kijelölése 30](#_Toc181897237)

[4.6.2. Jármű mozgatása 30](#_Toc181897238)

[4.6.3. Játék vége 32](#_Toc181897239)

[4.7. Környezeti elemek 32](#_Toc181897240)

[4.7.1. Zene 32](#_Toc181897241)

[4.7.2. Közlekedő járművek 33](#_Toc181897242)

[4.7.3. Napi ciklus 35](#_Toc181897243)

[5. Tesztelés, módosítások 36](#_Toc181897244)

[5.1. Tesztelési csoportok 36](#_Toc181897245)

[5.1.1. Tesztek felépítése 36](#_Toc181897246)

[5.1.2. Algoritmusok tesztelése 37](#_Toc181897247)

[5.1.3. Megjelenítés tesztelése 37](#_Toc181897248)

[5.1.4. Manuális tesztelés 38](#_Toc181897249)

[5.2. Tesztek eredménye 38](#_Toc181897250)

[5.2.1. Algoritmusok 38](#_Toc181897251)

[5.2.2. A csillag és előretekintő keresés összehasonlítása 38](#_Toc181897252)

[5.2.3. Vizuális tesztek 40](#_Toc181897253)

[5.2.4. Teljesítményteszt 40](#_Toc181897254)

[5.3. Továbbfejlesztési lehetőségek 41](#_Toc181897255)

[5.3.1. Platformok 41](#_Toc181897256)

[5.3.2. Többjátékos mód 41](#_Toc181897257)

[5.3.3. Komplex közlekedés 42](#_Toc181897258)

[Összefoglalás 43](#_Toc181897259)

[Summary 44](#_Toc181897260)

[Ábrajegyzék 45](#_Toc181897261)

[Irodalomjegyzék 46](#_Toc181897262)

[Melléklet 48](#_Toc181897263)

1. Bevezetés
   1. A szakdolgozat célja

Szakdolgozatom célja a Rush Hour játék elkészítése érdekes és skálázható nehézségű feladványokkal. A játék lényege, hogy az autónkat kijuttassuk egy járművekkel teli parkolóból. Habár a játékot fizikai és webes formában is megvalósították már, ezek a változatok nem véletlenszerű pályákat kínálnak, hanem előre megtervezett és fix nehézségű feladványokat, ami korlátozza a hosszú távú újrajátszhatóságot.

Nyilvánvaló, hogy sok játék tartalmaz rejtvényelemeket. De egy rejtvény statikus, míg egy játék dinamikus. Egy kielégítő játékot újra és újra lehet játszani, és különböző stratégiák vezethetnek a sikerhez. A számítógépes játék egy szoftverprogram, amelyben egy vagy több játékos döntéseket hoz a játékobjektumok és erőforrások irányításával, céljaik elérése érdekében [1].

Azért döntöttem a feladat mellett, mert mindig érdekelt a játékfejlesztés és a program a nehézségei érdekesek és megfelelő kihívást jelent a szakdolgozathoz.

* 1. Felhasznált algoritmusok és eszközök

A dolgozatban egy olyan megoldás kidolgozására törekszem, amely lehetőséget ad a nehézség kiválasztására és véletlenszerű pályák generálására mesterséges intelligencia alapú algoritmusok segítségével, mint például a visszalépéses keresés és az A csillag keresés. A választásom azért esett erre a megoldásra, mert ezek az algoritmusok könnyen alkalmazhatók és optimalizálhatók az adott problémára, valamint különösen hatékonynak bizonyultak hasonló játékok feladványainak létrehozásában. Azért is részesítettem előnyben ezeket az algoritmusokat, mivel korábbi próbálkozásaim nem vezettek eredményre. Az első megközelítés során a megoldás összekeverésével érte volna el véletlenszerű pályákat, ezek viszont nem voltak skálázhatók és könnyen megoldhatók voltak.

A játék motorjának a Unity Engine-t választottam a könnyű használat miatt és a későbbi továbbfejlesztési lehetőségekért a felhasználói igényeknek megfelelően.

* 1. A szakdolgozat felépítése

A dolgozat során részletesen bemutatom a pályagenerálás folyamatát, a nehézség meghatározását, a megoldó algoritmust és a játék apróbb részleteit, mint az interakció a járművekkel, a környezet és a játékciklus.

1. A feladat elemzése, a probléma érintő bemutatása
   1. A játék történelme

Az 1970-es években Don Rubin az Egyesült Államokban és Nobuyuki „Nob” Yoshigahara Japánban függetlenül egy új típusú csúszóblokk rejtvényt találtak fel, ahol minden blokk csak vízszintesen vagy függőlegesen mozgatható. A motiváció az, hogy minden blokkot úgy képzeljünk el, mint egy autót, amely előre és hátra tud haladni, de nem tud fordulni. [2]

* 1. A játék szabályai

Az alapjáték egy 6x6-os rácson játszódik, ahol autók és kamionok helyezkednek el. A feladat, hogy a saját autónkat kijuttassuk a parkolóból, viszont más járművek ezt megakadályozzák. A járművek elhelyezkedhetnek vízszintesen és függőlegesen és csak ezen a tengelyen mozgathatók, addig ameddig egy másik jármű nem állja az útjukat.

* 1. A probléma leírása

Érdekes és skálázható feladványok generálása a játék szabályai és a megoldhatóság biztosítása miatt bonyolult.

A játék bizonyítottan PSPACE-teljes. Azok a problémák, amelyek megoldhatók egy determinisztikus Turing-gépen, polinomiális mennyiségű tárhellyel. Formálisan, egy probléma a PSPACE osztályba tartozik, ha létezik egy olyan algoritmus, amely a probléma megoldása során legfeljebb O(n^k) memóriát használ, ahol n a bemenet mérete, és k egy konstans. [3]

* 1. A megoldandó problémák
     1. Generáló algoritmus

Az generáló algoritmus visszalépéses keresést használ. Egy üres pálya legenerálásával kezdődik, ahol minden mező értéke nulla. Ezután az egyes mezőkre letehető összes lehetséges jármű információját eltárolja. Miután lehelyez egy járművet lefut egy megoldó algoritmus, ami ellenőrzi, hogy megoldható-e a feladvány, ha megoldható akkor az összes letehető járművek listájából törlésre kerülnek a megfelelő elemek a játék szabályai alapján, ha nem megoldható a feladvány, akkor törlésre kerül a jármű a pályáról és újat helyez le.

* + 1. Megoldó algoritmus

A megoldó algoritmus A csillag kereséssel a jelenlegi állapotból minden jármű összes lehetséges mozgatásából generál további állapotokat. Heurisztika alapján kiszámolja az egyes állapotok értékét és a legkisebb értékű állapotot fejti ki, ameddig nem találja meg a megoldást vagy eléri az ötvenhárom lépést.

* 1. Technológiák
     1. Unity

A játék motorjának a Unity-t használtam, ami támogatja a C# nyelvet. Elsősorban az érthető és részletes dokumentációja miatt választottam, valamint az ingyenes boltból (Asset Store) könnyen hozzáférjek és integráljak modelleket a játékba. Egyik erőssége más motorokhoz képest, hogy rengeteg előre létrehozott függvényt tartalmaz, ami megkönnyíti a fejlesztést. Többek között matematikával kapcsolatos függvényeket és beépített fizikai motort is tartalmaz. A fejlesztés alatt és a végleges játékban könnyen lehet teszteket írni és futtatni a Unity Test Framework segítségével. Felhőalapú szinkronizálást és verziókövetést is biztosít.

* + 1. C# programozási nyelv

„A C# programozási nyelv a Microsoft új fejlesztési környezetével, a 2002-ben megjelent Visual Studio.NET programcsomaggal, annak részeként jelent meg”. [4]

* + 1. Adatbázis

Egy lokális SQLite adatbázis fog felelni az adatok tárolásáért. Azért döntöttem így, mivel a Unity motor is támogatja az SQLite-ot és ezért nem igényel előkészítést, valamint kevés adatot tárol az offline módban működő játékom.

Az SQLite azért is volt jó választás, mert könnyen egy fájlban el lehet tárolni az adatokat és nem szükséges szerver a működtetéshez. Minimális erőforrásokat használ és ezért alkalmas akár mobil, konzolos vagy asztali játékokhoz, amelyeket a Unity is támogat, így könnyen lehet, más platformra játékot készíteni. [5]

1. Szoftver tervezése
   1. Funkciók
      1. Játék nehézségének kiválasztása

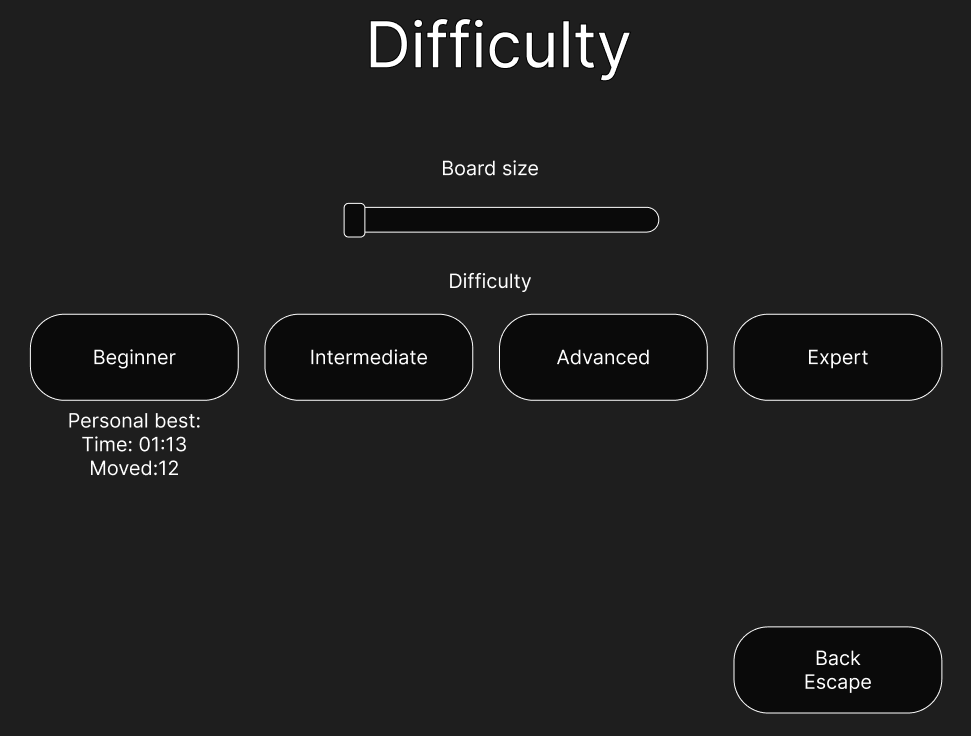
A fizikai Rush Hour játékban négy nehézségi szintet különböztet meg, amelyek a kezdő, gyakorlott, haladó és nagymester.

A szakdolgozatban elkészített játékban is ezeket a szinteket különböztettem meg.

