Eötvös Loránd Tudományegyetem

Informatikai Kar

Algoritmusok és Alkalmazásai Tanszék

DotsandBoxes és Trictactoe játékprogramok

Témavezető:

Veszprémi Anna

mesteroktató, egyetem

Szerző:

Mizsei Tamás Bálint

Programtervező Informatikus BSc.

Budapest, 2019

1 Tartalom

1. Bevezetés

Úgy gondolom, hogy csak azt az embert tudják teljesen lekötni tanulmányai, aki ténylegesen érdeklődik azok iránt. Már nincs olyan dolog, amire ha az ember rákeres, ne találna érdekes leírásokat, tudományos feljegyzéseket vagy akár útmutató videókat az interneten. Így szinte bárki képes elmélyíteni tudását a megfelelő eszközök segítségével. Amikor szakdolgozatom témáján gondolkodtam, egyből az egyik kedvenc videósorozatom jutott eszembe. A neve Numberphile és a leghíresebb nyilvános videómegosztó webhelyen található meg. Feltöltéseiknek megértése sok energiát, koncentrációt és időt igényelnek, mivel a matematika legérdekesebb részeit tárgyalják. Paradoxonok, nem megszokott (de helyes) számítási módszerek vagy akár érdekfeszítő történések a számtan történelméből is helyet kapnak repertoárjukban. Engem a sok-sok izgalmas téma közül az egyes „társasjátékok" matematikai megközelítése kötött le a legjobban. Hogyan lehet tökéletesen játszani, azaz hogyan lehet mindig a legoptimálisabb lépést tenni? Lehet-e egyáltalán? Ha igen, akkor az elég a győzelemhez? Elég a kifogástalan gondolkodás, vagy szerepet játszik a szerencse is? Ehhez hasonló kérdéseket boncolgatnak a szóban forgó felvételekben. Ezek között keresgetve egy motivált egyetemi hallgató nagyon gyorsan témára tud találni. Így határoztam el, hogy szakdolgozatomban a DotsandBoxes és Notacto néven elhíresült játékokat fogom megvalósítani, amelyeknek matematikai hátterét a fentebb említett források szolgálták. Létrehozásukat a Game Maker Studio 2 (GMS2) fejlesztői környezet segítette. Megoldandó problémáimat három nagy csoportba sorolnám: a felhasználói kezelőfelület megvalósítása, dekorálása, az egyes játékok különböző mechanizmusainak kitalálása, konstruálása, illetve a gépi játékosok gondolkodási módjainak tervezése, erősségeiknek igazságos beállítása. Eleinte a díszítést tartottam időt igénylőnek és fárasztónak, de a fejlesztést segítő program remekül könnyítette munkámat. Erről bővebben a fejlesztői dokumentációban fogok írni. Végül a mesterséges ellenfelekbe törött bele többször is a bicskám. Olyanfajta gondolkodást igényelt ezen probléma megoldása, amilyenre még nem volt szükségem eddig. Úgy érzem sokat fejlődtem több téren is és élveztem minden percét a munkának.

2 Felhasználói dokumentáció

2.1 A feladat ismertetése

Eleinte a célom az volt, hogy létrehozzak két olyan játékalkalmazást, amellyel az emberek többsége el tudja magát foglalni. Tehát nehezen megunható, többször elővehető programokat képzeltem el. Ezen gondolatom a megvalósítás végére megváltozott. Mostanra már azt mondanám, hogy ezen szoftvereket inkább tanulás, elsajátítás és gyakorlás szempontjából érdemes használni.

2.2 Célközönség

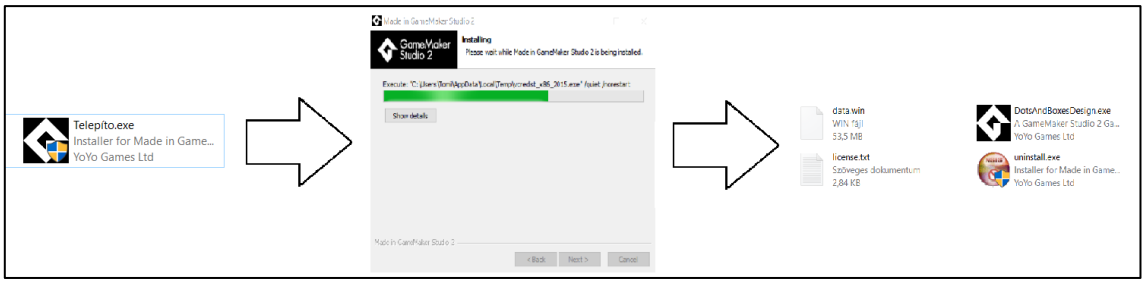
A játékszoftvereknek használatát 10 év felettieknek ajánlom. A programok segítségével megtanulhatunk elég magas szinten játszani a DotsandBoxes és Trictactoe nevű játékokkal.

2.3 A rendszer használatához szükséges/optimális szoftver, hardver környezet

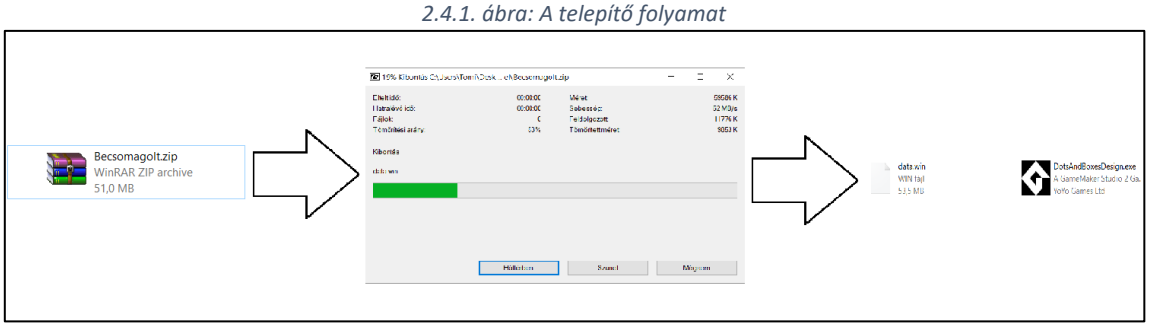
Minimum / optimális gépigények:

- **Operációs rendszer:** Microsoft Windows 7 / későbbi.
- **Videó kártya:** DirectX 10-et támogató / DirectX 11 alapú
- **RAM:** 2 GB / 8 GB

2.4 Telepítés, előkészületek



2.4.1. ábra: A telepítő folyamat

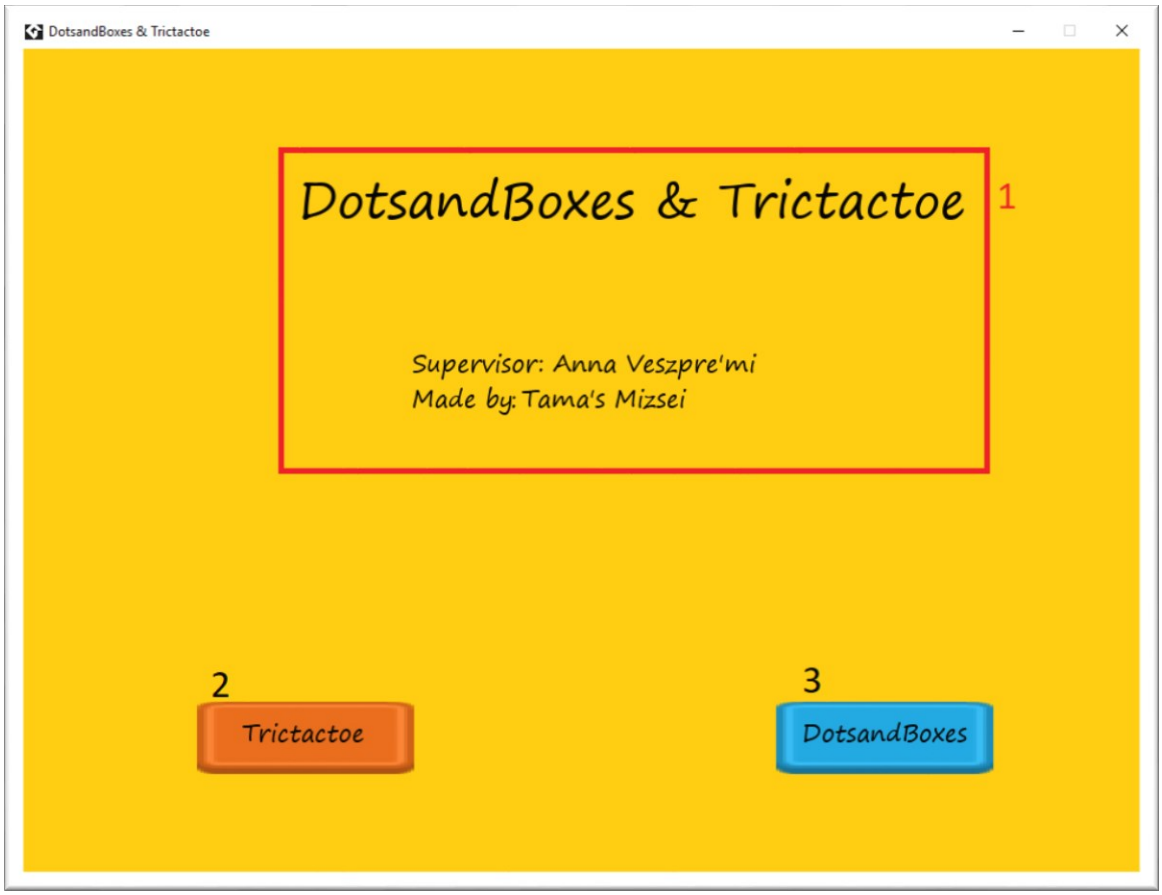


2.4.2. ábra: A kicsomagoló folyamat

A két alkalmazás igazából egy futtatható állományban van benne, így mostantól egyes számban fogom feltüntetni a programomat. A fejlesztői környezet segítségével kétféle formában lehet hozzáférni a szoftverhez: telepítővel és csomagolt állapottal. Az előbbi választással élve, az installációs állomány futtatását követően egy teljesen általános procedúra indul. Meg kell adnunk, hogy hol legyen a játék könyvtára, ezután egy kevés töltés után már megnyitható alkalmazást kapunk. A másik esetben egy becsomagolt (.zip) fájlt kell kibontanunk, viszont ekkor már nem szükséges a telepítés, mivel már azonnal egy játékra kész verziót tudhatunk magunkénak.

2.5 Indítás és kezdőképernyő

Valamelyik telepítési procedúrát befejezve több állományt láthatunk a kiválasztott mappában. Egy biztos: lesz köztük egy futtatható állomány (.exe). Ezt megnyitva már



2.6.1. ábra: A szoftver kezdőlapja

indul is a programunk. Kezdőlapként a 2.5.1-es ábrán lévő ablakot láthatjuk. Ezen képernyőn találhatók az alap információk (1-es számmal jelölt rész), illetve azok a gombok, mellyel az egyes játékokat indíthatjuk. A bal alsó részen lévő, piros színű billentyűre (2-es számmal jelölt) kattintva indul a Trictatoe (ismertebb nevén Notactoe). A másik oldalon alul, a kék színűre (3-as számmal jelölt) menve már játszhatunk is a DotsandBoxes-al.

2.6 A Trictactoe (Notactoe) játékprogram

Ugorjunk is az első program kellős közepébe. A kezdőképernyőről a megfelelő kattintással eljutunk egy újabb menübe (2.6.1.2-es kép). Ebben az ablakban már lehet hallani az alapzenét. A játék két fél között zajlik. A játékmenet körökre osztott, amelyekben az éppen soron lévő játékos bármelyik, „élő” táblán lévő, üres négyzetbe kattintva X-et rakhat. Ha egy táblán kijön az amőbából ismert három X egymás mellett

vízszintesen, függőlegesen vagy akár átlósan, akkor az adott tábla „hallottnak” számít. Ezekre már nem lehet többet X-et rakni. Célunk az, hogy az utolsó „élő” táblát ne mi „öljük” meg.

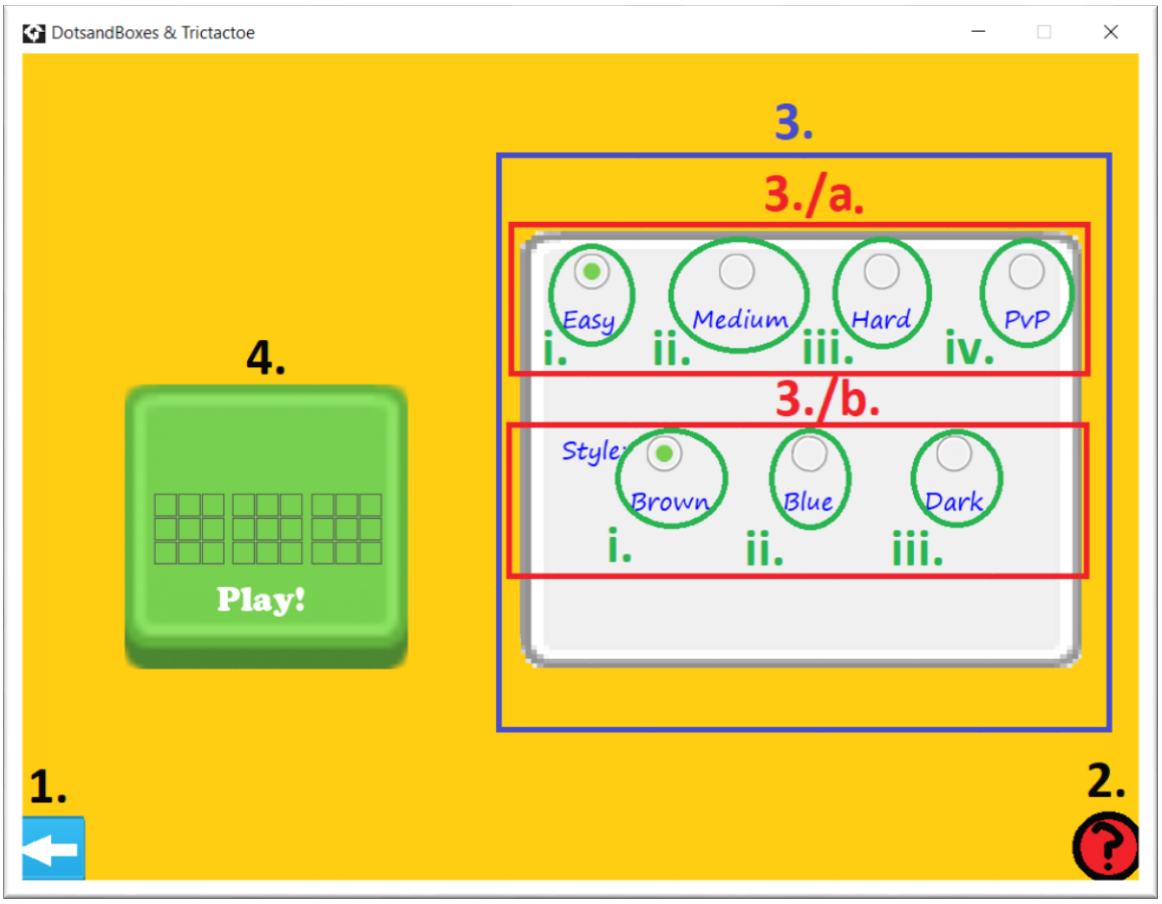
2.6.1 Főmenü

1. Visszagomb. Ennek segítségével vissza tudjuk magunkat navigálni a kezdőképernyőre.
2. Segítőgomb. Ezzel a gombbal egy útmutató ablakra ugrunk. Itt találhatjuk a játék szabályainak, mechanikáinak, céljának leírását. Bővebben erről az ablakról később olvashatnak.
3. Játékbeállító menü. Ezen részen lehet a következő játszma paramétereit testre szabni.
    a. Nehézségek és egymás elleni játszma. Valamelyik választógombra kattintva beállíthatjuk a kívánt nehézségi fokozatot vagy éppen azt, hogy egymás ellen szeretnénk játszani. Persze ezen választógombok közül egyszerre csak egy lehet aktív.
        i. Könnyű nehézségű gépi játékos.
        ii. Közepes nehézségű gépi játékos.
        iii. Nehéz gépi játékos.
        iv. Egymás elleni játékmód.
    b. Stílusok. Ezen három választógomb valamelyikére kattintva kiválaszthatjuk a nekünk legjobban tetsző színkombinációt, amelyek most különböző fantázianevekre hallgatnak. Ez azt jelenti, hogy mindhárom esetben más-más kinézetet kap mind a háttér, mind a négyzeteken belüli keresztek (tick).
        i. Barna stílus. Háttérszín barna, keresztek zöldek.
        ii. Kék stílus. Háttérszín sötétkékes, keresztek sárgák.

iii. Sötét stílus. Háttérszín sötétlila, keresztek bordók. Ezt a kinézetet arra szántam, hogy kímélje a szemet, ha valaki sötétebb környezetben játszana.

2.6.1.1. ábra: A stílusok grafikái

4. Indítógomb. Ha már mindent igény szerint konfiguráltunk a játékbeállító menüben, akkor itt az idő erre a gombra kattintani. Ekkor a játék megfelelő paraméterekkel rendelkező színterének ablakára ugrunk és kezdetét veszi a játszma.

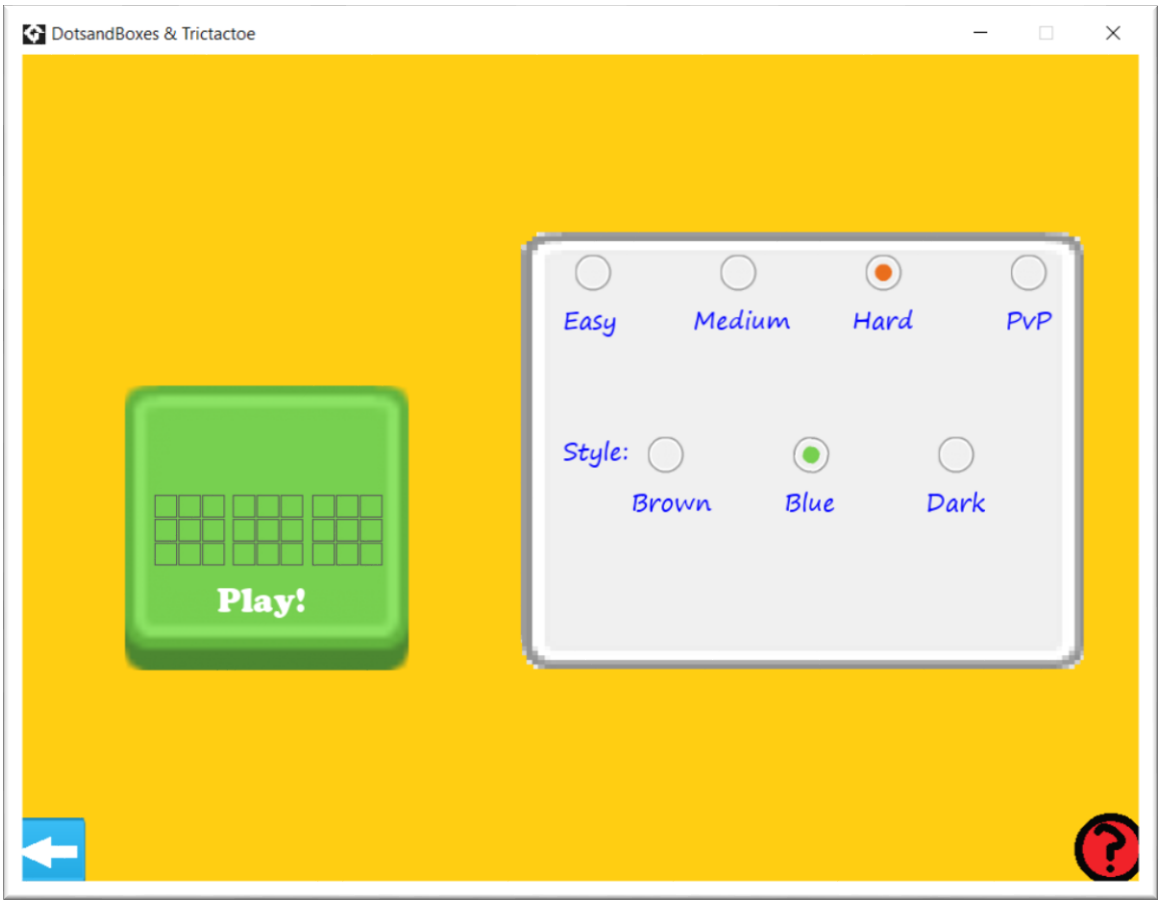


2.6.1.2. ábra: A Trictactoe menüje

2.6.2 Főmenü példa

Úgy gondolom minden jobban érthetőbb egy példával, így vegyünk egy lehetséges konfigurációt, nézzük meg a beállítás menetét, és hogy ebben az esetben hogy fest a főmenü. Tegyük fel, hogy szeretnék egy olyan játszmát indítani, ahol egy nehéz gépi játékossal szeretnék versengeni a győzelemért, és a kék stílus áll hozzám a legközelebb. Ekkor mindössze annyi a tennivalóm, hogy a nehézségi fokozatoknál rákattintok a „Hard” szöveg feletti kiválasztógombra, majd a „Blue”-hoz tartozóra. Persze a sorrend nem számít. Ezek után (ha már ismerem a szabályokat) a „Play!” gombra kattintva indul is a

játék a gépi játékos ellen. Megjegyzésképp megemlítem, hogy ha a játékos bárhonnan a főmenübe lép, akkor az alapbeállítás az a könnyű gépi játékos és a barna stílus.