A nehézség függ a pálya nagyságától és a kiválasztott szinttől. A szintek közötti különbség az autónk kezdő pozícióját módosítja úgy, hogy minél nehezebb a feladvány annál távolabb kerül a kijárattól. A generálás során a szinttől függ hány járművet helyezhetünk le a pályára és azok hogyan helyezkednek el. Minden jármű, ami a pályára kerül egy előző járműnek állja az útját akár több irányból is, a nehézség módosításával a járműveket több autó blokkolhatja.

A felületi elemek (1.ábra):

* Pálya mérete (slider)
* Nehézség gombok



1.ábra: Nehézség beállítások menüterv

A pálya méretét egy Slider-rel lehet kiválasztani 4-től 7-ig korlátozva és a nehézségi szintet a megfelelő gomba kattintva. Az egyes nehézségek alatt megjelennek az eddig elért legjobb idő és mozgatási szám. A nehézséget kiválasztva a játék elindul a pálya generálása után.

* + 1. Játék ciklus

A játék a generálás folyamata után kezdődik és a feladvány megoldásáig vagy a felhasználó által történő megszakításig tart. A játék végén megjelenő felület a GameOver menü, amiben a feladvány megoldási ideje és lépés száma szerepel. A játékos eldöntheti, hogy folytatja-e a játékot vagy kilép a menübe.

Felületi elemek (2.ábra):

* Idő szöveg
* Lépés szám szöveg
* Új játék gomb
* Beállítások gomb
* Kilépés a menübe gomb
* Kilépés a játékból gomb

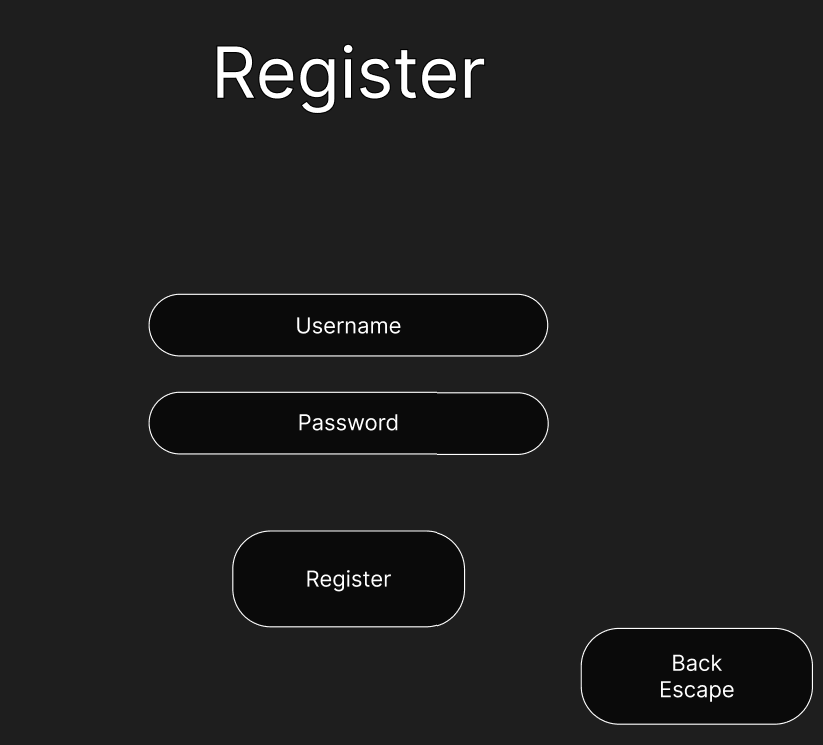


2.ábra: Játék vége felület terve

* + 1. Regisztráció

A felületi elemek (3.ábra):

* Felhasználónév beviteli mező
* Jelszó beviteli mező



3.ábra: Regisztrációs felület terve

A játék működéséhez nem szükséges regisztráció és bejelentkezés, mivel az adatbázis létrehozása során egy vendég fiókba lép be és ezt használja.

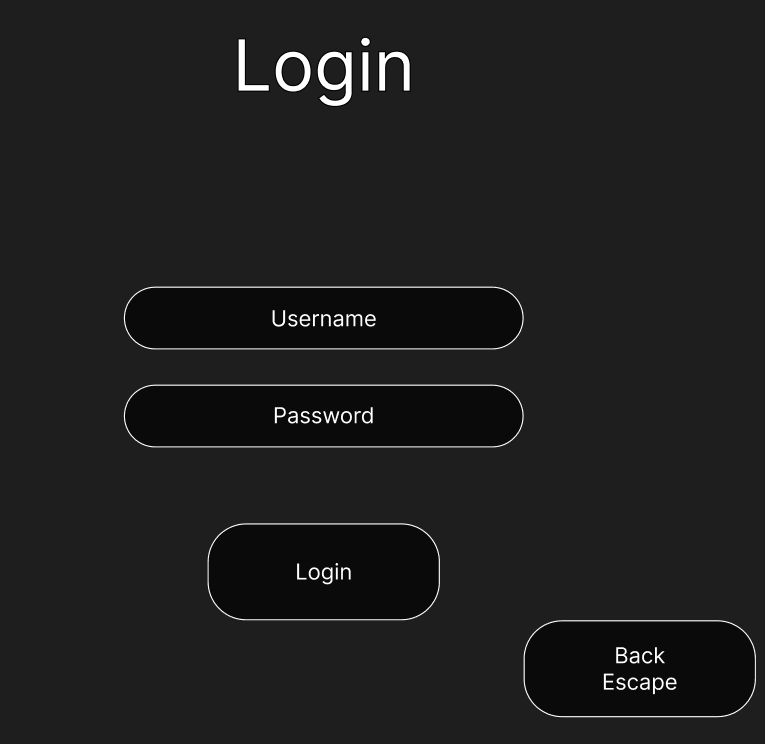
A regisztráció segít abban, ha esetleg több felhasználó szeretné a rekordjaikat vagy beállításaikat menteni, akkor ezek ne kerüljenek felülírásra a másik által és a fiókukhoz kötődjön.

A regisztráció során a felhasználó megadja a felhasználó nevét és a jelszavát, ami hash kódolva kerül az adatbázisba. A művelet sikeréről egy felugró üzenet értesíti a felhasználót. Sikertelen regisztráció történhet akkor, ha már létezik ilyen felhasználó vagy abban az esetben, ha a beírt adatok nem tesznek eleget a feltételeknek. A felhasználónévnek legalább 5 karakter hosszúnak kell lennie, a jelszónak pedig 10 karakter hosszúnak.

* + 1. Bejelentkezés

A felületi elemek (4.ábra):

* Felhasználónév beviteli mező
* Jelszó beviteli mező



4.ábra: Bejelentkezés felület terve

A játék minden indításnál a vendég felhasználóba lép be automatikusan. A bejelentkezés felületen beléphetünk egy regisztrált fiókba. A játékosnak meg kell adnia a felhasználó nevét és jelszavát. A sikeres bejelentkezés után a fő menü oldalra történik navigálás.

* + 1. Ranglista

A felületen (5.ábra) megjelennek az adatbázisba regisztrált játékosok eredményei a pálya méretétől és nehézségétől függően. A pálya méretét egy slider-el és a nehézségi szintet a lenyíló listából lehet kiválasztani. A lista változtatások után frissül és az új eredményeket mutatja. A sorrend a legkevesebb mozgatásból megoldott feladványt részesíti előnyben és utána idő szerint rendezi növekvő sorrendben.

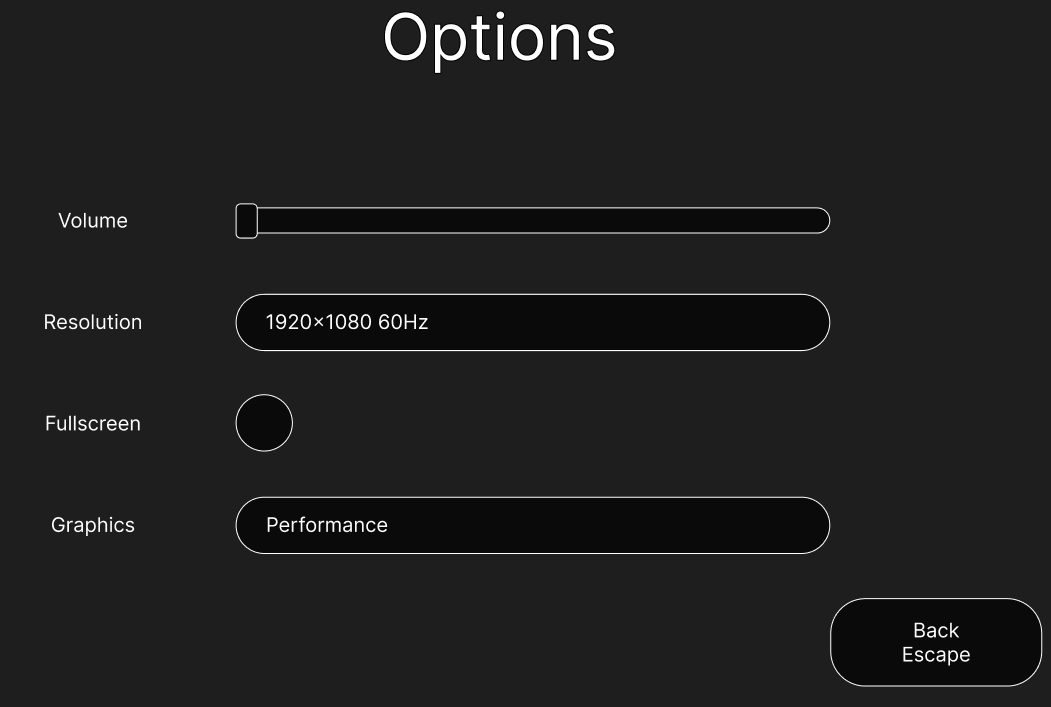


5.ábra: Ranglista felület terve

* + 1. Beállítások

Az oldalon a következő beállítási lehetőségeket lehet beállítani (6.ábra):

* Hangerő
* Felbontás
* Teljes képernyő
* Grafika



6.ábra: Beállítások felület terve

A beállításokat a bejelentkezett fiókhoz menti el, viszont ennek hiányában a vendég felhasználó beállításai íródnak felül.

A felületen a hangerő-t egy slider, a felbontást és grafikát egy lenyíló lista és a teljes képernyős módot egy checkbox segítségével lehet állítani.

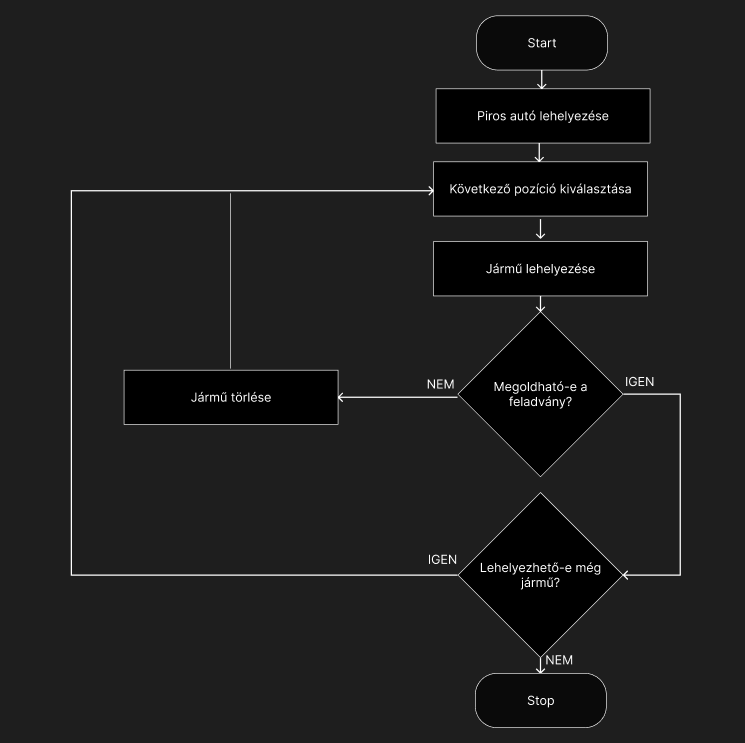
* + 1. Környezeti elemek

A játék egy parkolóban helyezkedik el és ezt egy város veszi körbe, ahol autók közlekednek. Több féle autó közlekedik az utakon a közlekedési szabályokat betartva. A kereszteződésekben közlekedési lámpák irányítják a forgalmat.

A játékban napi ciklus érzékelteti az idő múlását.

* 1. Algoritmusok működése
     1. Generáló algoritmus

Az algoritmus (7.ábra) a nehézség kiválasztása után egyből elkezdődik egy töltő képernyővel jelezve a játékosnak. Az algoritmus a piros autó lehelyezésével kezdődik és a nehézség és pálya méret alapján kiválasztja a következő autó pozícióját. Ezután letett jármű blokkolja az ez előttit lehetőleg mind a két irányból. Minden jármű után lefut a megoldó algoritmus, hogy biztosan megoldható legyen a feladvány. Az a jármű törlésre kerül, amelyik miatt nem lenne megoldható a feladvány. A megoldás eldöntése után, ha már nem helyezhető le több jármű, akkor befejeződik a generálás és a töltő képernyőt is felváltja a játék felülete.



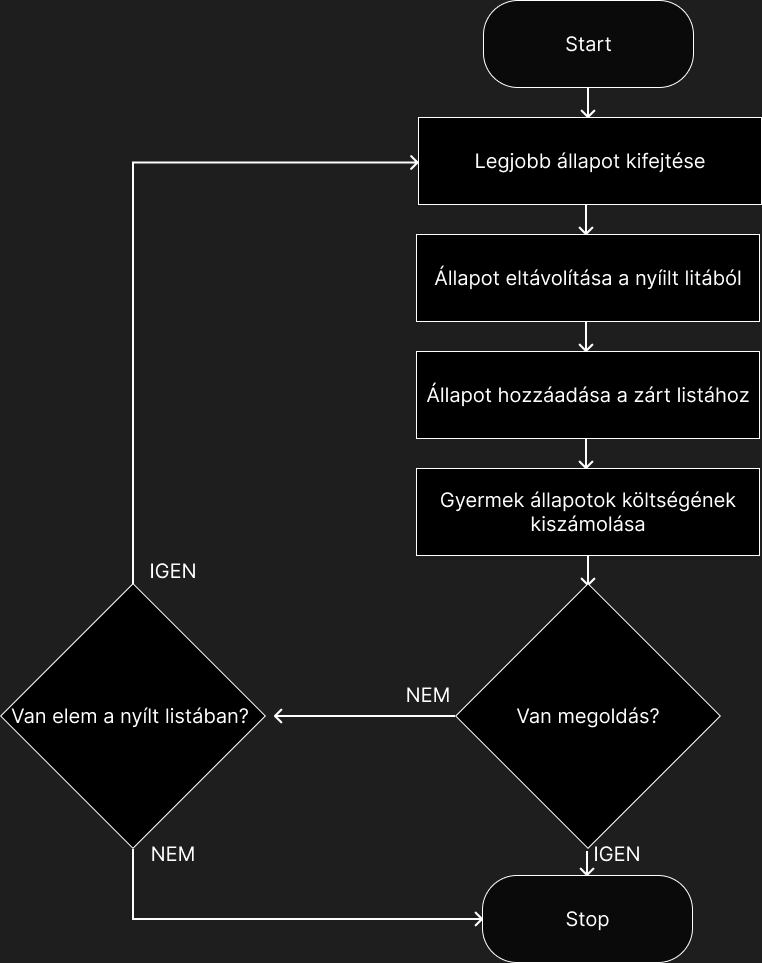
7.ábra: Generáló algoritmus folyamatábra

* + 1. Megoldó algoritmus

A solver generálás és játék közben is lefut. A generálás közben segít eldönteni, hogy egy feladvány megoldható-e és az algoritmus végén visszaadja hány lépésből oldható meg.

Játék közben a játékos a pálya jelenlegi állásából kiindulva elindíthatja a solver-t, ami megmutatja és meg is oldja a feladványt az autókat mozgatva.

Van lehetőség a feladvány visszaállítására is, ami szintén ugyanazzal az algoritmussal, visszaállítja a pályát a kezdő állapotba és nullára állítja az időt és mozgatások számát.



8.ábra: Megoldó algoritmus folyamatábra

* + 1. Járművek és pálya megjelenítése

A dolgozat során a logika és a megjelenítés szétválasztására törekedtem, az olvashatóbb kód, a könnyebb tesztelési lehetőségek és a javíthatóság növelése érdekében, ezért a megjelenítés kódja akkor fut le, amikor már a logikai részek lefutottak.

A pályát egy 2 dimenziós tömbként hozza létre a játék, ami a pálya méretétől függően feltölti az összes értéket nullával. A pálya létrehozása mellett létrejön egy olyan lista is, ami a 2 dimenziós tömb minden értékére létrehozza az oda letehető járművek tulajdonságait.

Miután létrejön az üres két dimenziós tömb lehelyezhető a pálya. A kijáratot külön pirossal jelöli.

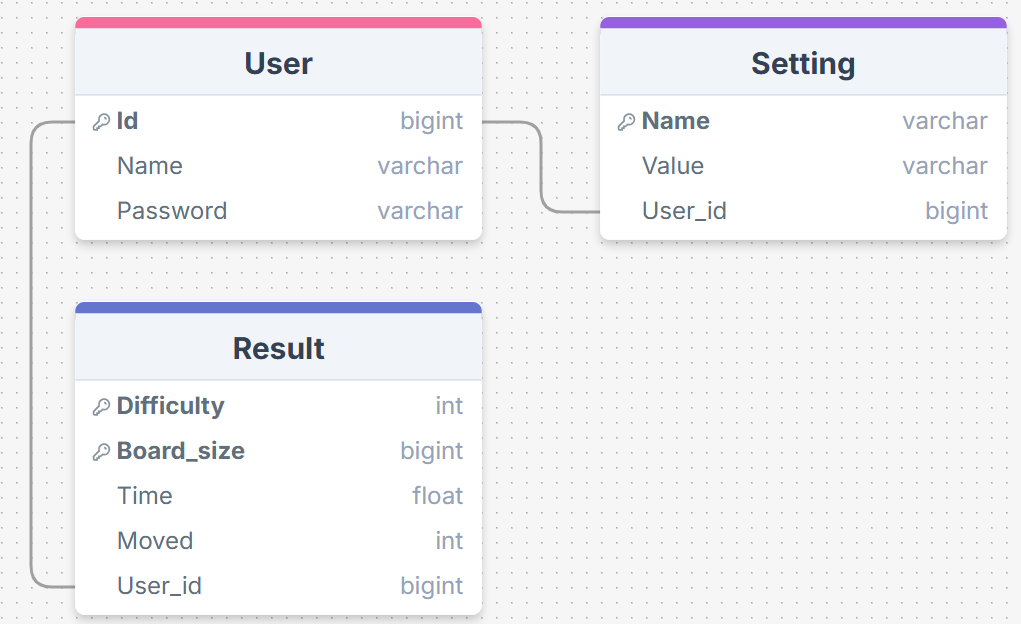
* 1. Adatbázis
     1. Adatbázisterv

SQLite adatbázisban lokálisan eltárolható adatok segítségével a játékban módosított eredmények, beállítások elmenthetők.

A játék indításakor az adatbázis törlés esetén is automatikusan létrejön és előkészíti a felhasználó számára.

Az adatbázis 3 táblából áll (9.ábra).

* felhasználó
* eredmény
* beállítás.



9.ábra: Adatbázisterv

A felhasználó táblában eltároljuk a felhasználónevet és jelszót. A jelszó hash kódolva kerül tárolásra. A játék indulásakor létrejön egy vendég felhasználó fiók, amiben minden adat eltárolódik, így nincs szükség regisztrációra a játék használatához. A funkció arra az esetre került be a játékba, ha esetleg több felhasználó szeretne egyszerre ugyanabban a játékban játszani, akkor külön elmenthetik eredményeiket.

Az eredmény táblában a pálya mérete, nehézsége és felhasználó azonosítója alapján kerül tárolásra a teljesített idő és mozgatások száma.

A beállításokban a fiókhoz kötve beállítható a játék hang ereje, felbontása, teljes képernyős módja és grafikája, amiket a játék indításakor be is tölt.

1. Szoftver kialakítása
   1. Unity
      1. Történelem

Az első Unity-verziót (1.0.0) David Helgason, Joachim Ante és Nicholas Francis alkotta meg Dániában. Az első termék 2005. június 6-án jelent meg. Céljuk egy megfizethető játékmotor létrehozása volt, professzionális eszközökkel az amatőr játékfejlesztők számára, miközben „demokratizálni kívánták a játékfejlesztési ipart”. [6]

* + 1. Hasznos funkciók

A Unity motor tartalmaz olyan matematikával kapcsolatos függvényeket, amelyek elengedhetetlenek egy játék elkészítéséhez. A dolgozat során használt függvény példák:

* Lerp
* Distance
* ScreenPointToRay
* Random.Range

A Lerp függvény a lineáris interpoláció-t valósítja meg. Lerp(a, b, t) = (1 - t) \* a + t \* b

A lineáris interpoláció két pont közötti legegyszerűbb interpolálási módszer egyenes vonalban [7].

A Lerp függvény a környezetben közlekedő járművek mozgatását végzi el, a jármű és előtte egy kis távolság között.

transform.position = Vector3.Lerp(transform.position, transform.position + transform.right \* step, Time.deltaTime \* speed);

A Distance függvény két pont közötti távolságot adja vissza. A járművek táblán végzett mozgatásánál a legközelebbi mező megtalálásánál van használatban.

Vector3.Distance(hitted.transform.parent.position, moveTo) < minDistanceFromClosestCell \* vehicle.size / 2

A ScreenPointToRay egy sugarat ad vissza a világtérben a kamera közeli síkjából indulva átmegy a pozíció (x,y) pixelkoordinátáin a képernyőn. A játékban a járművek az egérrel való mozgatásához nyújt segítséget, az egér pixelkoordinátáit sugárrá alakítani.