2.6.2.1. ábra: A menü a megfelelő lépések után, indítás előtt

2.6.3 Szabályok, útmutató ablak

Mint ahogy a 2.3.1-es résznek a második pontjában írtam, létezik egy gyorstalpaló, segítő ablak azoknak, akik meg szeretnék tudni gyorsan a játék alapvető elemeit, célját.

1. Visszagomb. Ezzel a gombbal visszajuthatunk a játék főmenüjébe.
2. Segítő diasor. Ez az egyetlen dinamikus panel az ablakban. Bal egérgombbal tudunk előre, míg jobb egérgombbal hátra lapozni. A hat képes sorozat segít megtanítani az alapvető mechanikákat, amelyek röviden összefoglalva a következők:
    a. Egy négyzetbe X-et úgy lehet rakni, hogy egyszerűen rákattintunk.
    b. Ha valahogy kijön három X egymás mellett (ugyanazon a táblán, vízszintesen, függőlegesen vagy átlósan), akkor az a tábla

„halottnak” számít. Ez annyit jelent csupán, hogy nem lehet ezek után rá X-et tenni.

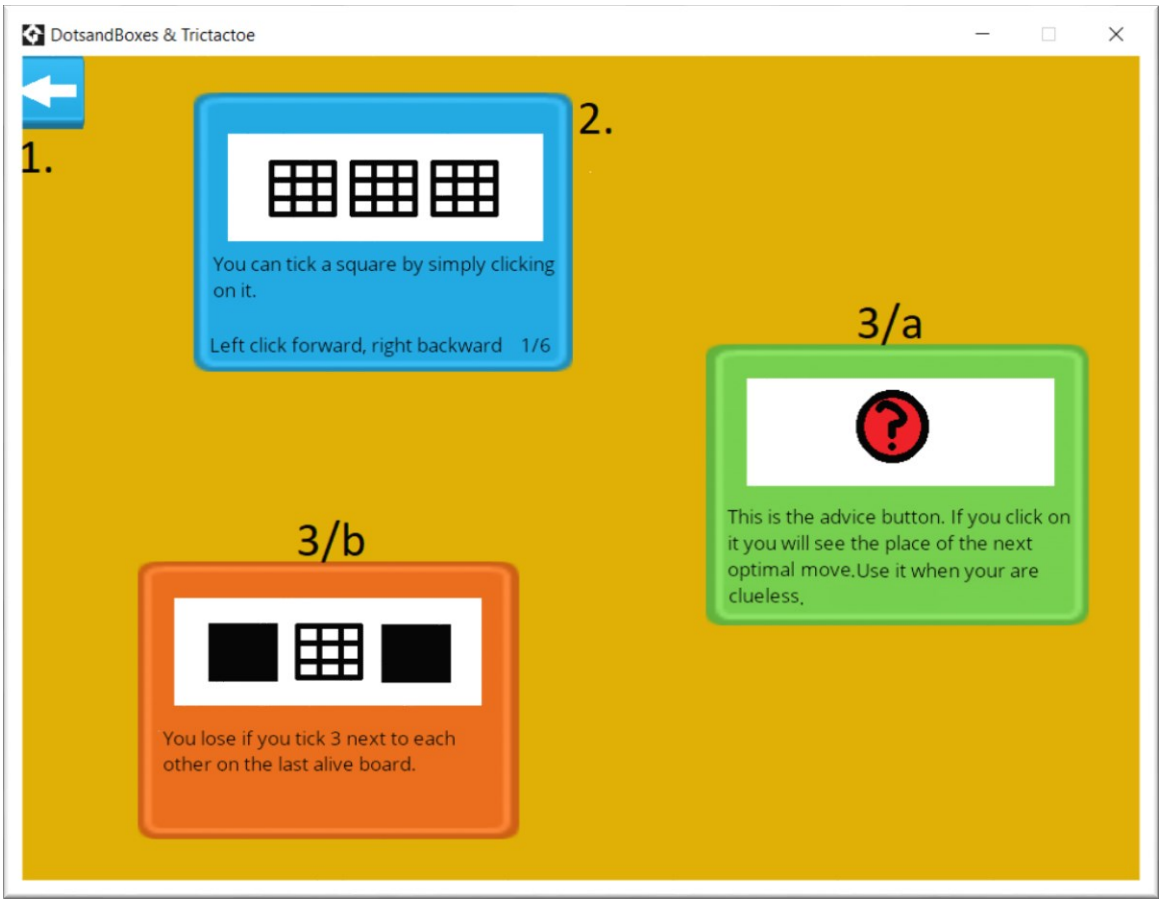
c. A játszma körökből áll. Mindenki egyszerre csak egy X-et rakhat, de azt bármelyik „élő” táblára. Minden X ugyanolyan színű. Ezzel is érzékeltetve azt, hogy egyenrangúak. Ez azért lényeges, mert az is előfordulhat, hogy három egymás melletti szimbólum közül kettő az enyém, egy az ellenfelemé, és így is „meghal” az adott tábla.

d. Fontos megjegyezni: egy tábla „megölése” nem jelent pontot az adott játékosnak.

3. Statikus panelek. A két panel további tájékoztatásokkal lát el bennünket a játékról.

   a. Felhívja a figyelmünket a tanácsgombra. Ez a billentyű láthatóvá teszi azt, hogy a gép hova tenne a helyünkben. Javasolt akkor használni, ha nem tudjuk eldönteni a megfelelő lépés helyét. Ezt a játéktéren egy feketén villogó X jelzi majd. A játékosnak nem muszáj odalépnie, ez csak egy tipp.

   b. Leírja a játék célját, lényegét. A játszmát az a játékos veszti el, aki „megöli” az utolsó táblát. Azaz a célunk az, hogy ne nekünk jöjjön ki három X egymás mellett az utolsó táblán.



2.6.3.1. ábra: Az útmutató ablak

2.6.4 A játéktér

Nos, már csak egy ablakkal kell megismerkednünk, ez pedig nem más, mint a játéktér ablaka. Erre a helyre a főmenüből, az indítógomb kattintásával juthatunk el. Itt már egy nyugodtabb zene szól a háttérben, ami kicsit jobban kedvez a gondolkodásnak. Ez a panel szinte csak dinamikus, interaktív részekből áll.

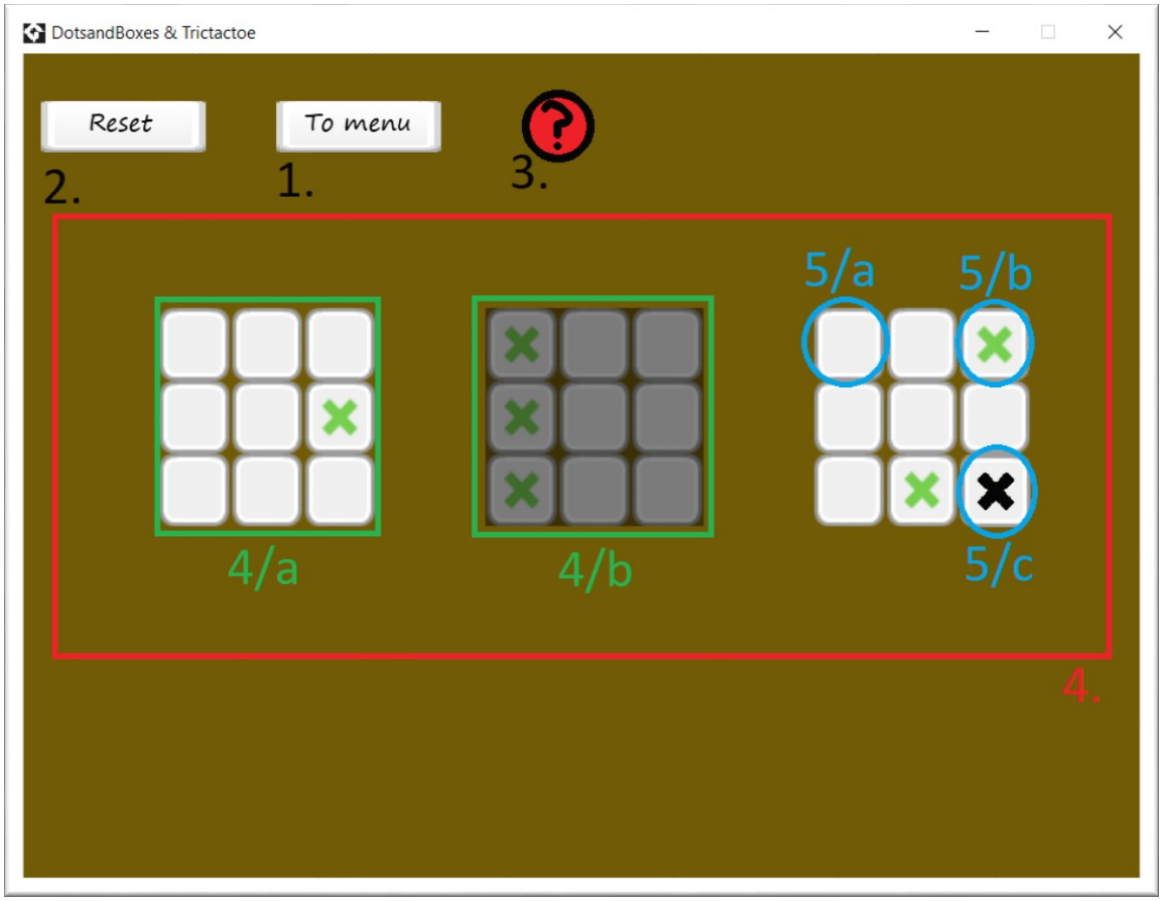
1. Visszagomb. Ennek segítségével vissza tudunk lépni a játék főmenüjébe.
2. Újraindító gomb. Ha erre a gombra kattintunk, akkor újrakezdjük a játékunkat. Fontos, hogy ekkor minden paraméter, amit indítás előtt beállítottunk, megmarad. Csupán a játéktérből távolítja el a szimbólumokat, illetve eltávolítja a halott táblákat.
3. Tanácsgomb. Ezt a gombot már a 2.3.3-as rész 3/a részpontjában kifejtettem.
4. Játéktér. Három darab 3x3-as négyzet a játék színtere.
    a. „Élő” tábla. Ezen táblákra még lehet X-et rakni.
    b. „Halott” tábla. Ezekre a táblákra már nem lehet X-et rakni.

5. Négyzetek. A táblák kilenc darab kis négyzetből állnak.

    a. Lehet a négyzet üres.

    b. Vagy lehet benne X.

    c. Tanács jelzője. Ez az X mindig fekete színű, stílustól függetlenül. Akkor jelenik meg és kezd el pislákolni, ha a játékos előzetesen a tanács gombot nyomta meg. Fontos megjegyezni, hogy ha egy adott helyzetnél egymás után többször is tanácsot kérünk, akkor előfordulhat, hogy a program más-más mezőt jelöl meg. Ennek az a magyarázata, hogy a gép ezeket a helyeket épp ugyanolyan jónak (netalántán rossznak) ítél meg, így teljesen mindegy melyikre rakunk.



2.6.4.1. ábra: A Trictactoe játéktere

Ha gépi ellenfél ellen játszunk, akkor a játék végét követően az újraindító gomb és a visszagomb alatt láthatjuk a „Nyertél” vagy „Vesztettél” szöveget. Így már teljesen készen állunk a játéktudásunk mesteri szintre emelésére a program segítségével. Nézzük tovább a másik alkalmazást.

2.7 A DotsandBoxes játékprogram

Mindkét játék szerkezete hasonló, így bemutatásuk is az lesz. A kezdőablakban a jobb alsó gombra kattintva tudunk eljutni a DotsandBoxes nevű játék főmenüjébe. Ekkor éppúgy, mint a másik játéknál is elindul ugyanaz az alapzene. Ezen játék is két fél között zajlik. Körünkben két (vízszintesen vagy függőlegesen) egymás mellett lévő pontot köthetünk össze egy vonallal. Pontot úgy tudunk szerezni, ha mi húzzuk be egy 1x1-es, vonalak által határolt négyzetnek (box) az utolsó oldalát. Ebben az esetben még egyszer mi jöhetünk. Célunk a játék végére több pontot szerezni, mint ellenfelünk.

2.7.1 Főmenü

1. Visszagomb. Ezzel a gombbal vissza tudjuk magunkat navigálni a kezdőablakhoz.
2. Segítőgomb. Erre kattintva egy útmutatóablakba jutunk, ahol megtudhatjuk a játék alapvető mechanikáit, szabályait és célját.
3. Játékbeállító menü. Hasonlóan az előző játékhoz, itt is beállíthatjuk a következő játékunk paramétereit. Vegyük észre, hogy itt egy picit több lehetőségünk van. Beállíthatjuk, ki legyen a kezdőjátékos, mi vagy a gép, illetve megadhatjuk játékterünk méretét. Az előbbi lehetőség kizárási okát a Trictactoe-ból majd később tárgyalom a fejlesztői dokumentációban. Ezen menü is ugyanúgy van megoldva, mint a 2.3.1 részben, kivéve a kezdési lehetőség kiválasztását. Ez a gomb nem választógomb, csak egy jelölőnégyzet. Annyiban különböznek egymástól, hogy míg például egy nehézségi fokozatot biztosan ki kell választanunk, a kezdési lehetőséget nyugodtan üresen hagyhatjuk. Ez azt fogja jelenteni, hogy nem mi fogunk kezdeni.
    a. Nehézségek beállítása
        i. Könnyű nehézségű gépi játékos
        ii. Közepes nehézségű gépi játékos
        iii. Nehéz gépi játékos
        iv. Egymás elleni játékmód
    b. Kezdési lehetőség.

c. Pálya mérete. Egy kisebb (4x4) és egy nagyobb (5x5) pálya közül lehet választani. Az utóbbit a már gyakorlottabb játékosoknak ajánlom, mert sokkal nehezebb rajta észrevenni a helyes lépés helyét.

   i. 4x4 pontból álló játéktér.

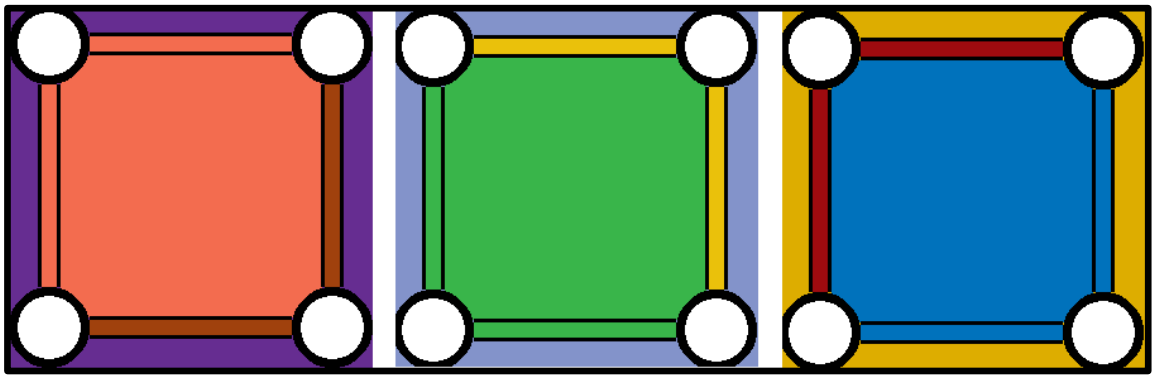
   ii. 5x5 pontból álló játéktér.

4. Stílusok. Itt is három darab stílusból lehet választani. Ez azt jelenti, hogy a játék indítása után mind a háttér, mind a különböző játékosok vonalainak színe más és más lesz. Válasszuk a nekünk legjobban tetszőt.

   a. Sárga stílus. Mustárszínű háttér. Kék, illetve bordó vonalak.

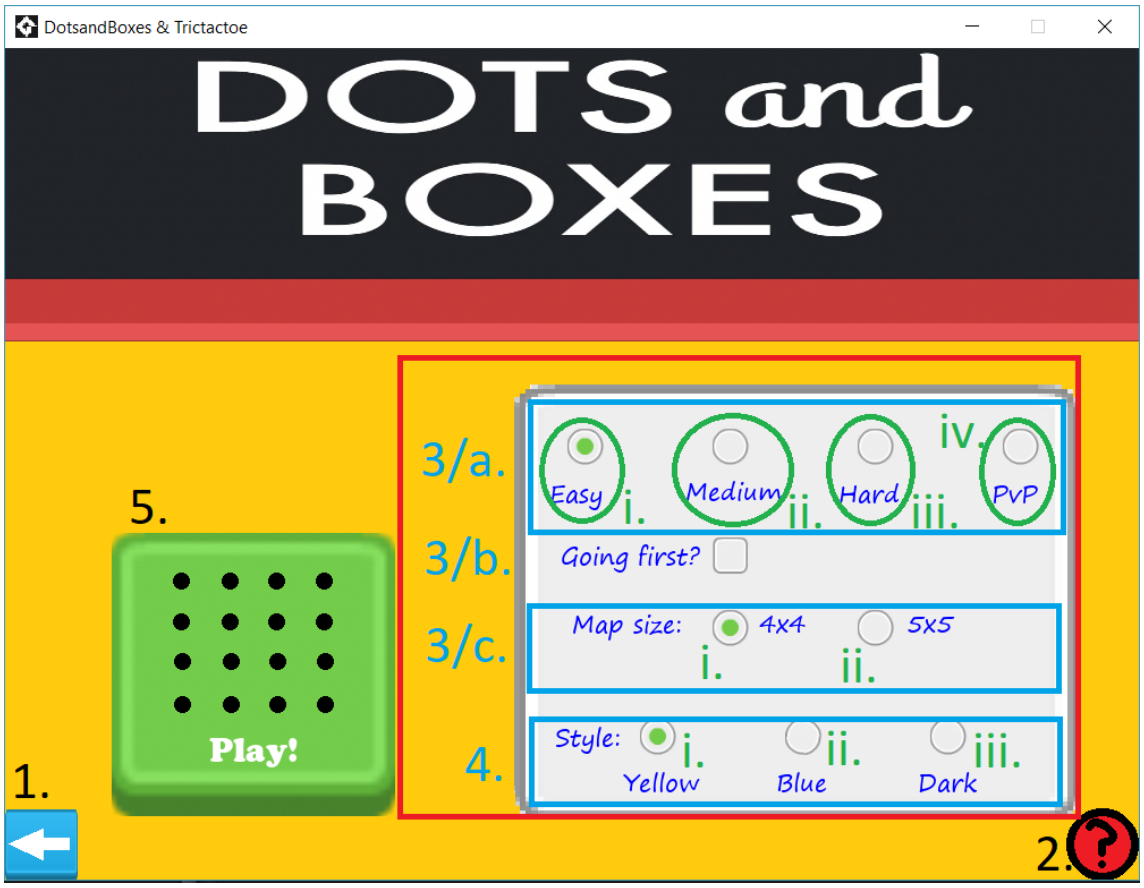
   b. Kék stílus. Halványkék háttér. Zöld, illetve sárga vonalak.

   c. Sötét stílus. Sötétlila háttér. Halványrózsaszín, illetve barna vonalak.