A standard sugár egy olyan entitás, amely egy adott térbeli pozícióból és egy irányból áll, amely meghatározza, hogy merre haladjon a térben. Ezt úgy is elképzelhetjük, mint egy lézert: elhelyezünk egy lézerpontot, majd onnan egy lézert bocsátunk ki, amely egyenes vonalban halad, és végül eléri valamilyen objektumot [8].

Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);

A Random.Range függvény egy véletlenszerű számot egyeneletes eloszlásból ad vissza két megadott szám között. A játék során a feladvány létrehozásánál van felhasználva.

Random.Range(0, forwardPlaceList.Count - 1)

* 1. Menürendszer
     1. Állapotok kezelése

Minden menü egy egység osztályon keresztül elérhető és értékeik állíthatók. A UIManager osztályon belül elérhető állapot változó (state), valamint előző (previousMenu) és jelenlegi (currentMenu) menü is eltárolódik. Az egyes felületek példányaira ezen az osztályon belül lehet hivatkozni.

A menük egyszerűbb kezelésére enum-mal lehet hivatkozni az egyes felületekre.

Minden menünek külön osztály példánya van, amiben szerepelnek az egyes felületi elemek, mint például lenyíló listák, gombok, szövegek és a felülethez tartozó funkciók.

* + 1. Navigáció

A navigációt az egység osztályon belüli függvények segítségével hajtja végre a játék.

A navigáció megkezdése előtt a mostani felületet az előző menü változóba kerül mentésre. Az a felület, amelyre navigál a jelenlegi menü változóba kerül. Értékadás után egy Coroutine függvény kerül meghívásra, amelyben a menü váltás animációjának végét várja meg. A SetNextMenu függvényben történik meg a tényleges navigáció. Az állapotváltozó megkapja a navigált oldal értékét. Lehetőség van gombok megnyomására enter gombbal és az előző oldalra navigálni Escape gombbal.

public void SetMenuActive(Menu menuCode)

{

previousMenu = GetMenuActive();

currentMenu = menuCode;

StartCoroutine(waitForOutAnimation());

for (int i = 0; i < menus[(int)currentMenu].transform.childCount; i++)

{ menus[(int)currentMenu].transform.GetChild(i).GetComponent<Animator>().SetBool("Out", false); }

SetNextMenu(currentMenu);

if (menuCode == Menu.Pause) { pauseUI.paused = true; }

else { pauseUI.paused = false; }

}

* 1. Menü elemek

A Unity motor alapvetően támogatja a menü elemek közötti navigációt, viszont ehhez ki kell jelölni egy elemet. A játék egy új felület betöltésekor automatikusan kijelöli az első interaktálható elemet, ha van és az adott oldalon az elemek között a billentyűzet segítségével a nyilakkal vagy Tabulátor gombbal lehet váltani.

public void NavigateWithTab()

{

currentInteractableIndex++;

if (currentInteractableIndex > interactableElements.Count - 1)

{ currentInteractableIndex = 0; }

EventSystem.current.SetSelectedGameObject(interactableElements[currentInteractableIndex].gameObject);

}

* 1. Adattárolás
     1. Adatbázis létrehozása

Az adatbázis a játék elején mindig létrejön, ha nem található egy. A database osztályon belül a CreateDatabase függvény egy SqliteConnection példány segítségével SQL parancsok alapján létrejönnek a szükséges táblák és a vendég fiók, amibe meg is történik a bejelentkezés. A bejelentkezett felhasználó ebben az osztályban kerül tárolásra és ezen az osztályon keresztül érhető el.

Az SQL injection elkerülése érdekben minden SQL parancs paraméterként kerül átadásra és futtatásra.

Az SQL injection egy olyan biztonsági sebezhetőség, amely adatbázis-alapú webalkalmazásokban fordul elő, amikor egy támadó rosszindulatú kódot juttat be az alkalmazásba, hogy jogosulatlan hozzáférést szerezzen érzékeny információkhoz [9].

* + 1. Modell osztályok

A modell osztályok az adatbázisban létrehozott táblák kezelhetőségét segíti C# nyelven.

* + 1. Kezelő osztályok

Minden táblának műveletei külön kezelő osztályban találhatók, amiket a Database osztályon keresztül lehet elérni a kezelő osztály példányából. Az összes tartalmazza az adatbázisba illesztő INSERT utasítást.

A felhasználó kezelő osztályban a következő függvények vannak:

* RegisterUser
* LoginUser
* HashPassword
* LogoutUser
* GetUserByUserName

A RegisterUser beilleszti a felhasználó által megadott adatokat, ha az megfelel a követelményeknek. A felhasználónévnek minimum 5 karakter és a jelszónak minimum 12 karakter hosszúnak kell lennie. Abban az esetben, ha nem található ilyen regisztrált felhasználó, akkor beillesztésre kerül az adatbázisba. A jelszó hash kódolva kerül tárolásra.

string HashPassword(string password)

{

using (SHA256 sha256 = SHA256.Create())

{

byte[] passwordBytes = Encoding.UTF8.GetBytes(password);

byte[] hashedBytes = sha256.ComputeHash(passwordBytes);

return Convert.ToBase64String(hashedBytes);

}

}

A kriptográfiai hashalgoritmus egy olyan algoritmus, amely tetszőleges hosszúságú bemeneti karakterláncokat vesz és ezeket rövid, fix hosszúságú kimeneti karakterláncokra térképezi. Az SHA-256 tömörítési függvénye 64 körből áll, kétféle nem-lineáris függvényt, ciklikus elforgatásokat és körökre jellemző konstansokat tartalmaz [10].

A felhasználót felugró ablak értesíti a művelet sikerességéről.

PopUp.SetActive(true);

PopUpMessageText.text = error.message;

A LoginUser ellenőrzi, hogy a beírt adatok alapján létezik-e ilyen felhasználó és helyes adatokat adott-e meg. Egy felugró ablak értesíti a felhasználót a bejelentkezés sikerességéről vagy az egyes hibákról. Hibát okozhat, ha nem létezik olyan felhasználónévvel regisztrált játékos, vagy helytelen jelszót adott meg.

if(checkUser.username == user.username && checkUser.password == HashPassword(user.password))

            {

                connection.Close();

                database.loggedInUser = checkUser;

                settings.GetSettings();

                return new Error(true, "You are logged in!");

            }

Bejelentkezés után a fő menüben is látható a játékos felhasználóneve és a bejelentkezés gomb helyére a kijelentkezés gomb kerül.

A LogoutUser függvény átállítja az adatbázis osztály bejelentkezett felhasználóját a vendég felhasználó adataira.

public void LogoutUser()

    {

        database.loggedInUser = new User(1, "guest", "pass");

    }

A felugró üzenet értékét az Error osztályban található message változóval lehet állítani. A háttérszíne az isSuccessful változótól függ.

public class Error

{

    public bool isSuccessful;

    public string message;

    public Error(bool isSuccessful, string message)

    {

        this.isSuccessful = isSuccessful;

        this.message = message;

    }

}

A GetUserByUsername függvény paraméterként fogad egy felhasználónevet és igazként tér vissza, ha létezik ilyen nevű felhasználó az adatbázisban, ellenkező esetben hamis lesz a visszatérési érték.

command.CommandText = "SELECT \* FROM `User` WHERE username = @username;";

command.Parameters.AddWithValue("@username", username);

A beállítások kezelő osztályában a GetUserSettings függvény a játék indításakor és bejelentkezéskor fut le, hogy a mentett beállításokat felülírja.

command.CommandText = "SELECT name, value FROM `Setting` WHERE user\_id = @user\_id;";

command.Parameters.AddWithValue("@user\_id", database.loggedInUser.id);

Minden beállítás egy szótárban kerül tárolásra, amiben a kulcs a beállítás neve, mint lehet volume, fullscreen, resolution és graphics. Mivel nem biztos, hogy minden beállítást módosított a felhasználó, ezért a játék indításakor alapértelmezett értéket kap az a beállítás, ami nem található a felhasználó által mentett beállítások között.

Dictionary<string, string> settings = database.settingHandler.GetUserSettings(database.loggedInUser);

        if (settings.ContainsKey("volume"))

        {

            volumeSlider.value = float.Parse(settings["volume"]);

        }

        else

        {

            volumeSlider.value = 0;

        }

Bármelyik beállítás módosítása esetén az éppen bejelentkezett felhasználó beállításaihoz kerül mentésre az adatbázisban, ha nincs bejelentkezve játékos akkor a vendég felhasználó beállításai módosulnak.

Az eredmények kezelő osztályában a GetResultsByBoardSize függvény az éppen bejelentkezett felhasználó eddigi eredményeit adja vissza a pálya méretétől függően.

command.CommandText = "SELECT \* FROM `Result` WHERE board\_size = @size AND Result.user\_id = @user\_id;";

command.Parameters.AddWithValue("@user\_id", database.loggedInUser.id);

command.Parameters.AddWithValue("@size", size);

Az adatok a nehézség választó felületen jelennek meg az egyes nehézség kiválasztó gombok alatt. A visszaadott listából az adott nehézség alatt található szöveg kerül felülírásra.

List<Result> results = databaseInstance.resultHandler.GetResultsByBoardSize(gameDataInstance.boardSize);

        for (int difficulty = 1; difficulty <= 4; difficulty++)

        {

            Result result = results.Find(res => res.difficulty == difficulty);

            if (result != null)

            {

                PRTexts[difficulty - 1].text = "Personal best\nTime: " + gameDataInstance.GetTimeInString(result.time) + "\nMoves: " + result.moved;

            }

        }

A függvény minden alkalommal lefut, amikor a játékos bejelentkezik másik fiókba vagy a pálya méretét változtatja.

A GetResultsByBoardSizeAndDifficulty függvény visszaadja az összes eredményt felhasználótól függetlenül egy adott pályaméreten és nehézségen. A visszakapott lista a ranglistába kerül.

SELECT User.username, moved, time

FROM `Result` LEFT JOIN `User` ON User.id = Result.user\_id

WHERE board\_size = @size AND difficulty = @difficulty

ORDER BY moved ASC, time ASC LIMIT 50;

A függvény lefut a ranglista felület betöltésekor, illetve ha a pálya mérete vagy a nehézsége módosul.

* 1. Feladvány létrehozása
     1. Nehézség és pályaméret kezelése

A nehézség kiválasztás felületen lehet állítani a pályaméretet és nehézséget a játék indítása előtt. Minden nehézség egy számnak felel meg a kezdő az egyes számot jelenti és a nehézség növekedésével inkrementálódik, a profi szintig, ami a négyes számot jelenti.

* + 1. Generálás folyamata

A generálás egy aszinkron folyamatként fut le, mivel egy töltőképernyő jelenik meg az algoritmus kezdetén.

A folyamat elején, ha már történt generálás, akkor az előző generált pálya törlésre kerül és az új kerül a helyére.