2.7.1.1. ábra: A sötét, kék és sárga stílusok.

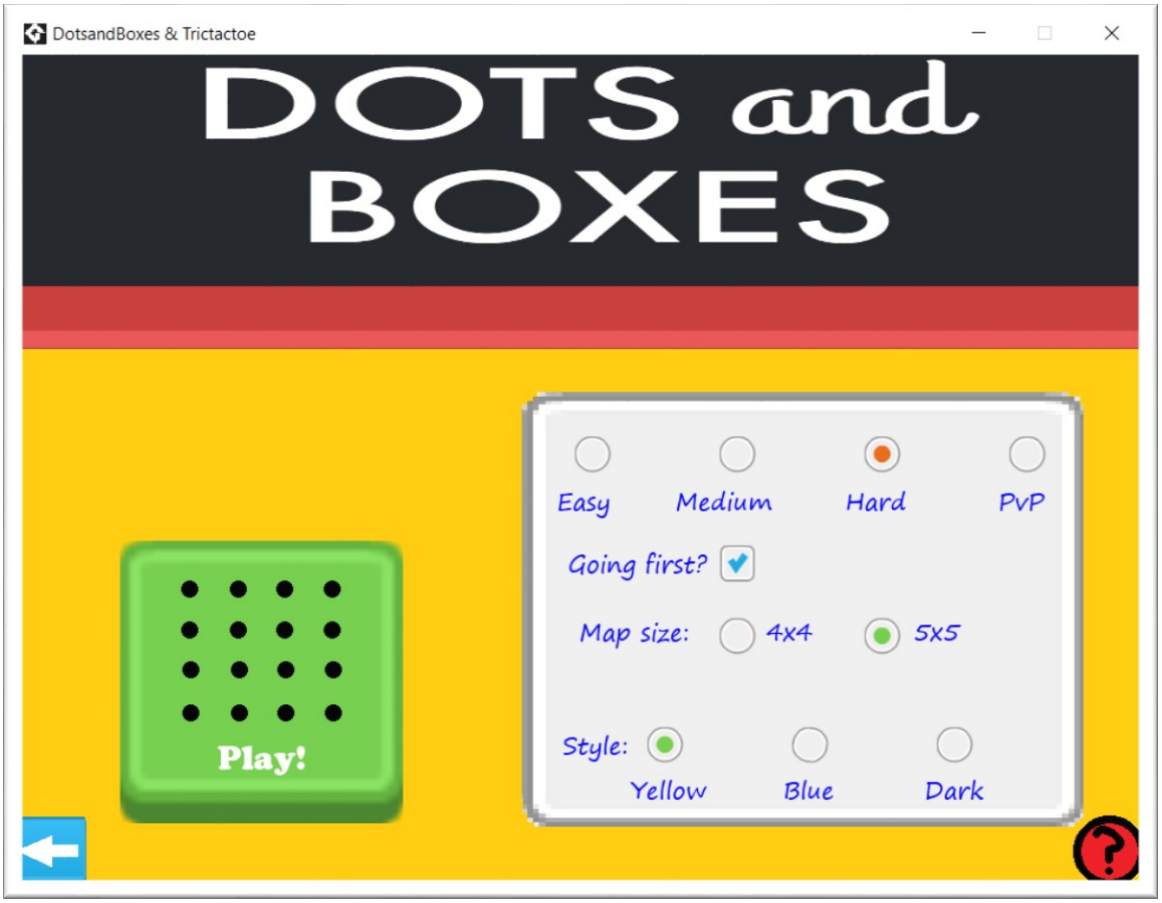
5. Indítógomb. Erre a gombra akkor menjünk rá, ha már mindent beállítottunk a játékkal kapcsolatosan, amit szerettünk volna. Ezzel átkerülünk a játéktér ablakába, ami a konfigurált paraméterek szerint néz ki és működik.



2.7.1.2. ábra: A DotsandBoxes főmenüje.

2.7.2 Főmenü példa

Nézzünk itt is egy példát a beállító ablak működésére. Mondjuk, hogy a legszívósabb ellenfél ellen szeretnék játszani úgy, hogy én kezdek. Mivel már magabiztos vagyok abban, amit csinálok, így a nagy pályán szeretném megmérettetni tudásomat. Stílusban pedig a sárga kinézet áll hozzám a legközelebb. Ekkor a következő képpen néz ki a főmenüm.



2.7.2.1. ábra: A főmenü a megfelelő lépések után

Ezután a „Play!” gombra kattintva kezdődhet a játék.

2.7.3 Szabályok, útmutató ablak

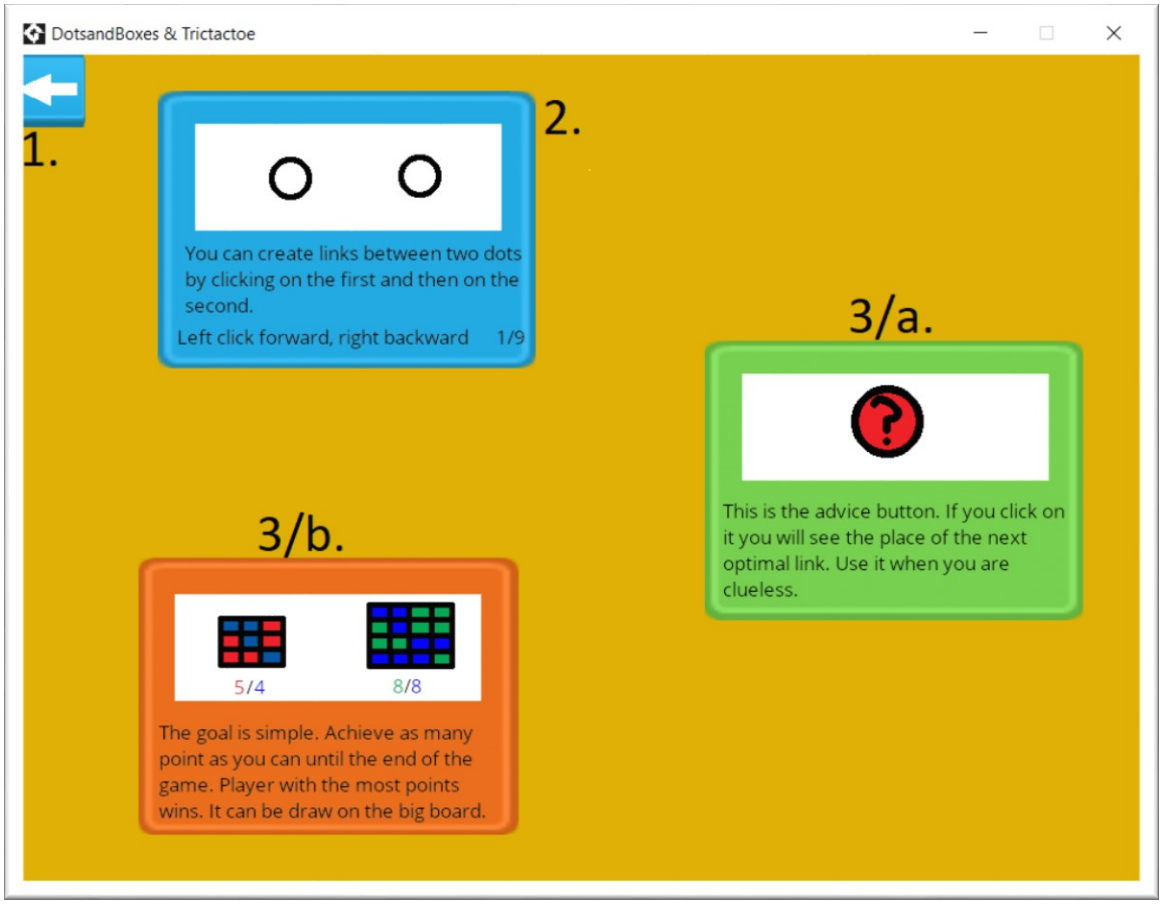
Hasonlóan működik, mint azt az első játéknál ismertettem. Egy külön ablak mesél nekünk a legfontosabb szabályokról, lehetőségekről. Ide a főmenüből a segítőgombbal tudjuk magunkat elnavigálni. A szerkezete is ismerős lehet.

1. Visszagomb. Rákattintva máris a játék főmenüjében találhatjuk magunkat.
2. Kilenc részes gyorstalpaló. Ez a kis diasorozat előre ugrik egy képpel, ha bal egérgombot nyomunk rá, és vissza, ha jobbat. A legfontosabb játékbeli elemeket tárgyalja. Röviden összefoglalva:
    a. Két, vízszintesen vagy függőlegesen egymás melletti pont közé úgy tudunk vonalat húzni, hogy először rákattintunk az elsőre, majd a

másodikra. Ha első választásunkat vissza szeretnénk vonni, csak egyszerűen kattintsunk a háttérre vagy a kijelölt pontra.

b. Ha befejezünk egy négyzetet (csúcsai a pontok, minden oldala egy hosszú vonal), akkor pontot kapunk és újra mi jöhetünk. Az mindegy, hogy a befejezett négyzet oldalai melyik játékoshoz tartoznak. Például fennállhat az az eset, hogy egy négyzet három oldala az ellenfelemé, viszont én fejezem be a negyedik oldallal és viszem el a pontot, majd jövök újra.

c. A játéknak akkor van vége, ha minden pontot elvitt valaki. Ez a kis pályán kilenc pontot jelent, a nagyon tizenhatot. Így az is érdekes, hogy az utóbbi esetben képesek vagyunk döntetlent játszani, míg az előbbiben nem. Az a játékos nyer, akinek több pontja lesz a végére.



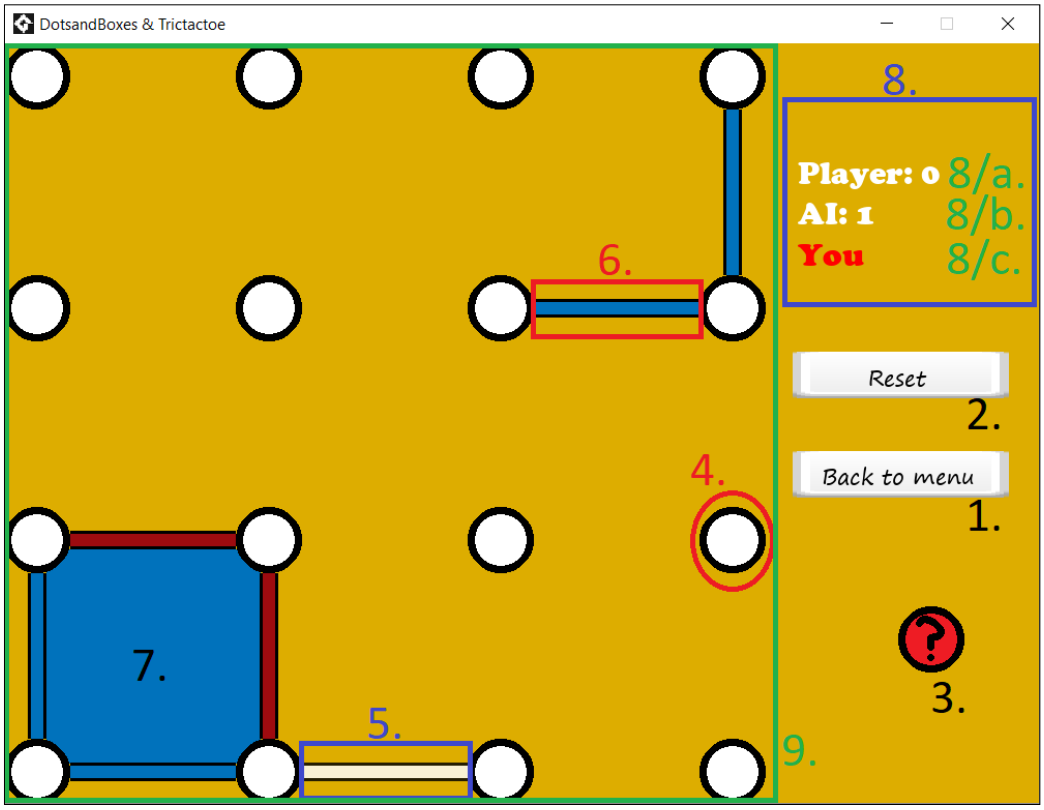
2.7.3.1. ábra: Az útmutató ablak

2.7.4 A játéktér

Már csak a játék színtere maradt az ablakok bemutatásából hátra. Erre a panelra a játék főmenüjében, a megfelelő beállítások után, az indítógombbal juthatunk el.

1. Visszagomb. Ezzel a gombbal visszajuthatunk a főmenübe.
2. Újraindító gomb. Rákattintva még egyszer nekikezdhetünk a játszmának az adott paraméterekkel. Kitörli a pontszámokat, vonalakat.
3. Tanácsgomb. Ennek segítségével kikérhetjük a gép tippjét, azaz hogy hova lenne a legészszerűbb rakni vonalat. Itt is előfordulhat, hogy többszöri rákkantintás esetén más-más tippet kapunk a programtól. Ez azért van, mert ilyenkor mindegyiket ugyanolyan jónak, vagy éppen rossznak ítéli a tanácsoló, így mindegy melyik pontpárt kötjük össze.
4. Pont. Ezekből áll a játékterünk nagy része. Függőlegesen vagy vízszintesen egymás mellett lévők közé lehet csak vonalat húzni.
5. Tanácsjelző vonal. A tanácsgombra kattintva megjelenik a program javaslata: fehér színnel villogtatja a javasolt lépést. Természetesen nem muszáj ezt választanunk, léphetünk kedvünk szerint.
6. Kék játékos által behúzott vonal. Az ilyenek által kialakított 1x1 oldalas négyzetek érhetnek játékbeli pontot.
7. Játékbeli pont. Az ilyenek gyűjtése a játszma célja. Színe mutatja, hogy melyik játékoshoz tartozik.
8. Eredményjelző panel.
    a. Emberi játékos pontszáma.
    b. Gépi játékos pontszáma.
    c. Az éppen soron következő játékost mutatja. Emberit piros „You”, míg a gépit kék „AI” felirat különbözteti meg

9. Játéktér.



2.7.4.1. ábra: A játéktér ablaka.

3 Fejlesztői dokumentáció

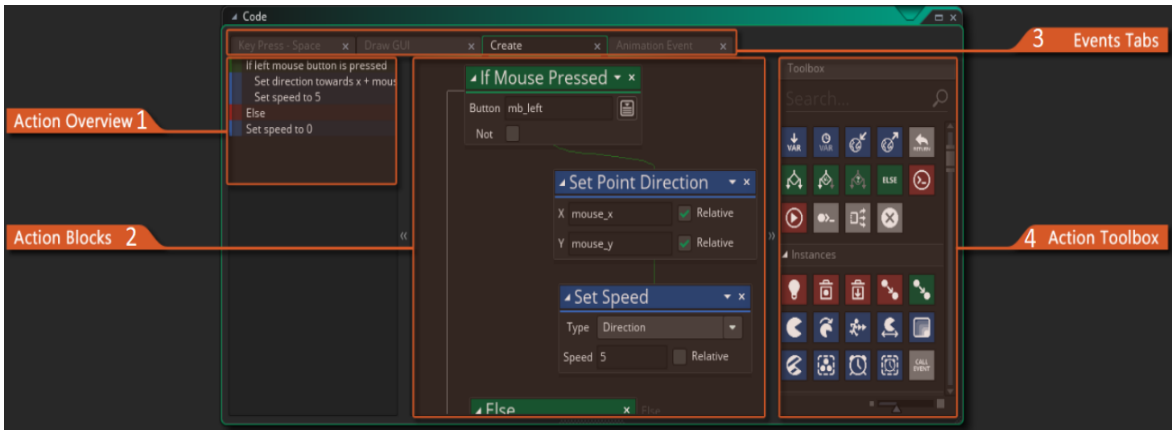
Most már belemehetünk mélyebben a programok működésének tárgyalásába, a függöny háta mögötti mozgatórugók megfigyelésébe. Mivel nem szokványos fejlesztői környezetek egyikét használtam a megvalósításhoz, így előbb rövid áttekintést szeretnék adni a fejlesztéshez használt Game Maker Studio 2 játékfejlesztő környezetről.

3.1 A fejlesztői környezet – Game Maker Studio 2

Ahogy már a bevezetésben említettem, ez a szoftver nagyban támogatott a tervezés és megvalósítás során felmerült problémák megoldásában. Amint a nevében is benne van, pont ezen nehézségek kiküszöbölésére született. Mivel egyre népszerűbbek lettek a kiadófüggetlen játékok (angolul: indie – independent video game), így nagy lett a kereslet egy nagyon könnyen érthető, akár kódolást nem feltétlen használó fejlesztői környezetre. Eddig még akár egy komolytalanabb játékszoftver megtervezése, megvalósítása és kiadása akár 10-50 ember 2-3 éves kemény munkájába került. Most ezen program segítségével akárki belefoghat saját ötletének megvalósításába, akár egyetlen kódsor leírása nélkül is hozhatunk létre játékprogramot.

3.1.1 Projekt indítása

A felhasználói felülettel kétféle projektet hozhatunk létre: kódot is engedélyező-, vagy kód nélkülit. Az utóbbi esetben egy varázslószerű programmal találhatjuk szembe

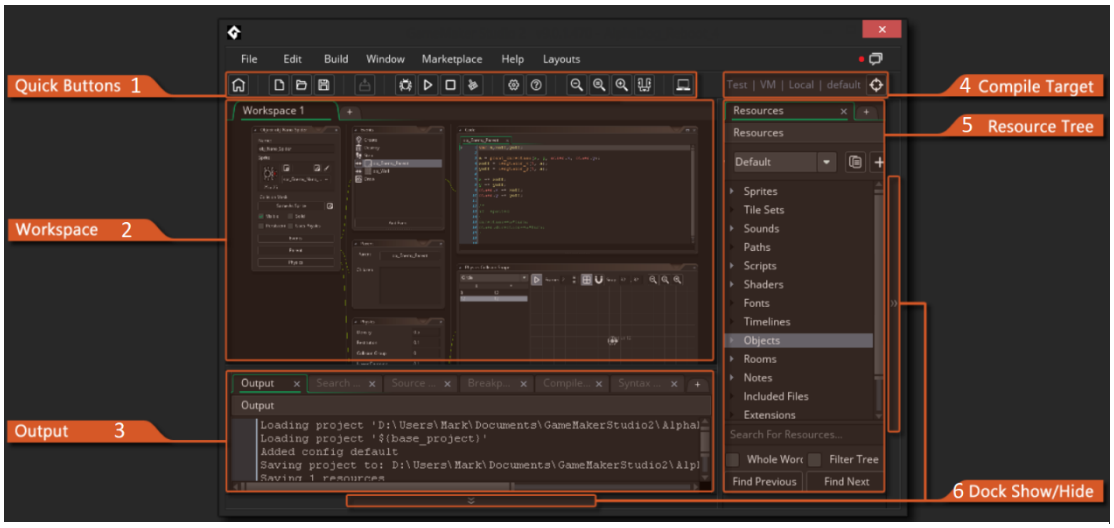


3.1.1.1. ábra: Drag and drop projekt felhasználói felülete

magunkat. Ezt a szoftver kézikönyve csak drag and drop (DnD), azaz „húzd és engedd” rendszernek nevezi. A név onnan ered, hogy a projektben a kódok helyét szimbólumok helyettesítik. Ezen jeleket a kívánt helyre húzva, majd egy-két paraméterét beállítva meg is kapjuk a kódunkat úgy, hogy nem is tudunk programozni. Persze ez a módszer nagyban korlátozza lehetőségeinket, és sok helyen szinte lehetetlenné teszi kigondolt ötletünk megvalósítását. Programozók számára ez a felület nehézkesnek és bonyolultnak tűnhet. Persze ez nem is nekik (nekünk) készült, hanem azoknak, akik nem kívánnak egy programozási nyelvet megismerni és nem érzik magukat képesnek a kódírásra. A felhasználói felület ebben az esetben négy fő részből áll.

1. Összefoglaló nézet az eddigi munkánkról.
2. Munkafelület. Ide kell a szimbólumokat behúzni.
3. Események. Ezekről majd később írok.
4. Szimbólumok. Szinte minden programozási eszköznek itt az előre megvalósított változata. Például találhatunk egy jelet, amin egy „if” felirat van. Ha ezt behúzzuk a munkafelületre, majd megadjuk, hogy mi legyen a feltétel, akkor már kész is a kétirányú elágazásunk.

A másik fajta projektben a fejlesztői környezet saját kódnyelvével, a Game Maker Language-el (GML) lehet programozni. Ennek a felhasználói felülete már ismerős lehet egy programozónak, mert ez hasonlít a legtöbb ilyen szoftver kinézetére. Ekkor szimbólumok és paraméterek beírogatása helyett csak kódolnunk kell.



3.1.1.2. ábra: A GML projekt felülete

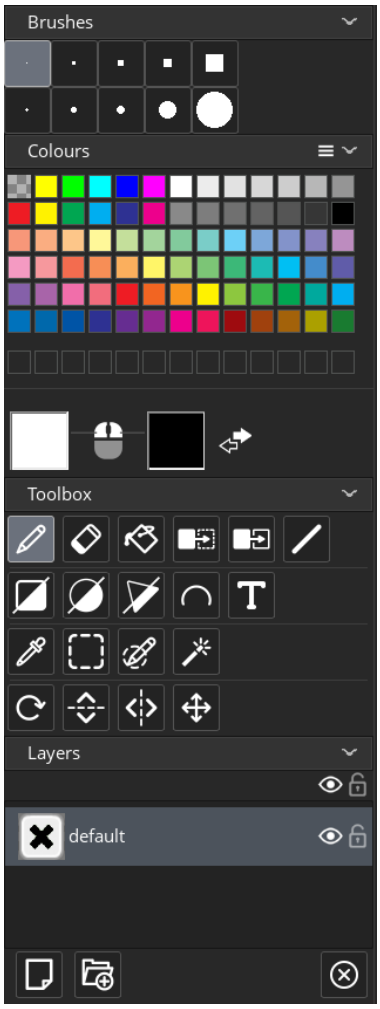
1. Gyorsgombok. Például: fordítás, futtatás, keresés stb…
2. Munkafelület. Itt láthatjuk a létrehozott objektumaink, szobáink kódjait, illetve textúráinkat is itt szerkeszthetjük.
3. Kimenet. Itt jelennek meg a debugger segítőszövegei vagy éppen a fordítási hiba.
4. Fordítási cél. Ki lehet választani, hogy éppen milyen platformra készítjük a programot. Lehetőségünk van Windowsra, Ubuntura, de akár különböző játékkonzolokra is szoftvert készíteni, mint például PlayStation 4, Xbox One.
5. Nyersanyagaink. Ide tudjuk létrehozni objektumainkat, textúráinkat, szkriptjeinket, szobáinkat, betűtípusainkat és hangjainkat. Ezen kívül lehetőségünk van importálni. Akár saját gépről, akár a fejlesztői környezet által ajánlott közösségi kiegészítőkből. Ezek lehetnek ingyenesek, de legtöbbször pénzbe kerülnek. Itt különböző kidolgozott textúra csomagok találhatók, de akadnak lekódolt, szkriptbe írt algoritmusok is.
6. Eltakaró/Előhozó gombok. Ezekkel tudunk néha több helyet adni egy panelnak úgy, hogy egy másikat becsukunk.

3.1.2 Egy projekt építőelemei

Mint ahogy már leírtam, egy projektnek vannak nyersanyagai. Fontosnak tartom ezek megismertetését, mivel egész munkánk ezekből épül fel. Persze teljes részletességgel nem fogok írni róluk, mert nagyon sok funkciójuk elég specifikus, így csak azokról teszek említést, amit használtam is munkám során.

3.1.2.1 Sprites

Ezeket az elemeket mostantól textúráknak is fogom nevezni. Általában egy objektum vizuális megjelenítésére szolgálnak, bár sokkal több funkciójuk is van. Egy ilyen elem egy vagy több darab képből álló sorozat. Ha az utóbbi eset áll fenn, akkor lehetőségünk van ezeket a képkockákat egymás után lejátszani. Így tudunk animációt megjeleníteni a textúrák segítségével. Ezen képsorozat elemei meg vannak számozva 0-tól. Hivatkozni rájuk az *image_index* beépített változóval tudunk. Munkámban ezt elsősorban a gomboknál használtam ki. Első sorszámú képnek az alapot, míg másodiknak a lenyomott változatott állítottam be. Így amikor egy gomb objektum találkozik az egérrel, akkor csak az *image_index = 1* kóddal átállítom kinézetüket. Majd ha már nincs rajta a kurzor, akkor visszaállítom. Fontos megjegyezni, hogy textúrákat nem muszáj objektumhoz kapcsolni, ki lehet őket egyszerűen csak rajzolni egy szoba valamelyik területére, vagy a szoba háttérképeként. Így nem lesz funkcionalitásuk, csak díszítőelemekként szolgálnak. Ilyenek például a segítő ablakokban a statikus panelek.

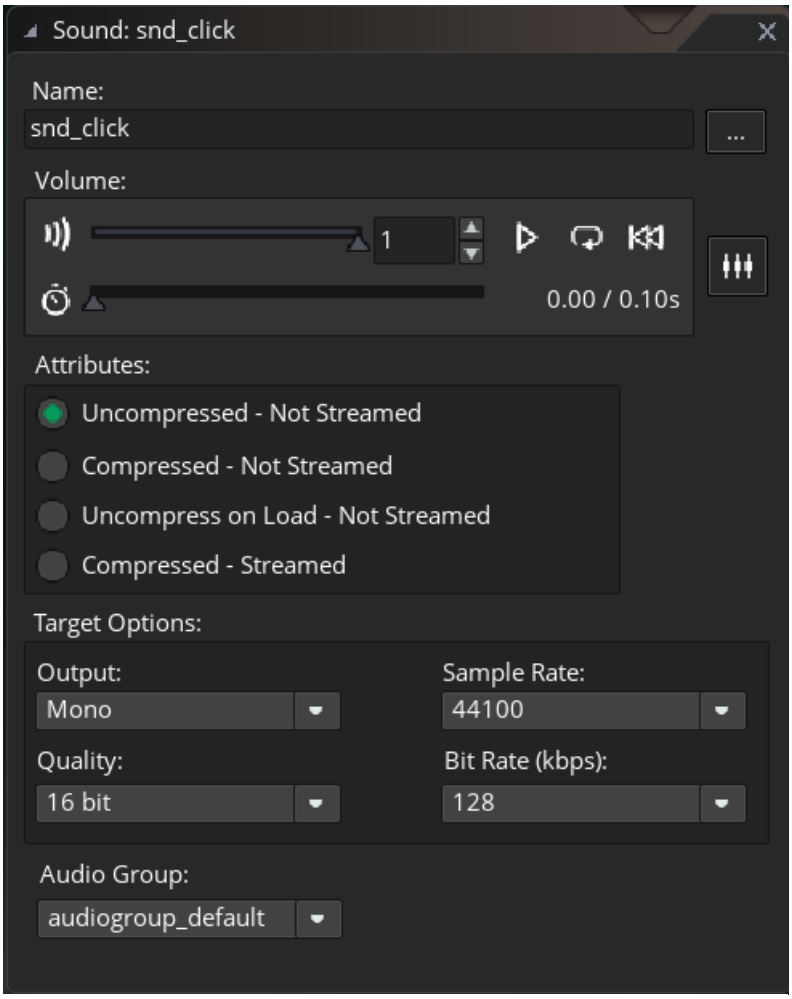


3.1.2.1.1. ábra: A textúraszerkesztő

A textúrákat saját gépünkről (.png formátumban) be tudjuk húzni, vagy akár mi is tudunk újakat létrehozni a *sprite editor* segítségével. Ebben a szerkesztőben a legáltalánosabb eszközök állnak rendelkezéseinkre: rajzolás, kitöltés, vonalhúzás, színváltás, tükrözés, rétegek kezelése stb. Ezekkel szinte bármit meg lehet valósítani egy kis kreativitással.

3.1.2.2 Sounds

A hangokkal nagyon keveset foglalkoztam munkám során. Úgy gondoltam, hogy egy logikai játékhoz elengedhetetlen egy kellemes zene, illetve a kattintások és egér-ráhúzás hangok nélkül elég üresnek tűnik az alkalmazás, így végül egy pici keresgélés után hanghatásokkal tudtam javítani szoftverem minőségén. Így visszagondolva nem volt nehéz: a lejátszani kívánt mp3 formátumú állományt be kellett húzni a hangokhoz. Ekkor megjelent előttem a hangszerkesztő. Itt olyan általános dolgokat lehet beállítani, mint például a hangerő. Ezek után már képesek is vagyunk a hang lejátszására a megfelelő eseménykor, például ha rákattintunk egy gombra. Ezt a beépített *audio_play_sound()* függvénnyel érhetjük el. Később, a megvalósításnál részletezem a beépített függvények használatát.



3.1.2.2.1. ábra: A hangszerkesztő

3.1.2.3 Scripts

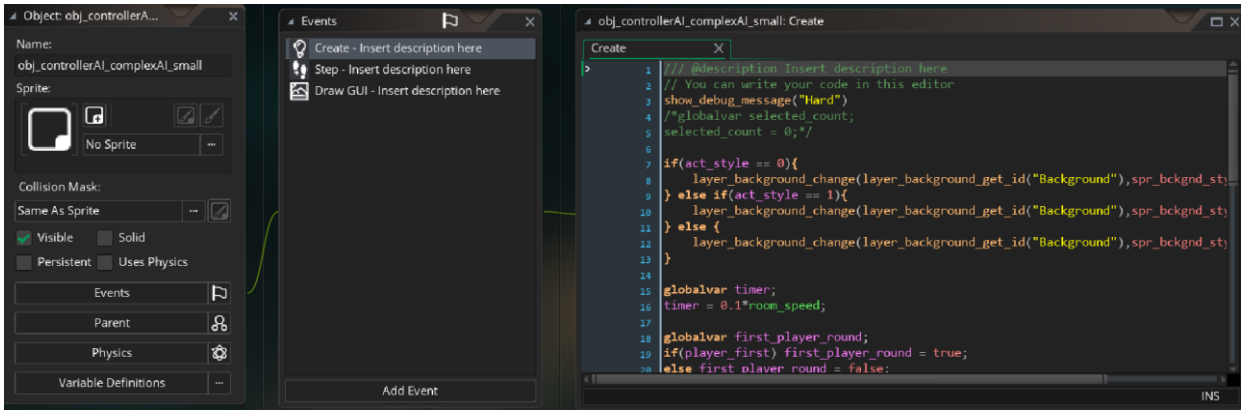
A szkriptek kódrészletek, amelyek lehetővé teszik a kódismétlés problémájának kiküszöbölését. Ugyanúgy működnek, mint más programozási nyelvekben a függvények, csak ezek mindig publikusak. Argumentumai számát nem tudjuk megadni, hanem a kódban módunkban áll használni az *argument0,..., argument9* beépített változókat. Amennyiben ezek közül hármat használtunk, akkor a fordítóprogram egyből háromargumentumos függvénynek tekinti, így hibát dob, ha kevesebb vagy több argumentummal használjuk. Visszatérési értékünk és a bemenetek típusait nem kell megadnunk előre. Rettenetesen erős eszköz, ha kódunk újra felhasználásán gondolkodunk. A legtöbbet a Trictactoe játékban, a táblák szimbólumaik megállapításához használtam ezeket.

3.1.2.4 *Fonts*

Külön elemfajta a betűtípus. Hasonlóan az előbbiekhez, saját magunk is létre tudunk hozni akár alap, akár különleges karaktertípusokat, vagy egyszerűen behúzhatunk olyanokat, amelyek már telepítve vannak a gépünkre. Mind a játékok főmenüjében, mind a játékterekben más-más típusban vannak írva a szövegek. A rajzoló eseménynél lesz nagyobb szerepük.

3.1.2.5 *Objects*

Előre felhívom a figyelmet arra, hogy ez nem a szokásos objektum orientált programnyelvekben megismert példány, hanem ez inkább az egyedeket leíró osztály. Ezeket úgy lehet felfogni, mint tervrajzokat, amelyekből később tudjuk gyártani példányainkat, amiket most a Game Maker Studio 2 esetében *instance*-nak fogunk nevezni. Minden objektumhoz hozzá tudunk társítani egy textúrát, de nem muszáj. Sőt, a legfontosabb, irányítóosztályainknak nem is szeretnénk textúrát, mivel ők majd a háttérben irányítanak gépi játékosokat, körök váltakozását vagy akár a stílusok beállítását. Legérdekesebb elem ebben a témakörben az események, angolul *event-ek*. Egy objektumnak nagyon sok eseménye lehet. Ezek igazából kódok helyei, amik egy adott történéskor futnak le. Például a *create* eseményben, tehát az objektum keletkezési eseményében leírt kódrészlet akkor fog lefutni, amikor az adott osztályból keletkezik egy példány. Ezen elemeknek is vannak beépített változói, mint például a textúrájának az azonosítója vagy éppen az animációjának a sebessége.



3.1.2.5.1. ábra: Egy objektum textúra-, eseménykezelője és az esemény kódja

3.1.2.6 Rooms

A szobák az objektumok egy speciális alosztályának felelnek meg. Ezek eseményeibe is tudunk kódot írni és vannak beépített változói is, mint például a szélessége, magassága és a különböző rétegeinek neve. Az utóbbi segítségével képesek vagyunk például egy irányítóobjektummal a szoba hátterének megváltoztatására. A legfontosabb tulajdonsága egy ilyen speciális objektumnak az, hogy szó szerint a nyersanyagaink közül, az osztályokból rá tudunk húzni egyet, vagy akár többet is. Ekkor az adott elem megjelenik a szobában, már példány formájában. Így vagyunk képesek statikus szobákat létrehozni mindössze néhány pillanat alatt. Fontos megjegyezni, hogy az én esetemben egy szoba egy ablaknak felel meg, viszont ez nem mindig így van, mivel be lehet állítani nézeteket (viewport), amelyek segítségével képesek vagyunk egy hatalmas szobában mindig csak egy képernyőnyi területet kivetíteni a játékosnak. Ez egy nagyon erős eszköze a fejlesztői környezetnek, de nekem nem kellett ezzel élnem, mert minden szobám egy ablaknak felel meg.

3.1.3 A GML programozási nyelv

Mint ahogy már említettem, a fejlesztői környezetnek sajátos programozási nyelve van. Fontosnak tartom legalább érintőjelleggel megvizsgálni, mert tulajdonságai között akadnak elég érdekes dolgok is. Nagyon sokszor a C++-hoz fogom hasonlítani, mert ránézésre nincs sok különbség közöttük és ez a nyelv áll hozzám a legközelebb.

3.1.3.1 Változók

A GML-ben egy változó két értéket kaphat: valós számot vagy szöveget. Ezen típusokat nem kell feltüntetnünk változók létrehozásakor. Deklarálásuk alapján három fajta látókörük lehet:

1. Példányváltozó. Ezek létrehozásához nem kell semmilyen kulcsszó. Egyszerűen <változónév> = <érték> paranccsal már készen is állunk a használatukra. Ezen változóknak az a különlegessége, hogy csak egy példányban láthatóak, de annak minden olyan eseményében, amely a változó létrehozását tartalmazó esemény után következik. Ez azért nagyon fontos, mert képesek leszünk később ezek megváltoztatására más példányokon keresztül a *with* paranccsal. Például, hozzunk létre egy *player* példányt *hp* példányváltozóval a szobában és mellé egy *controller* nevű irányítóobjektumot. Az utóbbi *step* eseményébe a következő sort írom:

```
with(player){
self.hp -= 2;
}
```

Ekkor minden *frame*-ben, azaz képkockában le fog futni az előző kódrészlet, ami azt fogja eredményezni, hogy a *player* objektum *hp* példányváltozója fogyni kezd, méghozzá a *controller* objektum által. Így fog érvényesülni az objektumelvű programozásban már megismert példányok közötti interakció.

2. Lokális változó. Ennek a változónak látóköre csak arra az eseményre terjed ki, ahol létrehoztuk őt, majd az esemény végén törlődik. Létrehozásukhoz a *var* kulcsszó szükséges, azaz *var* <változónév> (= <érték>) paranccsal vagyunk képesek deklarálni őket. Ebben az esetben már opcionális az értékadás, miközben a példányváltozóknál szükséges.

3. Globális változó. Ezen változók látóköre az egész projektre terjed ki. Bárhol lehet rá hivatkozni, meg lehet változtatni. Deklarálásuk a *globalvar* kulcsszóval történik. Deklarálásukkal egy időben nem kaphatnak kezdőértéket. Vegyük például az alábbi utasítást.

```
globalvar timer = 0;
```

Erre fordítási hibát fog dobni a fejlesztői környezet. Így két sorban, a következő képpen kell megoldani.

```
globalvar timer;

timer = 0;
```

Igaz, hogy deklarálás szerint három nagy csoportot tudunk képezni, de van még egy nagyon fontos típusa a változóknak, mégpedig a beépítettek. Ezek példányokhoz tartoznak, és nem írhatjuk felül a nevüket, viszont értéküket megváltoztathatjuk. Ilyen például egy szoba hátterének az azonosítója.