A törlés után a pálya méretétől függően lehelyeződik minden mező és a piros autó a pálya egy véletlenszerű pontjára a harmadik oszlopban. A pozíciója függ a pályamérettől és nehézségtől. A nagyobb nehézségeken távolabb kerül a cél mezőtől.

await InsertVehicle(CreateVehicle(1, 2, new int[] { Random.Range(Mathf.Min((int)(2 \* (int)gameDataInstance.difficulty / 1.5f), boardInstance.size - 2), boardInstance.size - 1), 2 }, Direction.Vertical), boardInstance.board);

Az CreateVehicle függvény az kapott paraméterek alapján visszaad egy Vehicle példányt, ami tartalmazza a jármű azonosítóját, méretét, kezdő pozícióját és orientációját.

A kezdőpozíció az autó első koordinátáját tartalmazza egy tömbben. Az összes koordináta lekérdezhető egy listában az orientációtól függően a GetPosition függvénnyel.

for (int coordinate = 0; coordinate < this.size; coordinate++)

{

   position.Add(new int[] { startPosition[0] + (direction == Direction.Vertical ? coordinate : 0), startPosition[1] + (direction == Direction.Horizontal ? coordinate : 0) });

}

Az orientáció egy Direction enum, aminek értéke lehet Vertical, illetve Horizontal.

public enum Direction

{

    Vertical,

    Horizontal

}

Az InsertVehicle függvény a létrehozott járművet illeszti be az adott mezőre. A lehelyezett járműhöz tartozó Place objektum törlésre kerül az összes lehetséges lehelyezhető jármű listájából és meghívja a megoldó algoritmust. A Place osztály írja le a lehetséges helyek listájának egy elemét. Egy elem tartalmazza a méretet, pozíciót és orientációt. A tulajdonságok alapján megadható minden mezőre milyen jármű helyezhető le.

public class Place : IComparable<Place>

{

    public int size;

    public int[] placePosition;

    public Direction direction;

}

Abban az esetben, ha a megoldó algoritmus hamisan tér vissza, akkor a feladvány nem megoldható és az InsertVehicle függvény is hamisan tér vissza, viszont ha feladvány megoldható, akkor a RemovePlace függvény hívódik meg, amiben a csúcsforgalom játék szabályai szerint a lehetséges helyek listájából törlésre kerülnek az így helytelen helyek.

Azok a mezők törlődnek, amelyek nem tesznek eleget a következő szabályok valamelyikének:

* Nem kerül jármű egy másik jármű helyére
* A piros autó elé nem kerülhet függőleges jármű

Példa az egyik szabály esetén előforduló törlésre:

boardInstance.places.RemoveAll(place => place.placePosition[direction] == position[direction] - 1 && place.placePosition[1 - direction] == position[1 - direction] && place.size == 2 && place.direction == (Direction)direction);

A piros autó lehelyezése után a többi jármű generálása kezdődik a PuzzleGenerator osztály GenerateVehicles függvényének meghívásával. A függvény kiértékelődésének száma a nehézség számának kétszerese.

for (int i = 0; i < (int)gameDataInstance.difficulty \* 2; i++)

{

   await GenerateVehicles();

}

A GenerateVehicles függvényben minden jármű úgy jön létre, hogy egy másik már lehelyezett járműnek az útját állja. A lehelyezett autókat sorba véve először szembe és utána hátulra jön létre egy jármű abban az esetben, ha található hely.

A helyek megtalálásához egy segéd listában a pálya létrehozása előtt előre létrehozva szerepel az összes lehetséges pozíció és hogy milyen jármű helyezhető el oda.

A járműveket összesen a pálya méretétől és nehézségtől függően, addig helyezzük le ameddig már nem található üres hely vagy a ciklus változó eléri a pálya méret \* nehézség értéket.

A szemben blokkoló jármű lehelyezése:

List<Place> forwardPlaceList = boardInstance.GetPlace(vehicles[id], true);

if (forwardPlaceList != null)

{

   while (forwardPlaceList.Count > 0 && !firstResult)

   {

       Place place = forwardPlaceList[Random.Range(0, forwardPlaceList.Count - 1)];

        forwardPlaceList.Remove(place);

     firstResult = await InsertVehicle(CreateVehicle(vehicles.Count + 1, place.size, place.placePosition, place.direction), boardInstance.board);

    }

}

Amikor már nem helyezhető le jármű, akkor az algoritmus futása befejeződött és a játékos számára a töltőképernyő eltüntetésével és a játék felület megjelenésével van jelezve.

* + 1. Megoldhatóság ellenőrzése

A megoldó algoritmus aszinkron módon fut le.

A Solver osztály Search függvényének meghívásakor fut le a kereső algoritmus, a játékos számára egy töltőképernyő jelzi az algoritmus futását.

A megoldás folyamata egy gráfban történik, aminek csak a kezdő állapota ismert és a végállapotnak a megállapítására csak a heurisztika szolgál. A Node absztrakt osztály egy állapotot ír le, amiben van egy tábla, ami a jelenlegi állapot pályáját írja le 2 dimenziós tömbben, egy mélység változó, a heurisztika értéke, egy referencia az abban az állapotban mozgatott járműről és a szülő állapot. Tartalmaz egy GetChildren függvényt, amit meghívva az adott állapot összes lehetőség következő gyermek állapot legenerálható. A függvény a GetVehicles függvény meghívásával kezdődik, ami az adott állapot táblájában található összes járművét adja vissza egy listában. Gyermek állapotot a járművek mozgatásával készül el. Minden lehetséges mozgatási lehetőségből készül egy új állapot a mélységet növelve. A CreateChild függvény egy virtuális függvény, amit az öröklött osztály ír felül, az adott öröklött osztályból készít egy új példányt.

List<Vehicle> vehicles = GetVehicles();

List<Node> children = new List<Node>();

foreach (Vehicle vehicle in vehicles)

{

   int[,] board = (int[,])this.board.Clone();

    for (int i = 0; i < vehicle.possibleMoves.Count; i++)

    {

        Node node = CreateChild(vehicle, vehicle.possibleMoves[i], board);

        children.Add(node);

    }

}

A Node osztálytól öröklődik a NodeForSolution és NodeToCompareBoard osztályok, amelyek a megoldó algoritmust alkalmazzák más heurisztikával. A NodeForSolution heurisztikája a feladvány megoldására függ a piros autó távolságától a cél mezőtől és attól, hogy hány jármű álla az útját közte és a cél között. A távolság mezőnként egy értékkel növeli, az útjában álló járművek a pálya méretének nagyságával növelik a heurisztika értékét, végül ehhez hozzáadódik a mélység értéke is.

for (int i = 0; i < board.GetLength(0); i++)

{

   if (board[i, 2] != 1 && board[i, 2] != 0)

   {

        cost += board.GetLength(0);

   }

    if (board[i, 2] == 1)

    {

        cost += i;

        break;

    }

}

cost += depth;

A NodeToComapreBoard heurisztikája a feladvány visszaállítására a teljes táblát hasonlítja össze a játék indításakor legenerált pályával. Minden olyan mező, ami nem egyezik meg ezzel a táblával egyel növeli az értékét.

cost += (board.GetLength(0) \* board.GetLength(1));

for (int i = 0; i < board.GetLength(0); i++)

{

    for (int j = 0; j < board.GetLength(0); j++)

   {

        if (board[i, j] == goalBoard[i, j])

        {

           cost--;

        }

    }

}

A Graph osztály tartalmazza az adott gráf nyílt és zárt listáját. A nyílt lista tartalmazza a kifejtendő állapotokat heurisztika szerint sorba rendezve. A zárt listában a már kiejtett állapotok vannak.

public class Graph

{

    public List<Node> openList;

    public List<Node> closedList;

    public Graph()

    {

        openList = new List<Node>();

        closedList = new List<Node>();

    }

}

A Solver osztályban lévő Search függvény hajtja végre a megoldás keresését. Az algoritmus az gráf létrehozásával és az első állapot felvételével indul.

Graph graph = new Graph();

Node firstNode;

firstNode = new NodeForSolution(firstBoard, 0);

graph.openList.Add(firstNode);

Az algoritmus addig fut ameddig a nyílt listában található elem vagy ameddig el nem éri az 53-as mélységet. A legjobb állapot kifejtésével a nyílt listából a zárt listába kerül.

bestNode = graph.openList.First();

graph.openList.Remove(graph.openList.First());

graph.closedList.Add(bestNode);

List<Node> children = bestNode.GetChildren();

A gyermek állapotokat megvizsgálva abban az esetben, ha a heurisztika értéke nulla, vagyis a feladvány megoldódott, akkor visszatér a függvény igaz értékkel, ha nem és még a nyílt vagy zárt listában ez az állapot nem szerepel, akkor bekerül a nyílt listába.

if ((children[nodeIndex].cost - children[nodeIndex].depth) == 0)

{

   stepsToSolve = children[nodeIndex].depth;

   resultNode = children[nodeIndex];

    return await Task.FromResult(true);

}

else

{

   if (!graph.openList.Any((node) => node.Equals(children[nodeIndex])) && !graph.closedList.Any((node) => node.Equals(children[nodeIndex])))

    {

        graph.openList.Add(children[nodeIndex]);

    }

}

A lépések visszafejtésére a Node osztályokban található szülő referencia segítségével megkapható a megoldás útja.

while (node.parent != null)

{

   solution.Add(node);

    node = node.parent;

}

solution.Reverse();

* 1. Játékmenet
     1. Jármű kijelölése

A generálás folyamata után a játék elindul és az autók el vannak helyezve a pályán. Ezek után a játék felülete jelenik meg és az első indításkor egy segítő felület, ami leírja az iránytást.

Egy járműre kattintva és az egeret mozgatva a jármű követi az egér mozgását. A kijelölés a képernyő egér koordinátáiból indított sugárral történik. A sugár addig halad ameddig el nem talál, egy olyan objektumot, aminek a LayerMask tulajdonsága jármű, ekkor az eltalált GameObject mentésre kerül egy változóba, amiből elérhető minden komponens.

RaycastHit hit;

Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);

if (Physics.Raycast(ray, out hit, Mathf.Infinity, carLayer))

{

   hitted = hit.transform.gameObject;

}

A kijelölés után a jármű Vehicle nevű komponenséből megkapható a pontos helyzete, mérete és iránya.

VehicleComponent vehicleComponent = hitted.GetComponent<VehicleComponent>();

vehicle = new Vehicle(vehicleComponent.id, vehicleComponent.size, vehicleComponent.startPosition, vehicleComponent.direction, boardInstance.board);

A kijelölt járművet a játékos felé egy fehér körvonal jelzi.

* + 1. Jármű mozgatása

A kiválasztott járművet ameddig a játékos nyomva tartja a gombot az egér mozgatásával lehet elhelyezni a kívánt pozícióra. A kiválasztáskor az adott jármű Vehicle osztályában lefutó GetMovablePosition függvény visszaadja az összes olyan pozíciót, ahova az autó lehelyezhető, ezeket a pozíciókat zöld körvonallal jelöli a játék. Amikor a játékos elengedi az egér gombát a jármű jelenlegi pozíciójához képest a legközelebbi mező helyére kerül, ez a hely vastagabb zöld körvonallal van jelölve.