3.1.3.2 Tárolók – különleges változók

Igaz, úgy volt, hogy egy változó csak két fajta értéket vehet fel, de a gyakorlatban több eszközt biztosít a GMS számunkra. Ilyenek a következők:

1. Tömbök. Ezen változók több érték tárolására szolgálnak. Nem muszáj, hogy megegyezzenek a benne lévő elemek típusai. Működésük és használatuk más

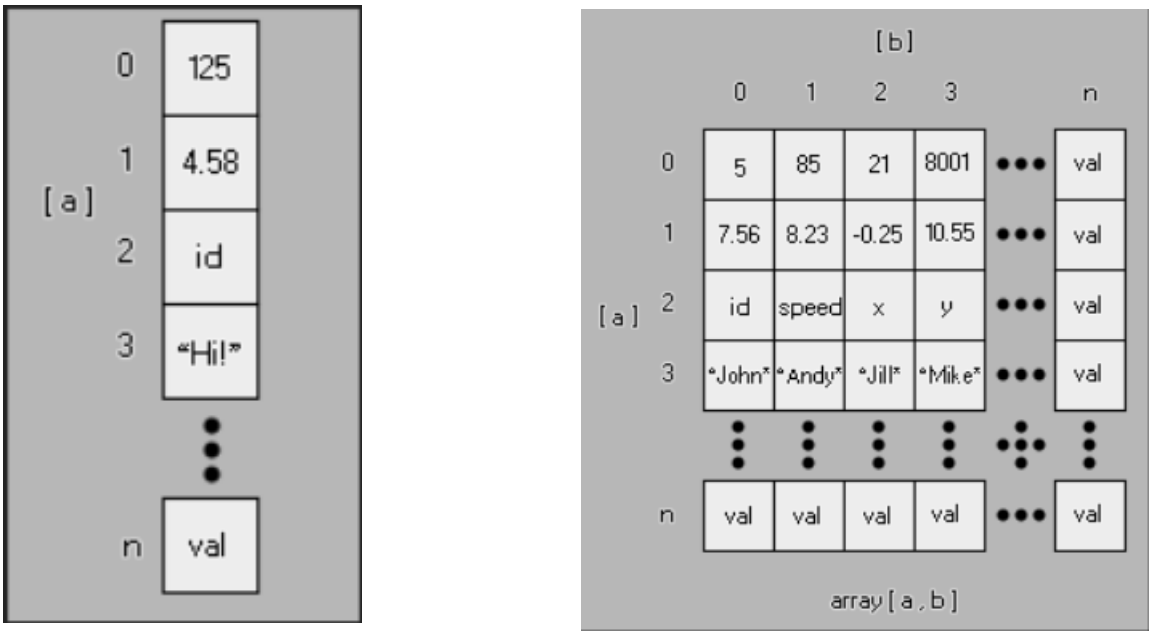
programozási nyelvekhez hasonlóan történik. Deklarálásuk ugyanúgy történik, mint bármelyik változónak:

(<kulcsszó>) <tömb neve>;

Ezek után a nullától számozott indexek és szögletes zárójelek segítségével adhatunk értéket az egyes elemeknek:

<tömb neve>[index] = érték;

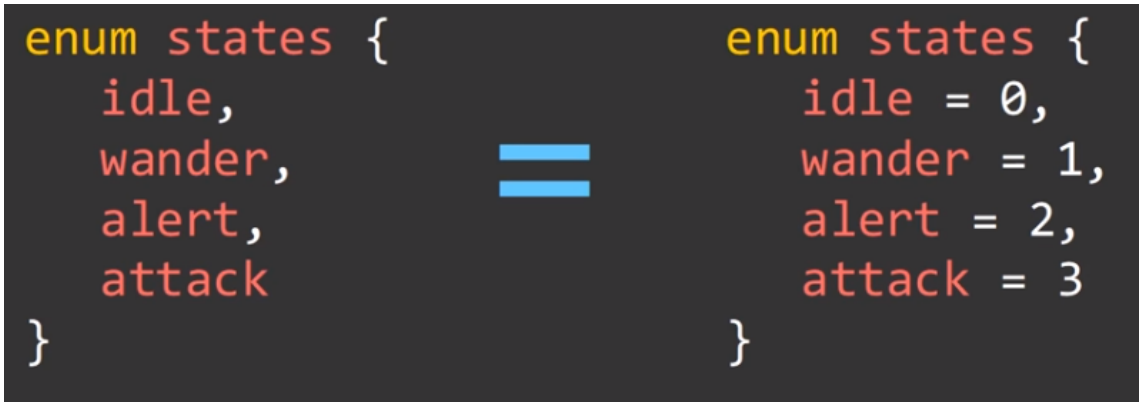
Vegyük észre, hogyha példányváltozóként szeretnénk tömböt létrehozni, akkor a deklarálás után még nem biztos, hogy tömb lesz. Miután adtunk valamely indexén lévő elemnek értéket, azután kezdi kezelni tömbként a fejlesztői környezet a változót. Fontos, hogy a fejlesztői környezet külön metódusokat ad arra, hogy létrehozzunk tömböket. Ezeket személy szerint nem használtam, mert nem volt szükségem a hozzájuk járó segítő függvényekre, mint például a keverés, maximum kiválasztás, vagy éppen a sorba rendezésre. Kétdimenziós tömböt *[index1, index2]* módon kell indexelni.



3.1.3.2.1. ábra: Az egy- és kétdimenziós tömbök reprezentációja

2. Boolean vagy logikai változó. Ez a megszokott módon működik a beépített *true* és *false* értékek segítségével. Ezen kívül a nyelvben minden valós szám, ami kisebb, mint 0.5, hamisnak, míg, amelyik nagyobb vagy egyenlő 0.5-nél az igaznak számít.

3. Enum vagy felsoroló változó. C++-ban megszokott rendszer alapján működik.



3.1.3.2.2. ábra: A felsoroló változó deklarálása

Ha a 3.1.3.2.2-es kép alapján járunk el, akkor a *states* nevű *enumerator* változónak az első (azaz nulladik) tagjára a *states.idle* utasítással tudunk hivatkozni.

3.1.3.3 Műveletek és kifejezések

A programozási nyelvben a megszokott +,-,*,/ műveletjelekkel tudunk a valós számok halmazán számolni. Fontos megjegyezni, hogy a + jel szövegek esetében konkatenációt jelent. Működnek a *+=, -=,* **=* illetve */=* műveletek is, amelyekkel néhány karaktert megspórolhatunk a kódunkban. Például a *hp += 3;* utasítás egyenértékű a *hp = hp + 3;* sorral és így tovább. A modulot és egész osztást a *mod* illetve *div* utasításokkal érhetjük el. Egyváltozós műveletek a *!,* amellyel kifejezéseket tagadhatunk, a -, ami segítségével egy szám -1-szeresét vehetjük, a ++, illetve -- jelekkel egy skalár értékét növelhetjük vagy csökkenthetjük eggyel.

Aki ismeri a C++-t, annak már menne is a kifejezések gyártása ezen a nyelven. Az összehasonlító műveletek: *<, <=, ==, !=, >=, >,* ahol sorban: kisebb, kisebb egyenlő, egyenlő, nem egyenlő, nagyobb, nagyobb egyenlő. Ezek használatával és az *&&, ||, ^^* (és, vagy, kizáró vagy) szimbólumok segítségével vagyunk képesek igaz vagy hamis kifejezések előállítására.

3.1.3.4 Elágazások és ciklusok

Valamilyen programozási nyelvben jártas személynek ez a rész talán unalmas lesz, de fontosnak érzem gyorsan áttekinteni a legfőbb építőelemeinket ebben a nyelvben, mert vannak ismeretlen szintaktikai jelölések néhol.

1. Elágazások. Nincs semmi különbség a megszokottól:
    a. Egyágú: *if (<feltétel>) {<utasítások>}*

b. Többágú: *if (<feltétel1>) {<utasítások1>}*

else if (<feltétel2>) {<utasítások2>}…

2. Repeat ciklus. Ez a ciklus nekem újdonság volt, mivel még nem találkoztam hasonlóval. A szintaxisa a következő: *repeat(<szám>) {<utasítások>}*. Ez a ciklus <*szám*>-szor futtatja le az <*utasítások*>-ban foglalt kódrészletet.

3. While ciklus. Ez is a megszokott módon működik: *while (<feltétel>){<utasítások>}*. Amíg a <*feltétel*> igaz, addig ismétli az <*utasítások*>-okat. Ezen fajta ciklusnak a hátultesztelő változata: *do {<utasítások>} until (<feltétel>)*. Ugyanazt teszi, mint az utóbbi, csak az utasítások végén figyelt <*feltétel*> bekövetkezésekor lép ki a ciklusból.

4. For ciklus. Nincs különbség e között és a C++-os megvalósítás között. Szintaxisa: *for (<ciklusváltozó deklarálása>; <feltétel>; <utasítások>) {ciklusmag}*. Ez annyiban különbözik az előltesztelős *while* ciklustól, hogy ezzel képesek vagyunk az első lefutás előtt egy *ciklusváltozót* létrehozni, amivel befolyásolni tudjuk a *ciklusmagban* lévő utasításokat.

5. With. Már korábban, a változóknál (3.1.3.1) említettem ennek az utasításnak a fontosságát és hatalmas jelentőségét. Ezt írja a fejlesztői környezet a legerősebb eszközének, mivel ezzel valósították meg a példányok egymás közötti információcseréjét és kölcsönhatásukat. Vegyünk egy egyszerűbb példát. Legyen egy *obj_button* és egy *obj_background* példányom. Azt szeretném megvalósítani, hogy ha rákattintok az előbbi objektumra, akkor az utóbbiban változzon meg a keletkezésekor deklarált *background_id* nevű változó. Ekkor a következő kódot kell írnom az *obj_button* példány kattintás eseményébe:

   with (obj_background){ self.background_id = 1; }

   Már meg is oldottuk a problémát egy ilyen egyszerű sorral. Vegyük észre, hogy a kapcsos zárójelen belül lévő részt már írhatom úgy, mintha az *obj_background*-ban lennék, így szükségtelen rá hivatkozni. Biztonsági és olvashatósági szempontból ajánlott ilyenkor a *self* különleges beépített változót használni. Ez annak a példánynak az azonosítóját adja vissza, amelyikben írjuk. Esetünkben az *obj_background*-é. Az általános szintaxis:

   with (<példány aznosító(k)>){ <utasítások> }

Fontos megjegyezni, hogy nem csak egy példánynak állíthatjuk át valamely mezőjét, hanem egyszerre akár többnek is. Ha az utóbbira törekszünk, akkor akár a szobában lévő összes objektumot is megadhatjuk a beépített különleges *all* nevű változó segítségével. Ez visszaadja mindegyik, a szobában lévő példány azonosítóját, amelyeknek a mezőit, a *with* segítségével, egyszerűen módosíthatjuk kedvünkre. Ezeken kívül létezik még az *other* nevű különleges változó, amely mindig azt az azonosítót adja vissza, amelyikbe írtuk. Így a fenti példával élve, ha még a kapcsos zárójelbe írnánk egy *other.x += 40;* utasítást, akkor az *obj_button* példányt tolnánk el vízszintesen, negyven pixellel jobbra.

3.1.3.5 Események

Kódrészleteink lefutását általában egy történés váltja ki. Ezeket fogjuk eseményeknek nevezni és ilyenek köré fogjuk kódunkat szervezni. Ha szeretnénk kitalálni valamilyen új játékalkalmazást, vagy csak egy főmenüt szeretnénk csinálni már meglévő szoftverünkhöz, akkor biztosan eszünkbe jutnak olyan dolgok, mint például „ha rámegyünk erre a gombra, akkor visszalépünk” vagy „ha megnyomom a szóköz billentyűt, akkor ugorjon egyet a karakterem”. Így szinte egyből elkezdünk történésekben, interakciókban gondolkodni. Ezért is hatalmas segítség a fejlesztői környezet részéről, hogy külön szedte ezeket, és roppant egyszerűen tudunk velük dolgozni. Minden példány rendelkezik megannyi eseménnyel, viszont szinte mindegyiket képesek vagyunk helyettesíteni kettővel:

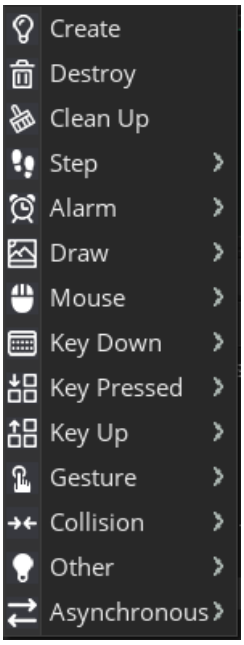
1. *Step* esemény. Erről már szó volt korábban, mivel ez a legfontosabb. Mivel az ebbe foglalt kódunk minden *frame*-ben, azaz alkalmazásbeli képkockában lefut, így szinte minden másik eseményt helyettesíteni lehet vele beépített függvények segítségével. Vegyük például a következő problémát. Azt szeretnék, hogyha egy adott gombra kattintunk egerünk bal gombjával, akkor ugorjunk át egy másik szobába. Ekkor, ha ismerjük a *Mouse left pressed* eseményt, akkor egyszerűen belemegyünk és beleírjuk a következő sort:

   room_goto(room_id);

   Így a *room_id* nevű szobába fogunk ugrani, ha rákattintunk a gomb példányra. Viszont ezen feladat megoldásához nem feltétlen kell ismernünk ezt az eseményt, elég ha a *step*-el és még két beépített függvénnyel tisztában vagyunk. A két segédfüggvényünk a *mouse_check_button_pressed(button),* illetve a

position_meeting(x,y,obj). Mindkettő igaz vagy hamis értéket adhat vissza. Utóbbi metódusunk azt ellenőrzi, hogy az *x,y* ponttal az *obj* példány érintkezik-e. Az előbbi függvény figyeli az egerünk gombjait és igazat ad vissza, ha a *button* gombot nyomjuk le. Ez az argumentum különleges, beépített értékeket vehet fel. Nekünk most a *button=mb_left* eset fog kelleni. Problémánk megoldásához annyit kell tennünk, hogy ellenőrizzük először, hogy egerünk érintkezik-e a gombbal, majd ha igen, akkor lenyomjuk-e a bal egérgombot. A következő, *step* eseménybe írt kódsorokkal már ki is váltottuk a speciális történést a step-*re.*

```
if( position_meeting(mouse_x,mouse_y,obj_button) and
    mouse_check_button_pressed( mb_left )){
    room_goto(room_id);
    }
```



2. *Create* esemény. Ez az egyetlen, amit nem lehet kiváltani *step* segítségével. Az objektumorientált programozásban megismert konstruktor szerepét tölti be ez az esemény, mivel akkor fut le, amikor példányunk keletkezik. Itt általában inicializálunk változókat, alapbeállításokat végzünk.

3.1.3.5.1. ábra: Az összes esemény listája

3.2 A játékszoftverek

3.2.1 Trictactoe

3.2.1.1 Algebrai megközelítés

Ahogy már a bevezetésben említettem, az ötletek alapjait a szoftverekhez az internetről, a *Numberphile Youtube* csatornájáról szereztem. A Notactoe, vagy esetünkben Trictactoe játékban egy hozzá készült dokumentáció keltette fel rettenetesen a figyelmemet. Ezen írás algebrai szempontból vizsgálja a játékot és így veszi szemügyre a tökéletes lépéseket. A dokumentumban lévő módszer gyors összefoglalója:

1. Egy tábla minden állásához rendeljünk hozzá különböző szimbólumokat (igazából szorzatokat), amelyek az a, b, c, d, 1 változókból állnak.
2. A táblákhoz rendelt szimbólumokat szorozzuk össze úgy, mintha algebrai változók lennének. Például az $a * a = a^2$ vagy éppen a $b * 1 = b$. Így kapunk egy összefoglaló algebrai kifejezést, ami képet ad a játék állásáról.
3. Redukáló szabályokkal is ellátnak bennünket, amelyek segítségével az összefoglaló kifejezést leegyszerűsíthetjük. Ezzel a módszerrel egy halmazba, pontosabban egy kommutatív monoidba rendezzük a szorzatokat.
4. Ezen monoid elemei között vannak különlegesek. Ha lépésünk után egy ilyen szimbólumot kapunk, akkor nyerésre állunk. Ezen szorzatok: a, b^2, bc, c^2.

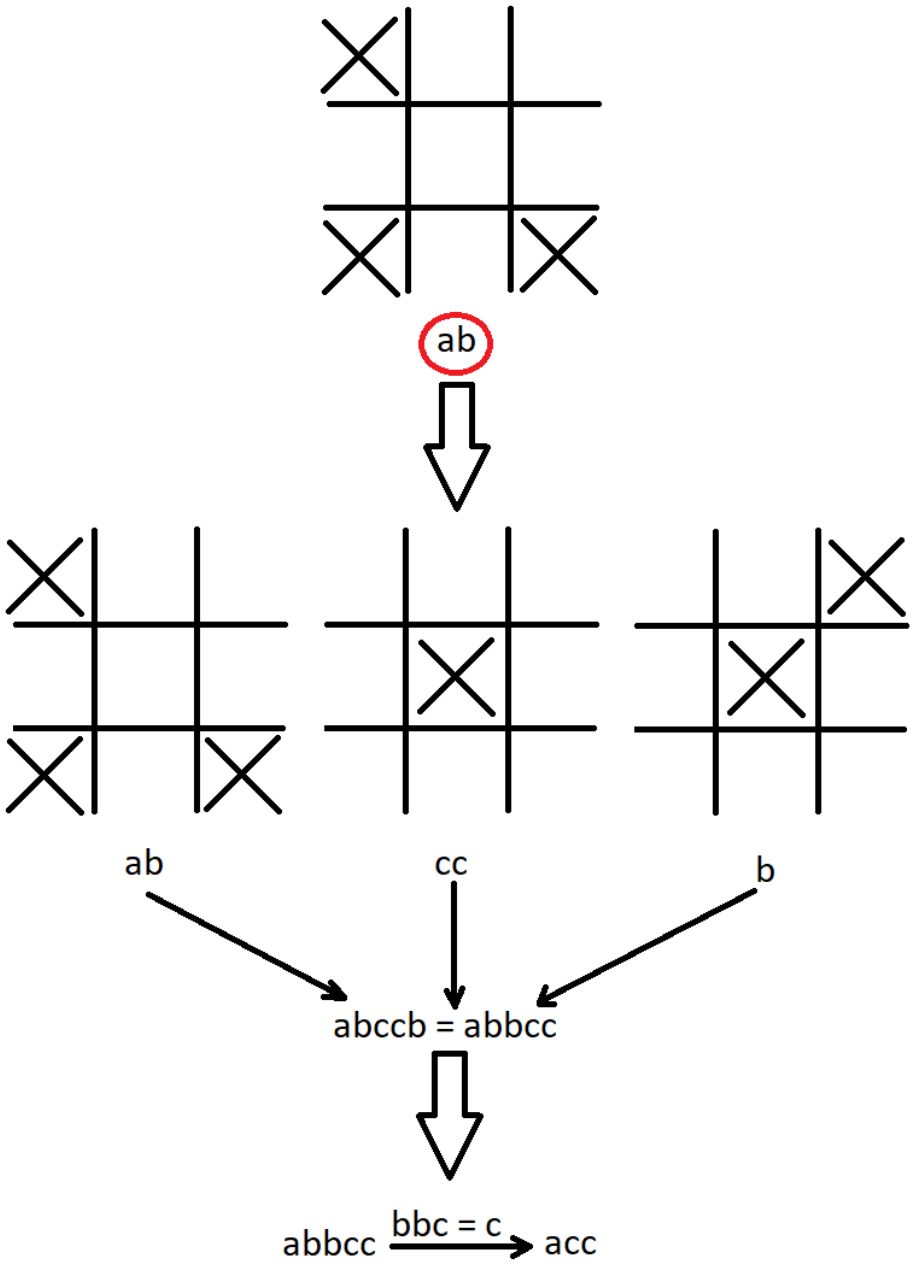
Persze a szóban forgó szorzatokat nem nekünk kell kitalálni, a fentebb említett dokumentáció tartalmaz egy oldalnyi leírást arról, hogy melyik álláshoz melyik szimbólum tartozik. Összesen 102 izomorf eset van, azaz ebben nincsenek benne ezek elforgatottjai. Ha a matematika eszközeivel szeretnénk leírni monoidunk szerkezetét, akkor a következő képpen nézne ki:

$$Q = \langle a, b, c, d \mid a^2 = 1,\ b^3 = b,\ b^2c = c,\ c^3 = ac^2,\ b^2d = d,\ cd = ad,\ d^2 = c^2 \rangle$$

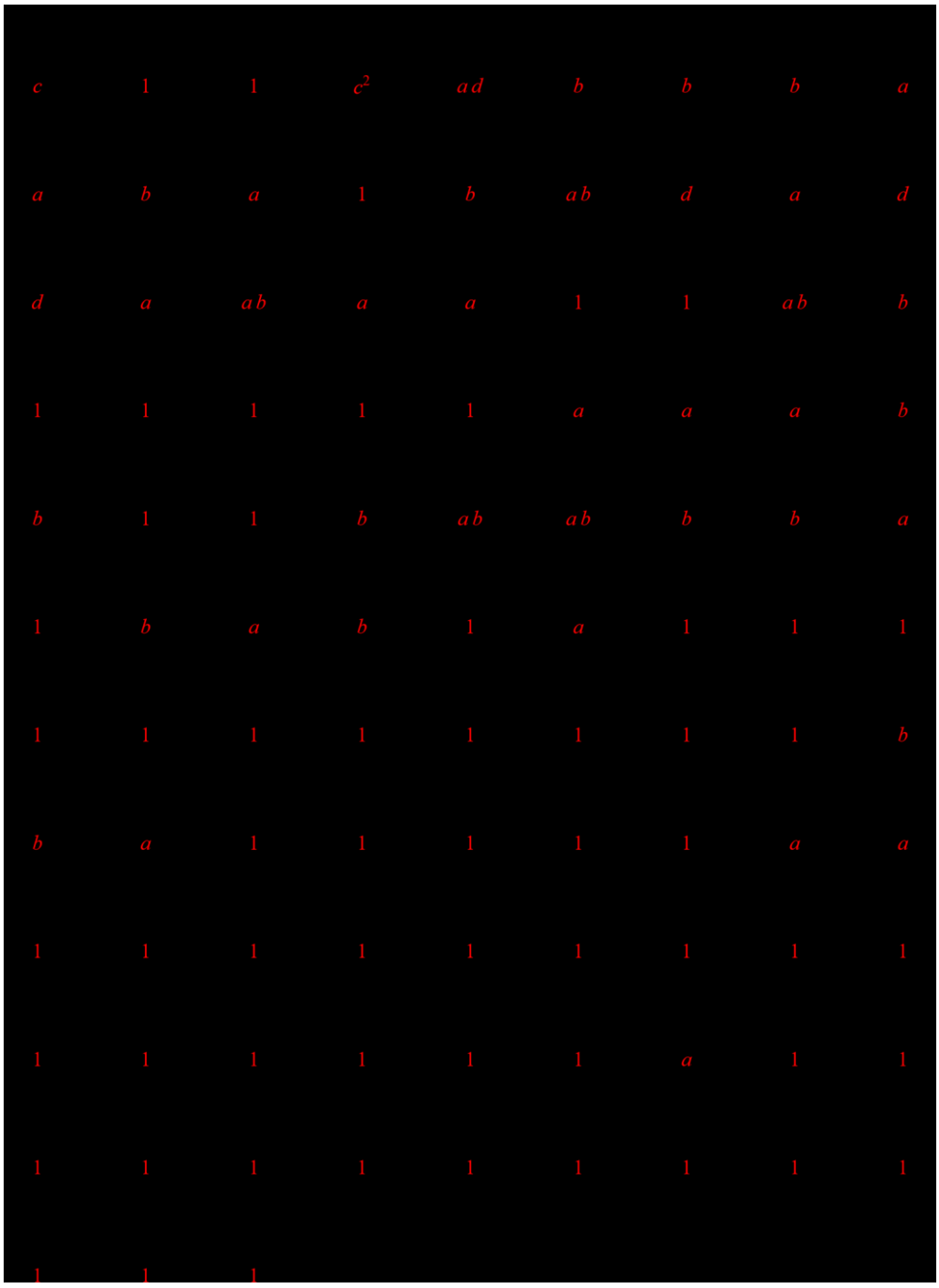
Ezen leírásból az következik, hogy a Q-nak tizennyolc eleme van, amelyek a következők:

$$Q = \{1, a, b, ab, b^2, ab^2, c, ac, bc, abc, c^2, ac^2, bc^2, abc^2, d, ad, bd, abd\}$$

A különleges, azaz nyerő szorzatok pedig: $P = \{a, b^2, bc, c^2\}$

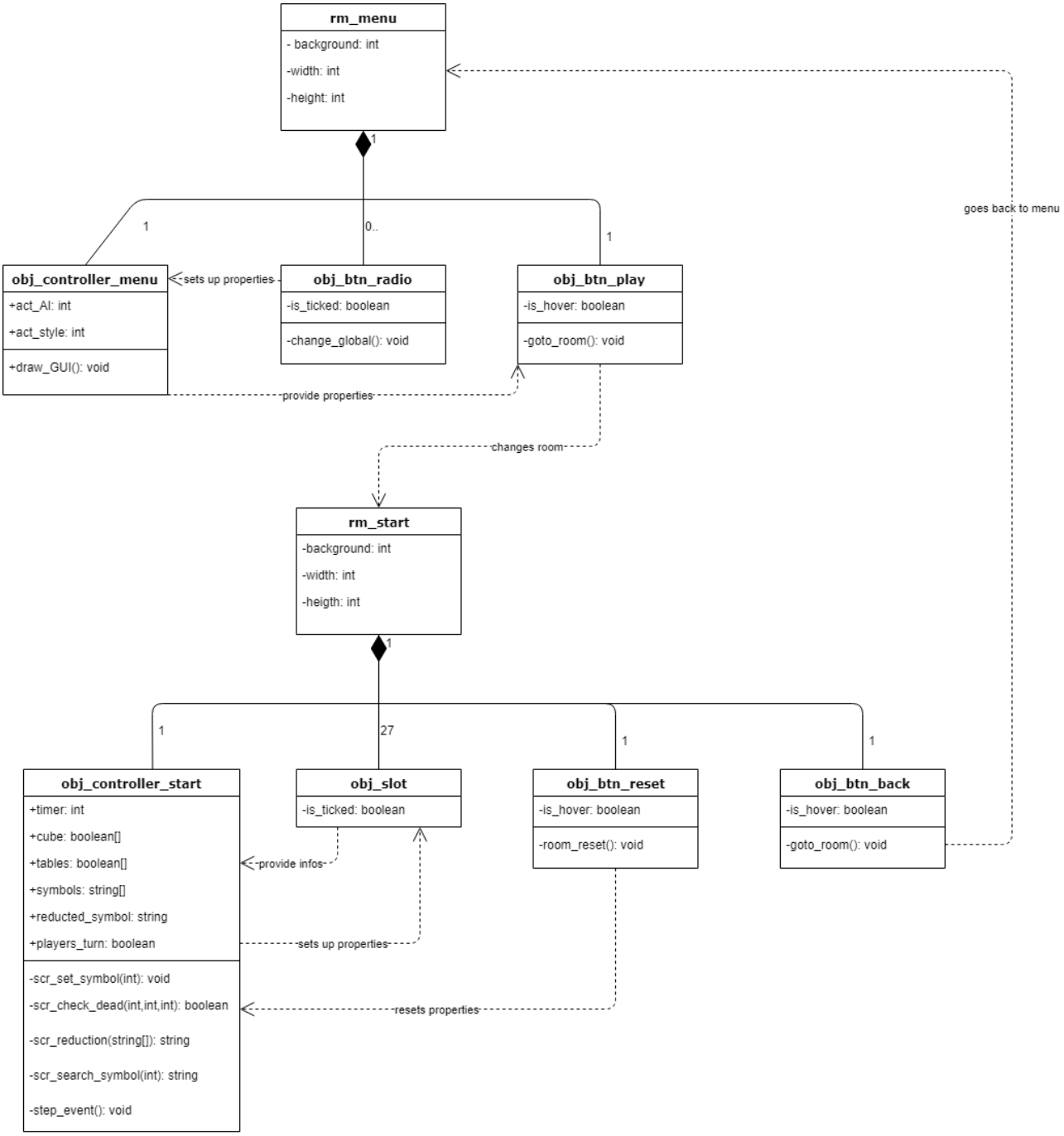


3.2.1.1.1. ábra: Az algebrai eljárás gyakorlatban. Felülről lefele: táblákhoz szimbólum rendelése, azok összeszorzása, annak redukálása.



3.2.1.1.2. ábra: A különböző állásokhoz rendelt szimbólumok listája

3.2.1.2 Tervezés és megvalósítás



3.2.1.2.1. ábra: A Trictactoe UML diagramja

Az UML diagram mutatja a szoftver általános működési elveit, a példányok szerepeit. A rendszer működésének megértéséhez nézzük végig a fontosabb komponensek szerepét, fontosabb változóinak jelentését.

1. *rm_menu:* Ez lesz a menünk szobája, esetünkben ablaknak is nevezhetjük. Erre szeretnénk elhelyezni beállítógombjainkat és az indítógombunkat. Ezek megfelelő működéséhez kell egy irányítópéldány, amely mozgatja az adatokat a háttérben.

a. *background*: A menünk hátterének azonosítója. Megvalósításban ez egy egész számként van ábrázolva és a létrehozáskor inicializálva lesz.

b. *width:* A menü szobánk szélessége. Ez a mező is egyből egész szám értéket kap, fejlesztői környezeten keresztül lett beállítva, mivel ha dinamikusan változtatnánk, akkor az ablakméretünk is más lenne, így felbontásunk elromlana. Elnyújtott vagy éppen összenyomott textúrákat eredményezne, így ezt és a következő mezőt is érdemes a fejlesztői környezetben egyszer beállítani és utána nem bántani.

c. *height:* Hasonlóan az előzőhöz, ez a menünk magassága.

2. *obj_controller_menu*: Ahogy a neve is sugallja, ő lesz a mi irányítóobjektumunk. Legfőbb feladata összegyűjteni a menüben lévő példányoktól a beállítási adatokat és ezeket tovább küldeni az indítógombnak.

    a. *act_AI*: Ez az aktuális gépi játékosunk nehézségi fokozata. Egész számként ábrázolva, -1,0,1,2 értékeket vehet fel, melyek jelentése: egymás elleni játék, könnyű, közepes és nehéz gépi játékos. Ezen változó adata a választógombok adataitól függ. Alapbeállításként a 0-t kapja.

    b. *act_style:* Hasonlóan az előző mezőhöz, ez az adattag is a beállítógomboktól függ. Az aktuális stílus sorszámát jelképezi. Továbbítva ezt az értéket az indítógombnak, az képes lesz a megfelelő stílus beállítására. 0,1,2 számok fogják jelenteni a barna, kék, illetve sötét stílust.

    c. *draw_GUI():* Privát eljárás, amelynek a feladata a menü felhasználói felületének kirajzolása, megjelenítése. Ez alatt a gombokra írt szövegeket értem. Gyakorlatban ez a fejlesztői környezet által biztosított *draw* eseménybe írt kódrészlet. Azért nem váltottam ki ezt *step*-el, mivel ez egy külön felhasználói felület kirajzolására készített esemény. Az ebbe írt beépített, kirajzolást segítő függvények a leghatásosabban működnek.

3. *obj_btn_radio:* A kiválasztógomb objektuma. Egy szobában egy sem, vagy akár több is lehet belőlük. Egyszerű választás elé állíthatjuk velük a játékost, például stílusválasztáskor. Nagyobb szerepük az, hogy az irányítóobjektum adattagjait változtatja meg.

a. *is_ticked:* Egyetlen példányváltozója kizárólag logikai értéket vehet fel és ennek segítségével fogják egymást figyelni az összetartozó, egy csoportban lévő kiválasztógombok. Előbbiekre példa a három gomb a stílusok kiválasztásához. Ezek összetartoznak, azaz figyelik egymást, hogy csak egy legyen mindig kipipálva. Kinézetük is ettől függ. Ha igazat vesz fel, akkor kijelölt, egyébként üres textúrát rajzolunk ki nekik.

b. *change_global():* Privát eljárás, amellyel képesek vagyunk az irányítóobjektumban lévő nyilvános adattagok megváltoztatására. Gyakorlatban úgy működik, hogy mindegyik kiválasztógomb *step* eseményében figyeli, hogy rákattintanak-e. Ha igen, akkor az összes többi, vele egy csoportban lévőről leveszi a jelölést és átírja a nyilvános értéket a hozzá tartozóra. Például, ha nekünk a könnyű gépi játékos van kiválasztva a menüben, akkor biztosak lehetünk, hogy az *act_AI* változó 1 értéket vesz fel. Viszont, ahogy rákattintunk a „Nehéz” felirat felett lévő kiválasztógombra, akkor az megváltoztatja a könnyűnek textúráját és ezzel együtt átállítja az *act_AI* nyilvános változót 2-re.

4. *obj_btn_play:* Indítógomb. Feladata az adott paraméterek melletti játszma indítása. Gyakorlatban úgy alakult, hogy csak egy pályaméretet volt értelme elkészíteni, mivel a többtáblás pályának a mechanikája ugyanilyen lett volna. Így végül ez csak egy statikus gomb egy animációval, amely akkor indul, ha ráhúzzuk egerünket.

    a. *is_hover:* Logikai példányváltozó. Ennek segítségével tudjuk, hogy egerünk a gombon van-e. Ebben az esetben egy animáció indul, mely egy Trictactoe vázlatos játszmáját mutatja be.

    b. *room_goto():* Privát eljárás. Ezen függvénnyel képesek vagyunk a játéktér ablakára ugrani, azaz elindítani a játékot. Fontosnak érzem megjegyezni, hogy azért nem kap az eljárás egy argumentet sem, mivel csak egyetlen ablakba mehetünk, azaz nincs más szerkezetű pálya, mint a három táblás.

5. *rm_start:* A játéktér szobája. Ugyanazon tulajdonságú adattagjai vannak, mint az 1-es pontban bemutatott *rm_menu*-nek, így nem tartom fontosnak újabb leírását.

6. *obj_controller_start*: Ez a játéktérnek az irányítóobjektuma. Az objektum gyűjti be az információkat a játék adott állásáról, változtatja meg a négyzetek és táblák

textúráját, itt gondolkodik a gépi játékos és teszi is meg lépését, illetve váltakoztatja a köröket. Mondhatjuk, hogy ez a példány a játszma mozgatórugója és legfontosabb része.

a. *timer:* Nyilvános, egész szám típusú adattag. Ezzel a mezővel képesek vagyunk a gépi játékos lépését kissé lelassítani. Ez azért fontos, mert egyrészt a játék olyan hibákat produkálna, mint például az emberi játékos körének megismétlése bármilyen ok nélkül, másrészt így sokkal jobban látható, hogy hová tette az X-ét gépi ellenfelünk. Előbbi hiba bizonyára azért lehet, mert nem konzisztens különböző objektumok *step* eseményének lefutásának sorrendje, így kiéheztetés jön létre. Mivel gyakorlatban a *step*-ben tudunk időzítőt állítani, így ezt a változót nullára kell inicializálni. Minden egyes *frame*-ben növelnünk kell eggyel értékét, így ha elér egy adott számot, akkor jöhet a gépi játékos. De mi is ez a szám? Ha például azt szeretném, hogy két másodperc teljen el mielőtt lép ellenfelem, akkor ezt a *timer*-t akkor kell lenullázni, ha eléri a 120-at. Ez azért van, mert a játék 60 képkockát mutat másodpercenként, így ha minden képkockában növelem, akkor két másodperc alatt 120-al növeltem. Létezik egy beépített változó, amely a *room_speed* névre hallgat. Ez mindig az adott szoba gyorsasága, azaz hogy hány képkockát játszik le a szoftver másodpercenként. Így, ha nem a statikus 60-as számmal szeretnénk dolgozni, mert lehetséges, hogy később megváltoztatjuk a játék gyorsaságát, akkor a következő sorral érdemes állítani az időzítőt:

```
if(timer >= 2*room_speed and !players_turn){
    //gépi játékos kódja
    }
```

b. *cube:* Logikai értékeket tároló egydimenziós tömb. Ez a változó fogja nyilvántartani, hogy melyik négyzetben van és melyikben nincs X. Neve abból adódik, hogy gyakorlatban ez egy háromdimenziós tömb, csak egy dimenzióssal ábrázolva. Ha azt szeretnénk megnézni, hogy az x-edik tábla, y-odik sorának, z-edik oszlopában van-e X, akkor a

$$cube[9*(x-1) + 3*(y-1) + (z-1)]$$

változót kell megvizsgálnunk. Igaz, nehezebben látható át így a kód, de a Game Make Language-ben sajnos csak kétdimenziós tömböt könnyű kezelni és létrehozni. Többdimenzióssal sokkal több gond lett volna, így emellett a megoldás mellett maradtam.

c. *tables:* Logikai értékeket tároló egydimenziós, háromelemű tömb. Ez egy segédváltozó, amellyel mindig könnyedén meg tudom állapítani, hogy melyik tábla él vagy halt meg. Fontos lesz az is a gépi játékosok számára, hogy az utolsó táblán vagyunk-e vagy sem, így ezzel a tömbbel gyorsan és egyszerűen tudtam kezelni ezt a problémát. Elemei kezdőértéknek az igazat kapják, mivel eleinte mindegyik táblánk él.

d. *symbols:* Szövegeket tároló, háromelemű tömb. Már volt szó a játék algebrai megközelítéséről. Abban különböző szimbólumokat rendeltünk az egyes állásokhoz. Ezen változó segítségével tárolom az egyes táblákhoz rendelt szimbólumokat. Ezek minden lépés után beállításra kerülnek. Az egyszerűség kedvéért, ha az 1-es számot rendeljük hozzá valamely táblához, akkor azt az üres stringgel helyettesítettem. Ez majd akkor lesz fontos, amikor ezen jeleket össze kell szorozni. Alapértékül minden eleme a „c” karaktert kapja, mivel az üres táblákhoz ezt a szimbólumot rendeljük hozzá.

e. *reducted_symbol:* Egy szövegtípusú változó. Ennek értéke a *symbols* elemeitől függ. Majd később láthatjuk, hogy egy eljárás során állítjuk be ezt a változót, miután összeszoroztuk a szimbólumokat és leegyszerűsítettük az adott szabályokkal. Ez lesz a végső, minden információt hordozó változó, amit a gépi játékosok figyelnek majd, mielőtt lépnek.

f. *players_turn:* Egyszerű logikai változó. Ennek segítségével vagyunk képesek váltakoztatni a köröket. Igazat kap értékül, ha a gépi játékos, hamisat, ha az emberi játékos lépett. Alapértékül igazat állítom be neki, mivel sajnos a közepes nehézségtől kezdve nem lenne értelme úgy játszani, hogy a gép kezd. Általában egy játékban meg tudjuk mondani, hogy kedvez-e a kezdési lehetőség. Ha a kezdő fél tökéletesen lép az egész

menet során, akkor a másik játékos visszafordíthatatlan hátrányba kerül, így nincs értelme ezt a változót hamissal inicializálni.

g. *scr_set_symbol(int):* Egész számot váró privát eljárás. Ennek segítségével állítjuk be a *symbols* változó argument sorszámú elemét. Mivel ez az eljárás elég bonyolult és műveletigényes, ezért mindig csak annak a táblának szeretnénk kiszámolni a szimbólumát, amelyre éppen valaki lépett. Részletesebben láthatjuk működését a Gépi játékosok fejezetben, mert ezt ők használják.

h. *scr_check_dead(int,int,int):*Három egész számot váró, privát eljárás. Ezen függvény minden kör után lefut, azt ellenőrizve, hogy lépésünkkel „megöltük-e" az adott táblát. Ha igen, akkor egy szürke textúrát tesz fölé, hogy többé ne tudjunk odakattintani, illetve beállítja a *tables* változó megfelelő elemét hamisra.

i. *scr_reduction(string[]):* Ez a redukáló függvényünk. Ennek segítségével képesek vagyunk az argumentumnak megadott *symbols*-okat összeszorozni, majd azt leegyszerűsíteni a szabályokkal. Visszatérési értéke egy szöveg, amely már a végső szimbólumsorozat.

j. *scr_search_symbol(int):* Ez a privát függvény az *scr_set_symbol* eljárásnak a segédfüggvénye. Képesek vagyunk a táblasorszámot argumentumnak adni és így kikeresni az ahhoz tartozó szimbólumot a listából. Gyakorlatban ez hat függvény összessége, amelyekkel addig szűkítem a 102 izomorf táblaállást, amíg meg nem találom az adott táblához tartozót.

k. *step_event():* Nyilvános eljárás. Ez a függvény felelős a körök váltakoztatásáért a *timer* segítségével. Gyakorlatban persze ez a *step* esemény, amelyben a kódunk van.

7. *obj_slot:* Egy négyzet objektuma. 3*9, azaz 27 darab ilyen példányunk van a játéktéren. Kétféle textúrája van: üres, illetve beikszelt.

   a. *is_ticked:* Logikai típusú példányváltozó, amely értékétől függ az objektum textúrája. Kezdőértéke hamis, mivel eleinte nincs X a négyzetekben. Kattintásra igazra vált.

8. *obj_btn_reset:* Az újraindító gomb objektuma. Rákattintva kezdőértékekre állíthatjuk a szobában lévő objektumok összes változóját. Így a játékos szemszögéből újraindul a játék. Kétféle textúrája van: alap és benyomott.

    a. *is_hover:* Logikai típusú adattag. Igaz értéket vesz fel, ha egerünk a példányon van, hamisat különben.