A GetClosestPosition függvény az egér lenyomásakor állandóan fut és minimum kereséssel kiszámolja a melyik mező van jelenleg a legközelebb. A moveTo nevű változó tartalmazza a jelenlegi legközelebb mezőt.

for (int pos = 0; pos < vehicle.possibleMoves.Count; pos++)

{

 Vector3 current = boardInstance.BoardCoordinateToWordSpace(vehicle.possibleMoves[pos]);

 if (Vector3.Distance(hitted.transform.parent.position, current) < minDistance)

 {

   minDistance = Vector3.Distance(hitted.transform.parent.position, current);

   moveTo = current;

 }

}

A GetOutlineCells az összes olyan pozíciót kijelöli, ahova a jármű letehető a jármű possibleMoves listájából.

foreach (int[] position in vehicle.possibleMoves)

{

   foreach (GameObject cell in spawnGridInstance.instantiatedCells)

   {

if (cell.transform.position == boardInstance.BoardCoordinateToWordSpace(position))

    {

        outlinedCells.Add(cell);

    }

    }

}

A SetOutline a GetOutlineCells függvény visszatérési értékéből vett celláknak a vastagságát állítja be. Tízes vastagságú körvonalra állítja a legközelebbi pozíciót, minden mást három egység vastagságúra.

foreach (GameObject cell in outlinedCells)

{

   if (cell.transform.position == moveTo)

   {

       cell.GetComponent<Outline>().OutlineWidth = 10;

   }

    else

   {

       cell.GetComponent<Outline>().OutlineWidth = 3;

    }

    cell.GetComponent<Outline>().enabled = true;

}

A legközelebbi mezőre történő lehelyezést vizuálisan az objektum pozícióját változtatva történik.

hitted.transform.parent.position = Vector3.Lerp(hitted.transform.parent.position, new Vector3(moveTo.x, (spawnVehicleInstance.vehicleYOffset \* vehicle.size), moveTo.z), Vector3.Distance(hitted.transform.parent.position, moveTo));

A logikai megvalósítást a ModifyBoard osztály MoveVehicle függvény meghívásával történik. A megadott táblából először törlésre kerül a jármű, majd a start pozíciója felülírásra kerül és újra be kerül a táblába. A mozgatás után a GameData osztályban a mozgatások számát inkrementálja.

public void MoveVehicle(Vehicle vehicle, int[] position, int[,] board)

{

RemoveVehicle(vehicle, board);

vehicle.Move(position);

InsertVehicle(vehicle, board);

}

* + 1. Játék vége

Minden mozgatás után, ha a piros autó lett mozgatva a ModifyBoard MoveVehicle függvénye ellenőrzi, hogy a cél mezőre került-e az autó. Abban az esetben, ha a játékos sikeresen megoldja a feladványt megoldó algoritmus nélkül, akkor az adatbázisból az adott feladvány mérete és nehézsége alapján lekérésre kerül az eredmény, ha nem található ilyen rekord, vagy jobban teljesített a játékos mozgatásban vagy időben, akkor felülíródik a rekord.

Result result = new Result(database.loggedInUser.id, (int)gameData.difficulty, board.GetLength(0), gameData.timer, gameData.prevMoved);

List<Result> results = database.resultHandler.GetResultsByBoardSize(board.GetLength(0));

if (results.Count >= (int)gameData.difficulty && results[(int)gameData.difficulty - 1] != null) {

database.resultHandler.AddResult(new Result(results[(int)gameData.difficulty - 1].userId, (int)gameData.difficulty, board.GetLength(0), gameData.timer, gameData.prevMoved)); }

Abban az esetben, ha a játékos megoldó algoritmussal oldja meg a feladványt, nem kerül az adatbázisba rekordként.

A játék végén megjelenik a játék vége felület, ahol a teljesített idő és mozgatások száma jelenik meg. A GameData osztály tartalmazza az idő és mozgatás adatokat.

A felületből van lehetőség új játékot kezdeni, a beállítások felületre váltani, kilépni a fő menübe vagy a játékból.

* 1. Környezeti elemek
     1. Zene

A játék indulásakor automatikusan kiválaszt egy zenét a listából és elindul, a felhasználó által beállított hangerőn.

A kiválasztás egy véletlenszerű szám generálásával történik.

    public void GetARandomClip()

    {

        player.clip = clips[Random.Range(0, clips.Count)];

        player.Play();

    }

A zene hosszát egy időzítő folyamatos inkrementálásával ellenőrzésre kerül.

Ha egy zene véget ér, akkor a zene lejátszó megáll és új zene kerül kiválasztásra és indításra.

if(player.clip.length > time)

        {

            time += Time.deltaTime;

        }

        else

        {

            player.Stop();

            time = 0;

            GetARandomClip();

        }

* + 1. Közlekedő járművek

A logikai játék mellett a játékot színesítve járművek közlekednek.

A járműveket a VehicleSpawner osztály Spawn függvénye helyezi el a Scene-ben időzítő segítségével. Az első jármű lehelyezése után véletlenszerű ideig várakoztatva van a Spawn függvény egy minimum és maximum érték között és ismét egy másik jármű kerül le.

yield return new WaitForSeconds(Random.Range(minWait,maxWait));

StartCoroutine(Spawn());

A VehicleSpawner osztály helyezi le a trolit és vonatot is a pályára, amit az isTrolley és isTrain változók segítségével lehet állítani. A nagy forgalom és a konzisztens képkocka frissítés érdekében egy Spawner maximum 5 autót helyezhet le egyszerre. Autó és kamionok lehelyezése egy két külön listából történik, a troli és vonat külön GameObject prototípusként szerepel az osztályban.

Vonat és troli lehelyezésének logikája megegyezik.

vehicle = Instantiate(Train, vehicleParent.transform.position, Quaternion.identity);

vehicle.transform.eulerAngles = new Vector3(0, 90, 0);

Autó vagy kamion lehelyezése esetén egy size változó segítségével módosítható a pontos pozíció, mivel a modell lokális koordináta rendszerében a középpont a geometriai középpontban található és a pontos pozícióhoz egy szülő GameObjectben ez a lokális koordináta kerül felülírásra.

vehicle = Instantiate(spawnVehiclesInstance.trucks[Random.Range(0, spawnVehiclesInstance.trucks.Count - 1)], vehicleParent.transform.position, Quaternion.identity);

size = 3.55f;

vehicle.transform.parent = vehicleParent.transform;

vehicle.transform.position = new Vector3(vehicleParent.transform.position.x, (vehicleYOffset \* size), vehicleParent.transform.position.z);

vehicleParent.transform.rotation = transform.rotation;

Miután egy jármű a pályára került folyamatosan halad egy egyenes útvonalon egészen addig, ameddig el nem ér egy bizonyos távolságot és törlődik a Scene-ből, vagy megáll egy piros lámpa miatt vagy egy másik jármű az útját állja. A VehicleAI osztályban szereplő stopped változó segítségével állítható meg a jármű, abban az esetben, ha a változó értéke hamis, akkor tovább haladhat előre. A timeSinceStop időzítő akkor indul el, amikor a stopped értéke igaz, ha a timeSinceStop elér egy adott értéket akkor a jármű törlődik a pályáról, a nagy forgalom és az elakadások elkerülése érdekében. A jármű pozíciója a Vector3.Lerp függvény visszatérési értékével változik.

if (stopped == false)

        {

            timeSinceStop = 0;

            transform.position = Vector3.Lerp(transform.position, transform.position + transform.right \* step, Time.deltaTime \* speed);

        }

A jármű RayCast segítségével érzékeli az előtte lévő GameObject-eket és tag alapján azonosítja, hogy piros lámpa vagy másik jármű van előtte, ekkor megáll és a stopped változó értéke igaz lesz.

RaycastHit hit;

if (Physics.Raycast(transform.position, transform.TransformDirection(Vector3.right), out hit, checkDistance))

        {

            if(hit.transform.tag == "car" || hit.transform.tag == "red")

            {

                stopped = true;

            }

}

Az autó törlése abban az esetben történik meg ha túl sokáig áll vagy elér egy bizonyos távolságot a kezdőponttól.

if (Vector3.Distance(transform.position, startPosition) > distanceToDestroy)

{

   Destroy(gameObject);

}

if (stopped == true)

{

   timeSinceStop += Time.deltaTime;

    if (timeSinceStop > timeUntilDestroy)

       {

            Destroy(gameObject);

        }

}

A közlekedési lámpák működését a VehicleStop osztály befolyásolja. A Stop coroutine függvény a wait változó által megadott ideig várakoztatja a végrehajtást.

yield return new WaitForSeconds(wait);

StartCoroutine(Stop());

A stopped nevű változótól függően a piros lámpa aktív lesz, ha a stopped értéke igaz ellenkező esetben az autók át tudnak haladni a kereszteződésen.

boxCollider.enabled = !stopped;

stopped = !stopped;

Vizuálisan a közlekedési lámpa fénye változik a stopped értékétől függően.

redLight.SetActive(!stopped);

greenLight.SetActive(stopped);

* + 1. Napi ciklus

A Scene-ben található Directional Light minden új képkocka után adott mennyiségű fokkal elfordul az X tengelye körül a scale nevű változótól függően, ha eléri a háromszázhatvan fokot, akkor az időzítő alapállapotba kerül, vagyis nulla lesz az értéke.

if(time < 360 \* scale)

{

    time += Time.deltaTime;

}

   else

{

    time = 0;

}

Light.transform.eulerAngles = new Vector3(time / scale, 0, 0);

A Unity-ben a Directonal Light segítségével automatikusan elsötétül a Skybox, az X tengelyen történő forgatás esetén, így egy napi ciklust szimulálva.

1. Tesztelés, módosítások

„Hiba azt feltételezni, hogy a tesztelés automatizálása csupán a manuális tesztelési folyamat rögzítése és lejátszása. Valójában az automatizálás alapvetően eltér a manuális teszteléstől: teljesen különböző problémák és lehetőségek merülnek fel. Még a legjobb automatizálás sem tudja teljesen helyettesíteni a manuális tesztelést, mert az automatizálás a kiszámíthatóságról szól, míg a felhasználók lényegében kiszámíthatatlanok. Ezért érdemes automata teszteket alkalmazni a várt eredmények ellenőrzésére, és manuális tesztelést a váratlan helyzetek kezelésére.” [11]

* 1. Tesztelési csoportok

Az algoritmus teszteket a Unity Test Framework segítségével lehet futtatni. A tesztek lefuthatnak Play mode-ban és Editor mode-ban.

Nyílt forráskódú NUnit egységtesztelési könyvtárat használja, amelyet kifejezetten a .NET nyelvekhez terveztek. A teszteket C# nyelven írják, és kényelmesen futtathatók a Unity szerkesztőbe integrált Unity Test Runner segítségével. [12]

* + 1. Tesztek felépítése

A tesztek külön assembly definition-ben vannak létrehozva, amihez a játék összes kódját hozzácsatolva érjük el.