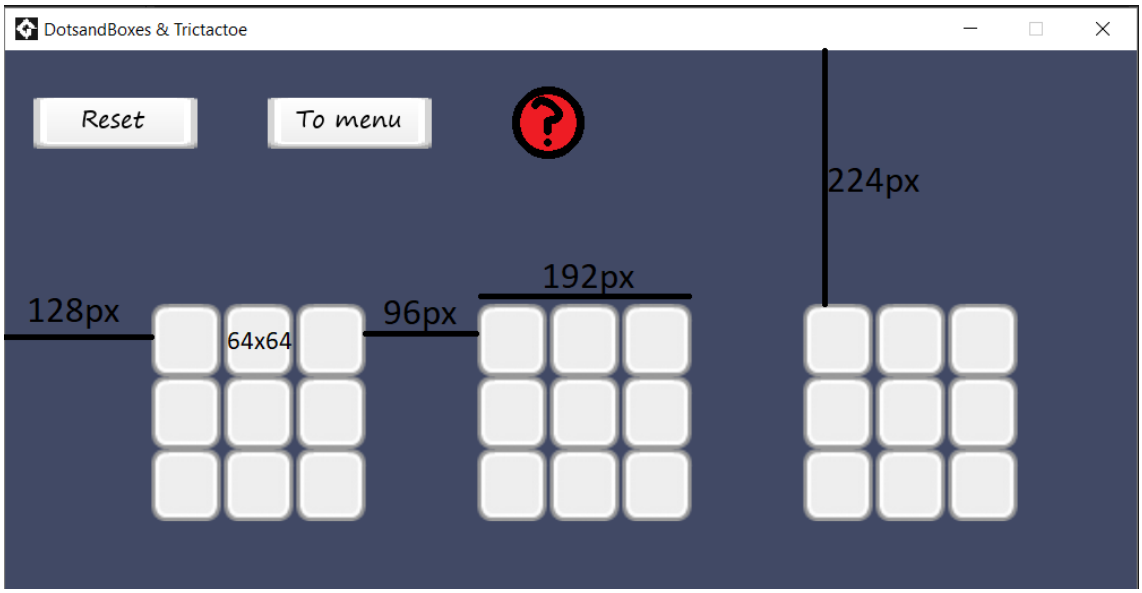
9. *obj_btn_back:* A visszagomb objektuma. Rákattintva visszajuthatunk a játék főmenüjébe. Ugyanazon módon működik szobaváltó funkciója, illetve példányváltozói szerepe, mint az *obj_btn_play*-nek.

3.2.1.3 Akadályok és megoldásaik

Előző részben a játékszoftver szerkezete lett bemutatva. Az implementálás közben felmerülő problémák megoldásait csak néhol érintettem, de kifejezetten fontosnak érzem részletezni őket, mert nagyon különleges, egyedi akadályokkal kerültem szembe a játék fejlesztése során. Megjegyzem, hogy egy ilyen fejlesztői környezetben, ehhez hasonló egyszerű játék színterét berendezni, mechanizmusait megírni rettentően egyszerű, így nem is térek ki rájuk.

- Probléma: Milyen adatszerkezettel legyen megoldva az információtárolás? Hogyan, mikor és milyen módszerrel legyen karbantartva a tároló ahhoz, hogy a gépi játékosok körükben teljesen képben legyenek a játék állásával?

  Megoldás: Adatszerkezetnek nem gondoltam semmi bonyolultat, mivel csak az a feladata, hogy a játéktér állapotát eltárolja, így maradtam a tömbnél. Karbantartása már nagyobb kihívás volt. Úgy gondoltam a legegyszerűbbnek, hogy ha a játéktérre feszítem ki adatszerkezetemet, azaz a négyzetek koordinátái alapján módosítom a *cube* tömb elemeit. Ezt a módszert azért éreztem hatékonyabbnak, mint mást, mert a fejlesztői környezet amúgy is számon tartja az egyes példányok x és y koordinátáját. Így végül az emberi játékos lépésének kódja a négyzetek, tehát az *obj_slot* objektumok *step* eseményébe kerültek. Itt egyben tudom egyszerűen ellenőrizni, hogy rákattintottak-e az adott négyzetre, és a beépített koordinátaváltozókkal ki is tudom számolni, hogy melyik tábla, hányadik sora, oszlopa. Ehhez viszont egy icipici matematika kellett. A játékban úgy működik a koordinátarendszer, hogy a bal felső sarokban van az origó és lefele az y, jobbra pedig az x nő. Első lépésként berendeztem a színteret.

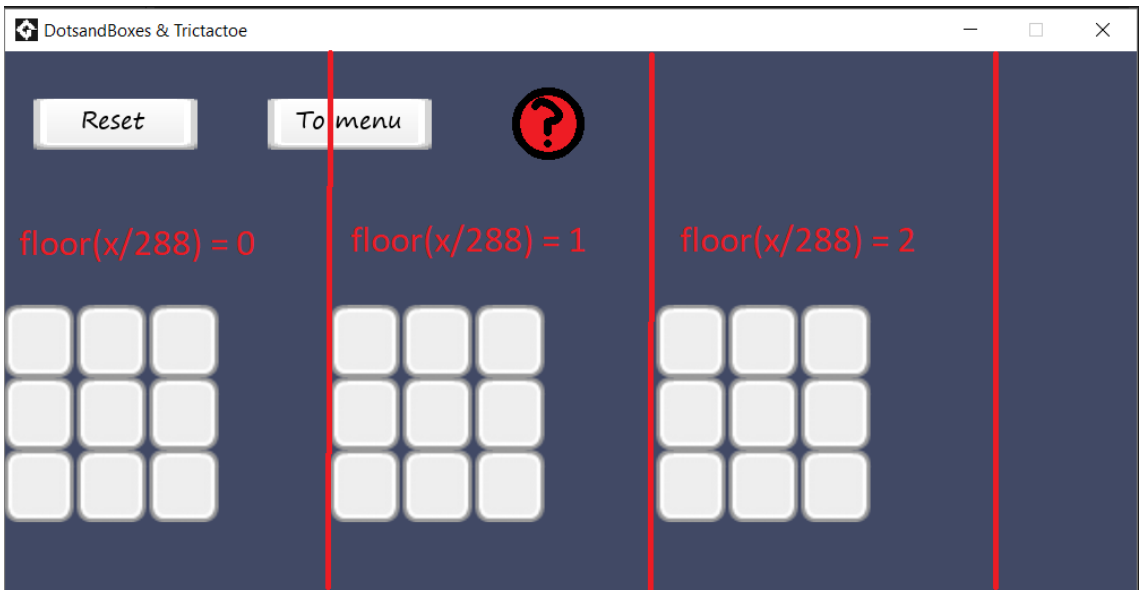


3.2.1.3.1. ábra: A játék színtere a fontosabb méretekkel

A táblaszám kiszámítását úgy oldottam meg, hogy eltoltam a táblákat 128-al balra, hogy ne legyen rés az első és az ablak széle között, majd nagyobb szeletekre bontottam a képernyőt, melyek meghatározzák majd a tábla számát. Így a képlet a következő:

$$cubeno = floor((x-128)/288);$$

ahol a *floor()* függvény az argumentum alsóegészrészét adja vissza, x pedig az adott négyzet vízszintes koordinátája. A 288-as szám a 96+192-ból jön ki, mivel e részeken húzok egy képzeletbeli függőleges vonalat.

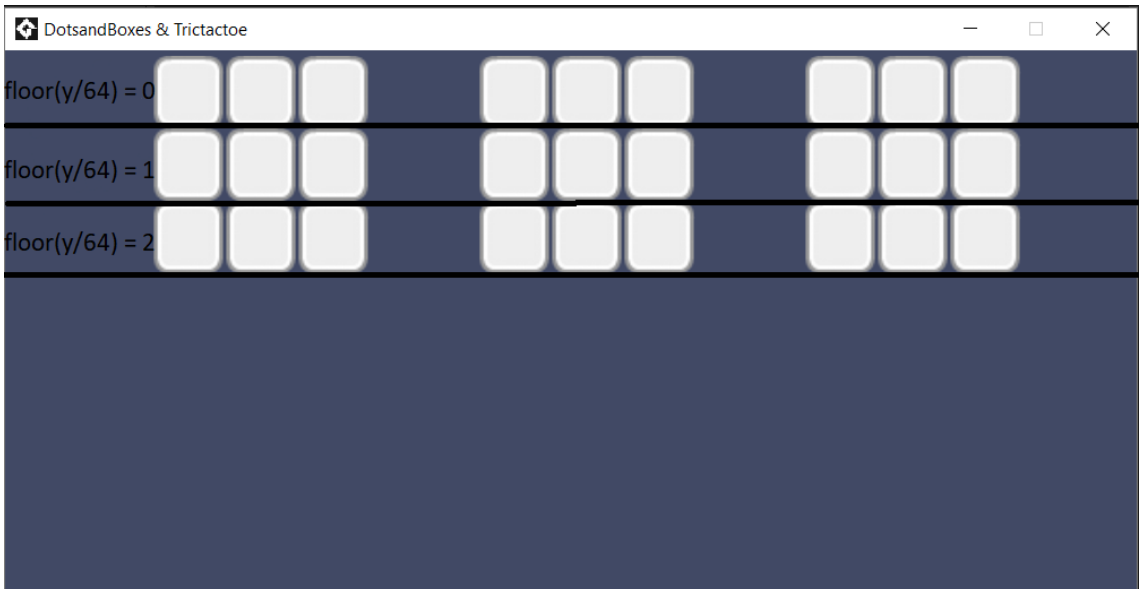


3.2.1.3.2. ábra: Színtér a képzeletbeli eloltás után, feldarabolva

A sor számát ugyanezen gondolatmenet alapján kapjuk meg. Felcsúsztatjuk a táblákat a színtér telejéig és feldaraboljuk. Így a sor képlete:

$$row = floor((y-224)/64);$$

ahol y az adott négyzet függőleges koordinátája. A 64-es számmal azért osztunk, mert aszerint darabolunk, mivel egy négyzet magassága 64 pixel.

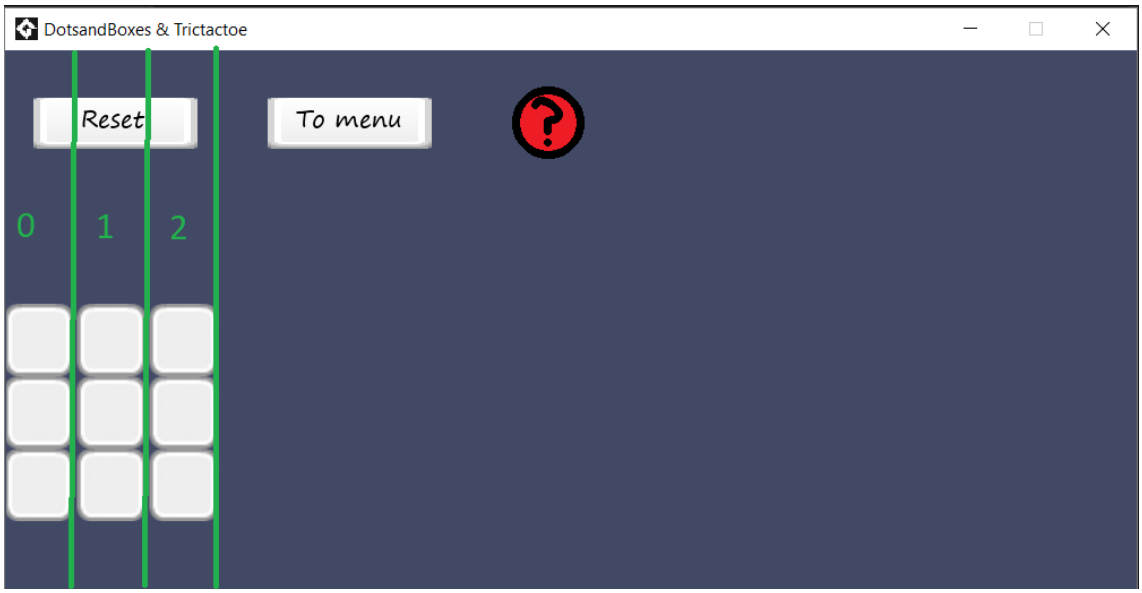


3.2.1.3.3. ábra: Színtér a képzeletbeli felcsúsztatás után, feldarabolva

Az oszlop számát egy kissé bonyolultabb számolni. Elképzelésem az volt, hogy az adott táblát áttranszformálom a színtér bal oldalára, majd ott feldarabolom oszlopait. Rendben, de ehhez a képlethez előzőleg tudnom kell a táblaszámot, hiszen attól függ, hogy mennyivel is kell balra tolnom. Ez nem probléma, mert egyébként is kiszámolnánk. Így az oszlopszám képlete:

$$col = floor(x-(128+cubeno*288)/64);$$

ahol *cubeno* az adott tábla száma. Ahogy látszik is, először 128 pixellel balra toltuk, mert akkora a rés az első tábla és az ablak széle között. Ezután, ha nem az első tábláról van szó, akkor még táblaszám*(96+192)-vel kell eltolni, hogy a bal szélére kerüljön a színtérnek. Így már csak ezt kell feldarabolni 64 pixelenként, azaz egy négyzet szélességenként.



3.2.1.3.4. ábra: Színtér a képzeletbeli transzformáció után, és a feldarabolással

Így készen is volnánk, már képesek vagyunk meghatározni egy adott négyzet helyét az adatszerkezetünkben. Ezután csak annyi volt a dolgom, hogy ellenőrizzem minden kattintáskor, hogy él-e az adott tábla, nincs-e még beikszelve az adott hely, illetve hogy az emberi játékos köre van-e. Ezeket a Tervezés és megvalósítás részben megtárgyalt *players_turn*, illetve *tables* változókkal már egyszerű munka volt. Az előbbi karbantartásáról a *step* esemény gondoskodik úgy, hogy minden lépés után megváltoztatja azt, míg az utóbbiéról az *scr_check_dead* függvény, amely megnézi, hogy lépésünkkel sikerült-e két X mellé egy harmadikat tennünk, azaz „megölni” a táblát.

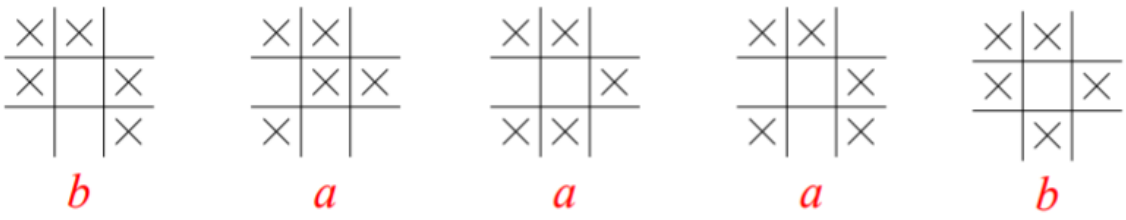
- Probléma: Hogy legyen implementálva az algebrai megközelítésből adódó módszer? Milyen keresési módszerrel találjuk meg egy adott táblaálláshoz tartozó szimbólumot azon hosszú listából? Hogyan és milyen típusú változókkal történjen ezek összeszorzása, majd egyszerűsítése?

  Megoldás: Amikor ehhez a részhez értem az implementációban, nagyon el kellett gondolkodnom, mivel tudtam, hogy ha ennél a módszernél nem törekszek eléggé a hatékonyságra, akkor az roppantul lehúzza programom gyorsaságát, így csökkentve minőségét. Elvárható, hogy a közepes és a nehéz gépi játékos tudjon előre gondolkodni: tekintse át a lehetséges lépések következményeit, hogy minél jobb döntést hozhasson. Ehhez viszont egy szimulációt kell majd neki elindítania, ahol lemásolja az egész adatszerkezetet magának, majd abban próbál valamilyen szempontok alapján döntést hozni jelenlegi helyzetében. Ha egy adott

szituációban ezt minden lehetséges lépésre megteszi, hogy tudjon mérlegelni egyes esetek között, akkor borzasztóan sokszor le fog futni ezen keresési algoritmus. Így ha már egy picit gyorsabb módszert találok, akkor sokkal hatékonyabb lesz programom.

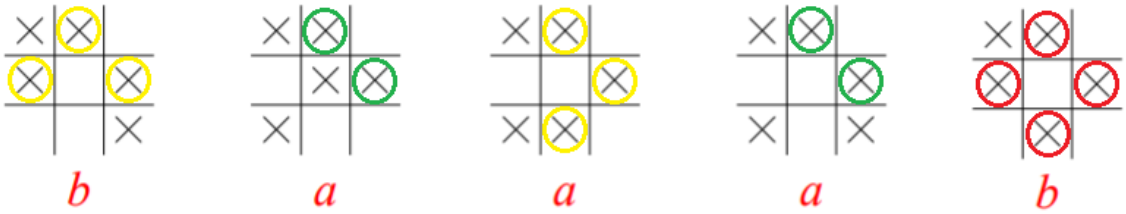
Tehát a feladat egy hosszú, különleges szimbólumokból álló listából minél gyorsabban megtalálnom a megfelelőt. Igaz hogy 102 darab lehetőségünk van, viszont ezek izomorfok, azaz igazából sokkal több van a listán, mint amennyit látunk. Azt találtam ki, hogy az input különböző tulajdonságai alapján minél kevesebb lépéssel addig kell szűkíteni a listát, amíg meg nem találjuk egyértelműen a keresett jelet. Így nem csak azt a problémát oldjuk meg, hogy gyors legyen a módszer, hanem azt is, hogy ha a bemeneti tábla tulajdonságaira alapozzuk keresésünk szűkítési kritériumait, akkor úgy is megtaláljuk, ha a listában éppen a tükörképe van csak benne. Tehát olyan attribútumokat kell keresni, amelyek egy forgatás hatására is megmaradnak, például van-e a tábla közepén X vagy sem. Rendben, akkor készen is vagyunk elméletben, de gyakorlatban mik legyenek ezen kritériumok? Úgy gondolom, hogy egy elég egyedi módszer ad hoc megoldást kívánt, így fogtam a hosszú listát és elkezdtem keresni az összefüggéseket ebből az új szemszögből. Fontos megjegyeznem, hogy nagyon sok táblahelyzet halott táblához tartozik. Ezekhez mindig az 1-es számot rendeljük, így velük nem kell foglalkoznunk. Máris nagyon sokat szűkítettünk, így sokkal hatékonyabban fog működni, de még korántsem végeztünk. Első és legésszerűbb csoportosítás a táblán lévő X-ek száma szerinti. Így hat darab halmazba tudjuk szervezni elemeinket, mivel hét X-et már nem tudunk úgy elhelyezni, hogy ne „öljük” meg táblánkat, sőt, hatot is csak úgy tudunk elhelyezni „előre”, hogy ha az egyik átlóból valamelyik üresen marad.

Példaként részletezem az öt darab X-et tartalmazó állások csoportosítását, a többi hasonlóan történt.



3.2.1.3.5. ábra: Fennmaradt táblák két szűrés után. Kritériumok: „élő” tábla legyen, öt darab X-el

Egyik fontos tulajdonság a középső oszlopban, illetve sorban lévő X-ek száma, a centrális X-et nem beleszámolva. Ezt a kritériumot nagyon sokszor használtam a szűkítésekben, így ez lett az egyik, amelyet egy *scr_count_cross(int)* nevű függvénybe szerveztem. Argumentuma a vizsgált tábla sorszáma. Az adott példánkban visszatérési értéke 2, 3 vagy 4 lehet. Ha 4-et számolt, akkor már készen is vagyunk, „*b”* szimbólumot állítunk be. Ha hármas számmal tér vissza, akkor sajnos még egyszer szűrnünk kell, méghozzá azon tulajdonság alapján, hogy van-e valamelyik átló két szélén X.



3.2.1.3.6. ábra: A táblák az scr_count_cross(int) segédfüggvény szűrése után.

Ha ez teljesül, akkor a keresett szimbólumunk megint a „*b”*, különben az „*a”*. Már csak azt kell megvizsgálnunk, hogy mi van, ha a 2-es számmal tér vissza a számoló függvényünk. Ebben az esetben igaz két esetünk van, de mindkettőhöz ugyanaz a jel tartozik, így az „*a”* szimbólumot rendeljük táblánkhoz. Készen is vagyunk az öt X-et tartalmazó táblákkal. Ugyanígy haladva tovább, kreatívan kiválasztva az egyes kritériumokat, végül képes voltam az *scr_set_symbol(int)* függvényt megírni, amelynek befejezésül hat darab segédfüggvénye lett, melyek az egyes csoportosítási tulajdonságokat ellenőrzik. Ezeket neveztem a tervezés során egyszerűen *scr_search_symbol(int)*-nak. A többi tulajdonságot vizsgáló függvény, az X-eket számlálón kívül:

1. *scr_count(int):* Visszaadja az argumentumnak megadott táblán lévő X-ek számát.
2. *scr_edges(int):* Megszámolja a tábla csúcsaiban elhelyezkedő X-eket.
3. *scr_count_empty_rows(int):* Az üres sorok és oszlopok számát adja vissza. Nem vizsgálja a középsőket.