Minden teszt elején a SetUp dekorátorral ellátott inicializáló függvény fut le az egyes osztályok eléréséhez és Start metódusuk futtatásához. Külön fájlokban található egy-egy osztály függvényeire megírt tesztek. A tesztesetek az egyes metódusok eredményeit tesztelik.

Példa a felépítésre:

[SetUp]

public void Init()

{

    boardInstance = InstanceCreator.GetBoard();

    boardInstance.size = 6;

    boardInstance.GenerateBoard();

}

[Test]

public void ShouldConvertCoordinate()

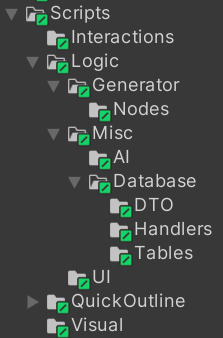
{

Assert.AreEqual(new Vector3(3.5f, 0f, 3.5f), boardInstance.BoardCoordinateToWordSpace(new int[] {1, 1}));

}

* + 1. Algoritmusok tesztelése

A mappaszerkezet felépítése (10.ábra):



10.ábra: Mappa szerkezet

Az algoritmusok tesztje a Logic mappában lévő szkriptekre készült Unit teszteket jelenti. Azokra a metódusokra készül teszteset, amelyek nem igényelnek beviteli eszközt és egyszerű adatszerkezetben visszakapható az eredményük, mint például a legnehezebb csúcsforgalmi feladvány egy két dimenziós tömbben:

int[,] testBoard = new int[,] {

{ 0,2,2,2,4,0 },

{ 3,3,1,0,4,5 },

{ 0,0,1,6,6,5 },

{ 13,10,10,9,7,7 },

{ 13,11,11,9,0,8 },

{ 12,12,12,9,0,8 }

};

A legtöbb függvény, aminek visszatérési értéke véletlenszerű vagy tesztelésük olyan adatszerkezetet igényel, ami a függvény funkcióját lényegtelenné teszi, mint például az összes lehetséges pozíció legenerálása.

* + 1. Megjelenítés tesztelése

A vizuális tesztek szintén csak azokra a függvényekre készültek, amelyek nem igényelnek beviteli eszközt, ezért a menüszerkezet manuálisan lett tesztelve. A vizuális tesztek Play mode-ban futnak le, ami egy új Scene-t hoz létre a Unity Editor-ban és futási időben történik a teszt. A tesztek felépítése megegyezik az algoritmusok tesztjével, viszont nem adatszerkezeteket hasonlít össze, hanem a GameObject-ek létezését pozícióját és orientációját vizsgálja. Ide tartozik a pálya és járművek lehelyezése, illetve a környezetben közlekedő járművek mozgása és közlekedése.

* + 1. Manuális tesztelés

Minden teszt, amire nem megvalósítható egy unit teszt megírása manuálisan lett tesztelve több tesztelő által.

* 1. Tesztek eredménye
     1. Algoritmusok
     2. A csillag és előretekintő keresés összehasonlítása

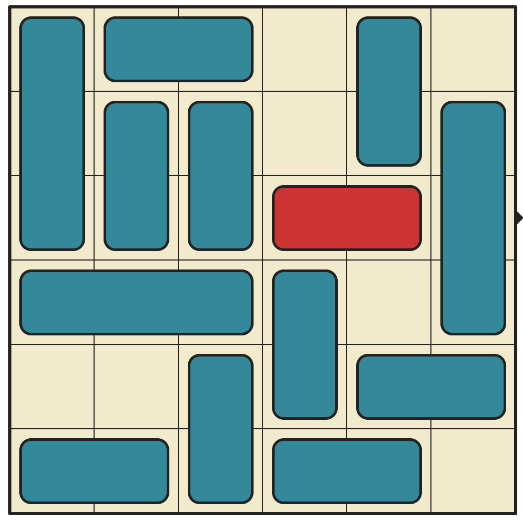
Az előretekintő algoritmus meghatározása:

„Bármely u∈V csúcs esetén t∈T az u preferált célcsúcsa, ha az u-tól t-hez vezető optimális út távolsága nem haladja meg az u-tól T bármely más csúcsához vezető út távolságát… Ha a heurisztikát mint sorrendezést használjuk, akkor a célhoz legközelebb eső csúcsot fogják először felfedezni, figyelmen kívül hagyva a megtett távolságot [13].”

Az A csillag algoritmus meghatározása:

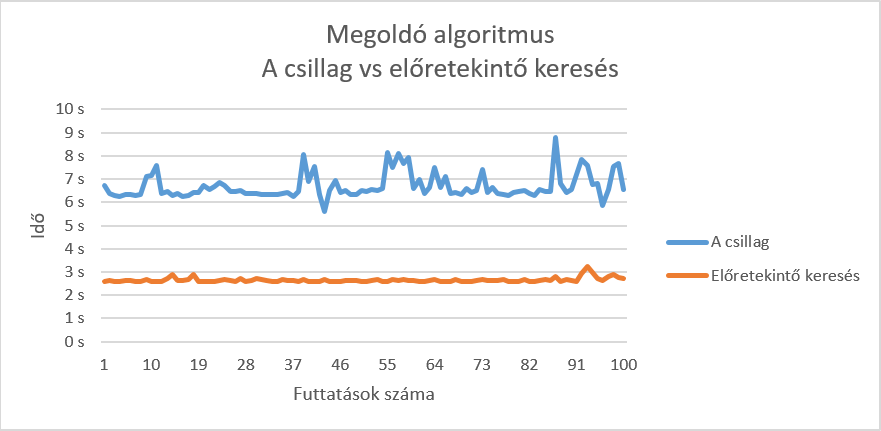
„Figyelembe veszi a tényleges távolságot a forrástól egy csúcsig, valamint a csúcs és a cél közötti becsült távolságot. A Dijkstra-algoritmussal ellentétben, amely pozitív súlyokkal, beleértve a nullát is, működik, az A\* algoritmus kizárólag szigorúan pozitív súlyokkal működik [13].”

A tesztelés során összehasonlítottam az A csillag és előretekintő algoritmus futási idejét. A teszthez Michael Fogleman által generált legnehezebb csúcsforgalmi falak nélküli feladványt (11.ábra) használtam fel.



11.ábra: Legnehezebb csúcsforgalmi feladvány falak nélkül [https://www.michaelfogleman.com/rush/]

A teszthez a két algoritmust százszor futtattam le (12.ábra) és a következő adatokat kaptam (1.táblázat):



12.ábra: A csillag és előretekintő keresés összehasonítása

1.táblázat: A csillag és előretekintő keresés futási idejének összehasonlítása

|  |  |  |  |
| --- | --- | --- | --- |
| Algoritmus | Átlag futási idő (s) | Maximum futási idő (s) | Minimum futási idő (s) |
| A csillag | 6,68 | 8,77 | 5,6 |
| Előretekintő keresés | 2,66 | 3,22 | 2,58 |

A tesztek eredménye azt mutatja, hogy az A csillag pontos megoldást nyújt, viszont futási időben háromszor annyi ideig futott átlagban, mint az előretekintő algoritmus. Az eredményekből az is kiderül, hogy az A csillag algoritmus futási időben instabil és legtöbb esetben az átlagnál tovább tart a futása. Lépésekben az A csillag a megoldástól 1-2 lépésben tér el a tényleges és legoptimálisabb megoldástól, az előretekintő algoritmus akár 20-25 lépésben is eltért.

|  |  |
| --- | --- |
| C:\Users\szabo\AppData\Local\Microsoft\Windows\INetCache\Content.Word\a csillag.png | C:\Users\szabo\AppData\Local\Microsoft\Windows\INetCache\Content.Word\eloretekinto.png |
| ***a*** | ***b*** |
| 13.ábra: Algoritmus összehasonlítás eredmények a – A csillag; b – Előretekintő keresés  Az ábrán (13.ábra) a mélység a csomópontok szintját jelenti a keresési fán, vagyis hány lépésből oldható meg a feladvány. A megnézett csomópontok száma a két algoritmus között nagyban különbözik, ezért az előretekintő keresés a kevesebb megvizsgált és kiterjesztett állapot miatt gyorsabban fut le. | |

* + 1. Vizuális tesztek
    2. Teljesítményteszt

A játék teljesítményét a Unity motor profiler statisztikáját használtam fel. Két különböző gépen teszteltem a játékot, generálás és játékmenet közben (2.táblázat).

2.táblázat: A tesztelésre használt számítógépek

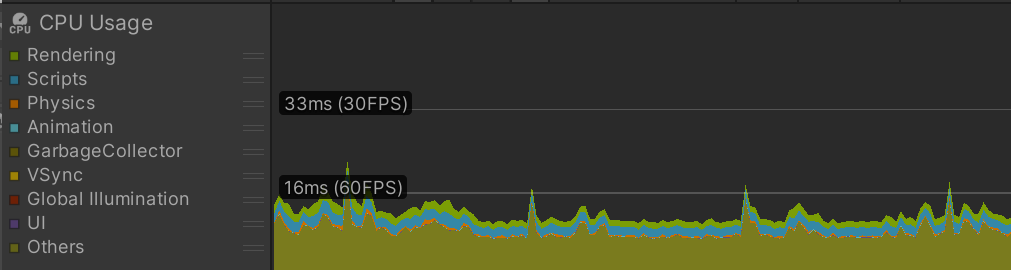
|  |  |  |  |
| --- | --- | --- | --- |
| Számítógép | Processzor | RAM | Videókártya |
| Első gép | Intel(R) Core(TM) i5-6500 CPU @ 3.20Ghz | 32 GB | Intel(R) HD Graphics 530 |
| Második gép | Intel(R) Core(TM) i5-6400 CPU @ 2.70Ghz | 16 GB | NVIDIA Geforce GTX 960 |

Az első számítógépen (14.ábra) a képkocka per másodperc játékmenet közben mindig 60 közelében volt. Generáló algoritmus futása esetén csökkent le 15 közelire.



14.ábra: Első számítógép teljesítése töltés közben

A második számítógépen (15.ábra) még a generáló algoritmus futása során sem volt érzékelhető a változás a képkocka per másodperc közel 80 volt.



15.ábra: Második számítógép eredménye töltés közben

A teljesítmény tesztek eredménye azt mutatja, hogy a játék gépigénye nagyon alacsony és más gyengébb eszközökön, mint például telefonokon is futtatható grafikai változtatások nélkül.

* 1. Továbbfejlesztési lehetőségek
     1. Platformok

A Unity motor ingyenesen ad lehetőséget más platformokra készíteni játékot. A játék alacsony hardver követelményeinek köszönhetően telefonokra és konzolra is kiadható lehet minimális változtatásokkal. A játék menüje fel van készítve más beviteli eszközökre, ezért csak a játékmeneten kell változtatni.

Telefonokon az érintés érzékeléséhez szolgáló Touch rendszer az ujjakat külön tudja érzékelni és még a következő eseményeket is:

* Érintés kezdete
* Érintés vége
* Mozgatás
* Nyomva tartás
* Eltörölt

Használat során az érintésekkel lehetne a járműveket és kamerát mozgatni.

A kontrollerel történő mozgatásra a Unity Input System alapértelmezetten tartalmazza a megfelelő beviteli függvényeket. A játékmenetben a kamera a jobb Joystick segítségével mozoghatna az Input.GetAxis függvény meghívásával az adott tengely átadásával. A járművek mozgatásához alapvetően egy jármű kerülne kijelölésre a bal Joystick segítségével, lehetne váltani a járművek között. Gomb lenyomására ki lehetne választani az adott járművet és szintén a bal Joystick segítségével mozgatni. A gomb ismételt megnyomására a mozgatás megtörténik és a mozgatások száma inkrementálódik.

* + 1. Többjátékos mód

A játék kiegészíthető lehetne egy olyan játékmóddal, ahol kettő vagy akár egyszerre több játékos is ugyanazt feladványt oldaná meg egymással versenyezve a jobb időért vagy mozgatási számért. A Unity motor a többjátékos mód létrehozására tartalmazza a Netcode for GameObjects elnevezésű bővítményt, amelyben RPC segítségével lehet függvényeknek megadni, hogy szerveren vagy kliensen fussanak le.

Az RPC célja a kommunikációt úgy feltűntetni, mint egy helyi eljáráshívást. Hibája, a lassúsága, mivel 10 ciklus ≈ 3 ns. [14]

A generáló algoritmus ServerRPC alkalmazásával futna le így garantálva, hogy minden játékos ugyanazt a feladványt oldja meg.

További bővítési lehetőség egy online ranglista bevezetése, ami a jelenlegi adatbázison nagy mértékben változtatna. A beállításokat tárolná lokálisan a játék, mivel ezek az adatok csak arra a számítógépre vonatkoznak, amelyikre mentésre kerültek.

Lokális adatbázis mellett az eredményeket és felhasználókat távoli adatbázisban kell tárolni.

* + 1. Komplex közlekedés

A jelenlegi járművek csak egyenes vonalban képesek haladni, viszont a kanyarodás is megoldható Bézier görbe segítségével.

„Az eljárást Pierre Bézier francia mérnök publikálta először 1962-ben, aki az autótervezésben alkalmazta a közelítő görbéket. Paul de Casteljau 1959-ben már kifejlesztette azt az algoritmust, amelyet a mai napig is használunk a Bézier-görbék előállítására [15]. „

A harmadfokú Bézier görbék tulajdonsága, hogy az első és utolsó pontján halad át, másik két pontja az irányító pont. A pontok között rekurzív lineáris interpoláció segítségével meghatározható egy adott t ponthoz tartozó pozíció a görbén.

P(t) = P0( -t3 + 3t2 – 3t + 1) + P1(3t3-6t2+3t) + P2(-3t3+ 3t2) + P3(t3)

A járművek a görbén közlekednének és a forgás iránya a görbe deriváltjától függene.

P'(t) = P0( -3t2 + 6t – 3) + P1(9t2-12t+3) + P2(-9t2+ 6t) + P3(3t2)

Ezt követően a jelenlegi rendszeren annyit kell változtatni, hogy előre definiált utakra kerülne a jármű és azon haladna végig a lámpáktól függően.

Összefoglalás

A szakdolgozatom célja a csúcsforgalom játék elkészítése volt érdekes és skálázható pályákkal. A feladványok generálásához mesterséges intelligencia alapú algoritmusokat használtam, mint például az A csillag algoritmus. A játék a Unity játékmotorban készült, az egyszerű használat és a felhasználói követelmények egyszerű implementálása érdekében.

A szakdolgozat megoldása megkövetelte a kutatómunkát az A csillaggal és a Unity motorral kapcsolatban. A tervezés során információkat gyűjtöttem a csúcsforgalom játék szabályairól és a megoldás egyszerűsítéséről és gyorsításáról, a brute-force módszer helyett. Több algoritmus kipróbálása után, mint például a szélességben először keresés, mélységben először keresés és az előretekintő keresés, arra az eredményre jutottam, hogy az A csillag a legmegfelelőbb a probléma megoldására a pontos eredménye miatt.

A játék egyszerűen bővíthető A Unity motornak köszönhetően, mivel több platformot is támogat, mint például az Android, IOS és konzolok. A beviteli rendszer lehetővé teszi, hogy más eszközökkel is lehessen irányítani a játékot, mint például kontrollerek vagy a telefon érintőképernyőjén keresztül. A Unityben apró változtatásokkal többjátékos módot is be lehet vezetni a játékba, online ranglistával és más játékmódokkal.

Összeségében a projekt jobban sikerült, mint arra számítottam, mert a megoldó algoritmus gyorsan és pontosan tudott lefutni és nehéz feladványokat is meg tudott oldani. A generáló algoritmus skálázható és érdekes feladványokat generált és a megoldhatóság mindig biztosított a megoldó algoritmusnak köszönhetően.

Summary

The goal of my thesis was to make a Rush Hour game with scalable and interesting puzzles. For the creation of the puzzles I used artificial intelligence type algorithms like A star. The game was made in the Unity game engine for the accessibility of the program and for to make easy updates following user demands.

The solution to the thesis involved i had to get a deeper understanding of A star and the Unity engine. The planning involved getting information about the rush hour game and how to simplify the problem so it is solvable and quicker to load than a brute-force method. After trying out multiple AI algorithms like breadth first search, depth first search and greedy best first search i came to the conclusion that A star is the best for the problem, because it had the best precision.

The game could be easily updated with Unity to other platform like Android, IOS or consoles. The input system in Unity allows for easy implementation of other input devices like controllers or it could be used with a touchscreen on phones for simple navigation through the menus or moving the vehicles. Unity also allows for a simple implementation for online multiplayer so the leaderboard could use every players results.

In conclusion the project performed better than I expected, because the solver algorithm ran very quickly and it efficiently found a good solution for the puzzle. The generation gives a scalable and interesting puzzle that is solvable each time due to the solver. Finally the presentation of the game was made better by the large community of the Unity Editor.

Ábrajegyzék

[1.ábra: Nehézség beállítások menüterv 11](#_Toc180685219)

[2.ábra: Játék vége felület terve 12](#_Toc180685220)

[3.ábra: Regisztrációs felület terve 13](#_Toc180685221)

[4.ábra: Bejelentkezés felület terve 14](#_Toc180685222)

[5.ábra: Ranglista felület terve 15](#_Toc180685223)

[6.ábra: Beállítások felület terve 15](#_Toc180685224)

[7.ábra: Generáló algoritmus folyamatábra 16](#_Toc180685225)

[8.ábra: Megoldó algoritmus folyamatábra 17](#_Toc180685226)

[9.ábra: Adatbázis terv 18](#_Toc180685227)

[10.ábra: Mappa szerkezet 36](#_Toc180685228)

[11.ábra: Legnehezebb csúcsforgalmi feladvány falak nélkül [https://www.michaelfogleman.com/rush/] 37](#_Toc180685229)

[12.ábra: A csillag és előretekintő keresés összehasonítása 37](#_Toc180685230)

[13.ábra: Első számítógép teljesítése töltés közben 38](#_Toc180685231)

[14.ábra: Második számítógép eredménye töltés közben 39](#_Toc180685232)

Irodalomjegyzék

1. Mark Overmars Game Maker Tutorial Designing Good Games  
   Elektronikus forrás:  
   <https://web.cs.wpi.edu/~imgd1001/a08/readings/Overmars_GoodGames.pdf> Megtekintés dátuma: 2024.11.03
2. Josh Brunner és mtsai 1 × 1 rush hour with fixed blocks is PSPACE-complete  
   Elektronikus forrás:  
   https://dspace.mit.edu/handle/1721.1/129837 Megtekintés dátuma: 2024.11.03
3. Gary William Flake ∗, Eric B. Baum Mathematical Games Rush Hour is PSPACE-complete, or “Why you should generously tip parking lot attendants”  
   Elektronikus forrás:  
   <https://core.ac.uk/download/pdf/82751815.pdf> Megtekintés dátuma: 2024.10.19
4. Illés Zoltán Programozás C# nyelven  
   Elektronikus forrás:  
   http://compalg.inf.elte.hu/~tony/Informatikai-Konyvtar/09-Programozas%20C-sharp%20nyelven/Programozas-Csharp-nyelven-Konyv.pdf Megtekintés dátuma: 2024.11.07
5. Jay Kreibich Using SQLite  
   Elektronikus forrás:  
   <https://books.google.hu/books?id=HFIM47wp0X0C> Megtekintés dátuma: 2024.10.19
6. John Haas A History of the Unity Game Engine  
   Elektronikus forrás:  
   <https://digital.wpi.edu/downloads/2f75r821k> Megtekintés dátuma: 2024.10.19
7. Lec. Hussein K. Jobair Chapter Three / Interpolation  
   Elektronikus forrás:  
   <https://coeng.uobaghdad.edu.iq/wp-content/uploads/sites/3/2021/11/Lec.-3-Interpolation.pdf> Megtekintés dátuma: 2024.10.19
8. Physics- Raycasting  
   Elektronikus forrás:  
   https://research.ncl.ac.uk/game/mastersdegree/gametechnologies/physicstutorials/1raycasting/Physics%20-%20Raycasting.pdf Megtekintés dátuma: 2024.11.06
9. Vugar Hacimahmud Abdullayev Alok Singh Chauhan SQL Injection Attack: Quick View  
   Elektronikus forrás:  
   https://www.researchgate.net/publication/368511340\_SQL\_Injection\_Attack\_Quick\_View Megtekintés dátuma: 2024.11.07
10. Hirotaka Yoshida Alex Biryukov Analysis of a SHA-256 Variant  
    Elektronikus forrás: https://www.researchgate.net/publication/221274799\_Analysis\_of\_a\_SHA-256\_Variant Megtekintés dátuma: 2024.11.06
11. Linda G. Hayes The Automated Testing Handbook  
    Elektronikus forrás:  
    <https://www.softwaretestpro.com/wp-content/uploads/2016/10/AutomatedTestingHandbook.pdf> Megtekintés dátuma: 2024.11.03
12. Asmo Jurvanen Automated Testing with Unity  
    Elektronikus forrás:  
    <https://www.theseus.fi/bitstream/10024/792842/3/Jurvanen_Asmo.pdf> Megtekintés dátuma: 2024.10.24
13. Daniel Monzonís Laparra PATHFINDING ALGORITHMS IN GRAPHS AND APPLICATIONS  
    Elektronikus forrás:  
    <https://diposit.ub.edu/dspace/bitstream/2445/140466/1/memoria.pdf> Megtekintés dátuma: 2024.11.03
14. Wyatt Lloyd Network Communication and Remote Procedure Calls (RPCs)  
    Elektronikus forrás:  
    <https://www.cs.princeton.edu/courses/archive/fall18/cos418/docs/L2-rpc.pdf>  
    Megtekintés dátuma: 2024.10.19
15. Dr. Kovács Emőd Komputergrafika – Matematikai alapok  
    Elektronikus forrás:  
    https://aries.ektf.hu/~hz/pdf-tamop/pdf-01/html/index.html Megtekintés dátuma: 2024.11.05

Melléklet