4. *scr_count_empty_rows(int):* Ugyanazt teszi, mint az előző, de az összes sort és oszlopot vizsgálja.

5. *scr_has_lonelyedge(int):* Azt vizsgálja, hogy az argumentumnak adott sorszámú táblának van-e „magányos” X valamely csúcsában, azaz olyan beikszelt négyzet valamely csúcsában, amelynek egyik szomszédja sincsen beikszelve. Logikai típust ad vissza.

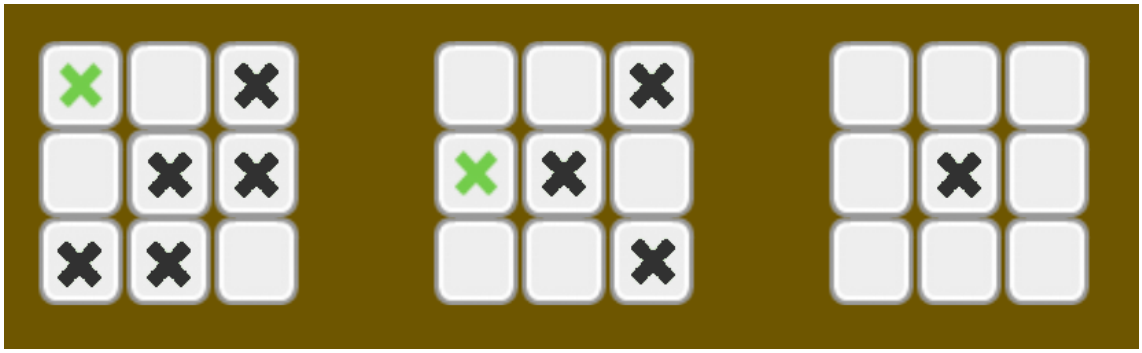
Rendben, most mindegyik táblához képesek vagyunk megtalálni a hozzá tartozó szimbólumot, de hogy szorozzuk össze, majd egyszerűsítsük le a kifejezéseket, ha három szöveg típusú változóban vannak szorzataink? Először is összekonkatenáljuk a szimbólumokat, hogy egy változóba tudhassuk őket. Így lesz már egy összegzett kifejezésünk, de még nincs leegyszerűsítve és a szorzatban lévő tényezők sincsenek sorrendben. Ekkor kell átkonvertálnunk egy számlálótömbbe a szövegtípusú változónkat a következő módon: a tömb első eleme az *„a”* szimbólumok számát jelzik, második a *„b”*-két és így tovább a tömb negyedik eleméig, amelyik a *„d”* jelek számosságát fogja tárolni. Ekkor elkezdhetjük redukálni ilyen formában a kifejezést. Ez úgy fog történni, hogy az egyszerűsítési szabályokat az egyes tényezők számával hozzuk összefüggésbe. Például ha egyik szabályunk a $b^2c = c$, akkor a tömb második elemét kettővel csökkentjük, hiszen végül is e képlettel eltüntetünk két „*b*”-t a kifejezésünkből. Ilyen módon implementálva a redukálási szabályokat és addig végezve őket, míg már nem tudunk többet használni közülük, befejezésül megkapjuk a végső kifejezést, amit már csak vissza kell alakítanunk szövegtípusba ugyanazzal a módszerrel, mint ahogy átalakítottuk, csak fordítva végrehajtva. Így még sorrendbe is raktuk a tényezőket a végső kifejezésben, amivel a függvény végén visszatérünk.

- Probléma: Mik legyenek a gépi játékosok taktikái? Mindegyik használja majd az algebrai módszert? Ha nem, akkor mit? Mennyire legyen erős a legnehezebb gépi játékos? Tökéletes játékra törekedjen mindenképp, vagy belefér, hogy adjon néha előnyt ellenfeleinek?

  Megoldás: Már a játék tervezési fázisában úgy gondoltam, hogy a három gépi játékosból csak a leggyengébb ne használja az algebrai módszert. Azt szerettem volna elérni ezzel, hogy a könnyű módon működő ellenfél meghozza a kedvét a

játékosnak és állítsa gondolkodását e játék megnyerésére, taktikák kifejlesztésére. Így ez a gép véletlenszerűen választ egy helyet, ahova lépni fog minden körében mindaddig, amíg legalább két tábla „él”. Ha ez már nem áll fenn, akkor taktikát vált, és elkezd egy picit jobban gondolkodni, így az utolsó „élő” táblán már törekszik a nyerésre. Tehát már nézi azt is, hogy ne ő rakja ki a három X-et előbb egy vonalban. Ezzel a gépi taktikával az volt a tervem, hogy a kezdő játékosok képesek legyenek elsajátítani győzelmük megalapozását addig, amíg csak egy tábla marad. Nagyon fontos szempontja a játéknak az, hogy ha már csak az utolsó táblára rakhatunk, akkor mi álljunk nyerő helyzetben, hiszen ott dől el a játszma kimenetele.

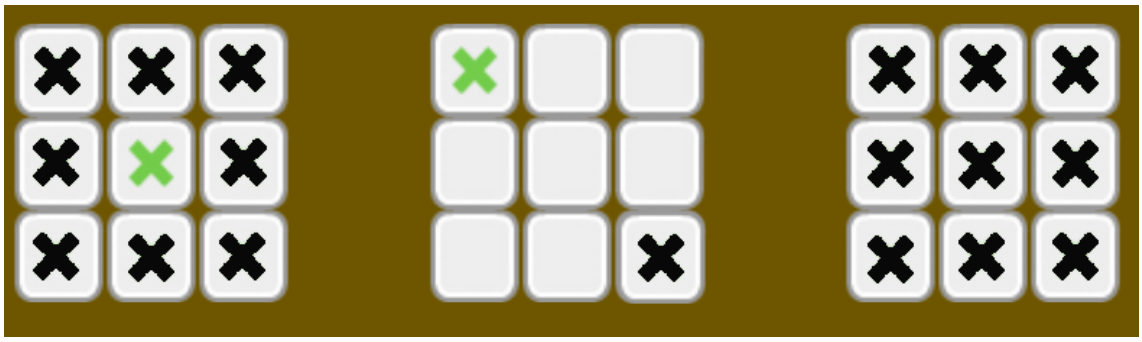
A közepes szintű gépi játékos már használja a szimbólumokkal kapcsolatos függvényeket, így próbál tökéletesen játszani ellenünk. Taktikája a következő: leszimulálja egy saját változóban minden lépés kimenetelét és ha vannak olyan helyek, ahova ha lép, akkor nyerő pozícióban lesz, akkor ezen helyek közül kiválaszt egy véletlenszerűt. Ha nincs ilyen lépésre lehetősége, akkor neki teljesen mindegy hova rakja az X-et. A jó lépésnél a véletlenszerűségre azért volt szükség, mert különben nagyon hasonlóak lennének a játszmák. Ezzel a módszerrel nagyon sok szituációba keverhetjük magunkat, így képesek leszünk megtanulni egyre több motívumot, trükköt a játékhoz. Itt ütköztem bele még abba a problémába is, hogy ha ezzel a taktikával engedjük a gépi játékost, hogy kezdjen, akkor szó szerint verhetetlen. Nem vagyunk képesek visszavenni az előnyt a játszam folyamán, ha a kezdő fél tökéletesen játszik. Ez alapvetően nem nagy hibája egy játéknak, mert ehhez minden trükköt és taktikát ismerni kell. Ezért hagytam ki ebből a szoftverből a kezdési lehetőség átruházását.



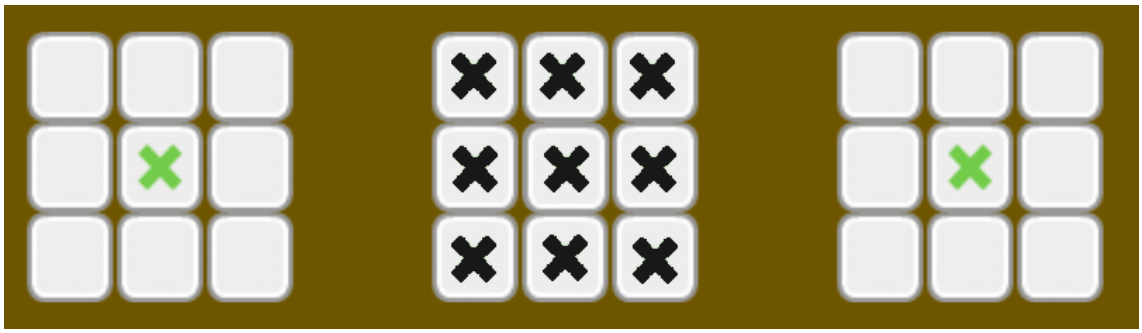
3.2.1.3.7. ábra: Egy játszma adott állásában. Zölddel: valódi X-ek, feketével: jó lépések, melyekből véletlenszerűen választ a közepes gépi játékos

Úgy gondoltam, hogy miután már magabiztosan megveri valaki a közepes gépi játékost, akkor tovább léphet a következő, nehéz szintre. Ehhez viszont az kell, hogy megtanuljon az ember tökéletesen játszani, hiszen ha már egyszer hibát követünk el, akkor a gép átveszi az irányítást és ő nem téveszt. Így a nehéz szintet elérőknek olyan taktikát kellett kigondolnom, amit még tökéletes játék mellett is nehéz megverni. Ahogy az előző, 3.2.1.3.7-es ábrán is lehet látni, egy adott helyzetben több jó lépés is létezik. Így logikus volt azt a taktikát implementálnom, amikor a gép arra törekszik, ha nem tud nyerő helyre rakni X-et, hogy miután rakott, nekünk minél kevesebb lehetőségünk legyen megtartani előnyünket. Ebből adódóan mondhatjuk, hogy ez a gépi játékos két lépéssel gondolkodni előre, azaz eggyel többel, mint a közepes. Ehhez az előző szintűhöz hasonlóan lefuttat egy szimulációt minden lehetséges lépésre, de ezekben nem csak a kijött végső szimbólumot vizsgálja, hanem azt is, hogy abból a helyzetből hány jó lépési lehetőségünk lenne. Ezeket összegyűjti, majd minimalizálja. Fontos megjegyeznek, hogy ha több minimumérték is van, akkor azok közül véletlenszerűen választ, nehogy a sok ismétlődéstől monotonná váljon a játék. Egy példában vessük is össze, hogyan is néz ki a gyakorlatban ez a különbség.

Mondjuk, hogy én az első számú táblának a közepére teszem első X-emet. Ez egy nagyon jó kezdés, így nyerő szimbólum jön ki lépésem után. Ekkor lássuk, hogy hova lép a nehéz, illetve közepes gépi játékos, és hogy ezek után hány lépési lehetőségem van ahhoz, hogy megtartsam előnyömet.



3.2.1.3.8. ábra: A közepes gépi játékos válasza egy jó kezdésre. Feketével láthatjuk azon helyeket, amelyekkel folytathatjuk tökéletes játékunkat



3.2.1.3.9. ábra: A nehéz gépi játékos válasza egy jó kezdésre. Feketével láthatjuk azon helyeket, amelyekkel folytathatjuk tökéletes játékunkat

Könnyen láthatjuk, hogy közepes szinten összesen 18, míg nehezen 9 hely marad optimális a folytatásra. Ironikus módon úgy gondolom, hogy ennek ellenére a közepes gépi játékos legyőzése több koncentrációt és trükköt igényel, mivel ellene játszva a teljesen véletlenszerű lépések miatt sokkal több szituáció alakulhat ki, amelyre mindig tudnunk kell a helyes megoldást. Nehéz módot játszva szinte mindig ugyanazon helyzetekbe kerülünk. Ha ezekre begyakoroljuk a jó lépést, akkor egy idő után könnyebb lesz megverni, mint a közepest. Megjegyzem, hogy mivel egy helyen szerettem volna tudni az összes gépi játékos kódját, így három blokkba szerveztem lépésüket segítő eszközeiket. Emiatt úgy lehet elképzelni a kódot, hogy:

ha (közepes vagy erős az ellenfél) {

//előre gondolkodik az algebrai módszer segítségével

} egyébként ha(erős és nem talált eddig jó lépést){

//még egy lépéssel előre gondolkodik és biztosan talál egy legideálisabb //lépést a nem optimálisak közül

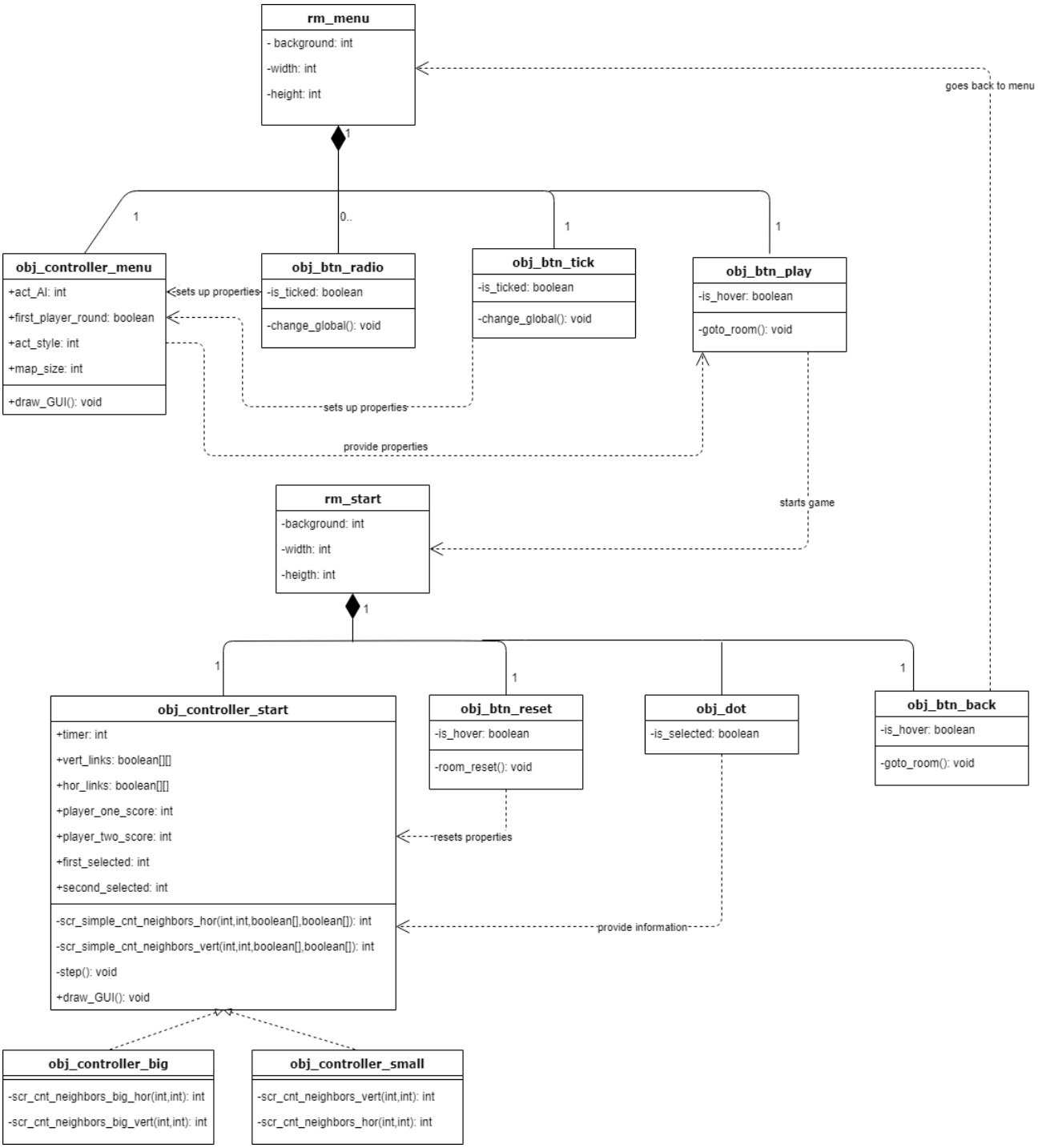
} egyébként ha(közepes vagy gyenge és nem talált eddig jó lépést){

//véletlenül választ egy helyet

}

3.2.2 DotsandBoxes

3.2.2.1 Tervezés és megvalósítás



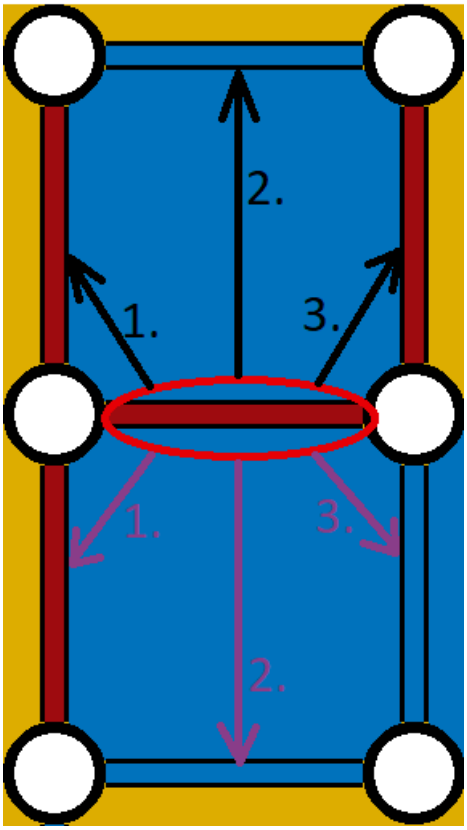
3.2.2.1.1. ábra: A DotsandBoxes játékszoftver UML diagramja

Könnyen észrevehetjük, hogy ez az UML diagram nem sokban különbözik előző program tervévtől. Ez azért van, mert próbáltam ugyanazon gondolatmenettel, rendszerben megtervezni a két játékszoftvert. Ezért e diagram kifejtésében néhány, már szerepelt, mezőt vagy objektumot ki fogok hagyni, mert ugyanazt a szerepet töltik majd be itt is, mint amit a 3.2.1.2-es részben már leírtam. Fontos megjegyeznem azt is, hogy az *obj_link* illetve *obj_point* objektumok nem kerültek bele a tervezetbe, mert funkcionalitásuk csak is dekoratív. Tehát használatuk és szerepük a program működésében implicit. Csak azért csináltam belőlük is példányokat, hogy használhassam a fejlesztői környezet beépített, sok mindent megkönnyítő metódusait. Ezek kifejtése a következő, Problémák és megoldásaik részben olvashatók.

1. *rm_menu:* 36. oldal 1-es pont.
2. *obj_controller_menu:* Mint ahogy az előző játéknál is láthattuk, a menünek érdemes létrehozni egy irányítóobjektumot, melyben összegyűjtjük az adatokat, hogy egy helyen tudhassuk őket. Ezen példány két adattaggal több, mint a Trictactoe-ban, mivel ebben a szoftverben képesek vagyunk már pályát is választani és kezdési lehetőségünket átruházni gépi ellenfelünknek. A többi adattag leírása a 37. oldal 2-es pontjában.
    a. *map_size:* Egész típusú nyilvános változó. Értéke lehet 0 vagy 1. Előbbi eset a 4x4-es pályát szimbolizálja, utóbbi az 5x5-öst.
    b. *first_player_round:* Logikai típusú nyilvános adattag. Igaz esetén az emberi, míg hamis esetén a gépi játékos kezdi a játszmát.
3. *obj_btn_radio:* 37. oldal 3-as pontja.
4. *obj_btn_tick:* Ez a menünkben elhelyezkedő jelölőnégyzet objektuma. Ugyanúgy, mint az *obj_btn_radio*, ez is az irányítóobjektumnak szolgálja az információkat úgy, hogy megváltoztatja annak nyilvános adattagjait saját példányváltozójától függően.
    a. *is_ticked:* Logikai típusú mező. Ezen változó, ha hamis, akkor a példány az üres textúráját állítja be magának, egyébként azt, amelyikben egy X található. Értékét kattintásra változtatja.
    b. *change_global():* 38. oldal 3-as pont b. része.

5. *obj_btn_play:* 38. oldal 4-es pontja.
6. *rm_start:* 38. oldal 5-ös pontja.
7. *obj_controller_start:* A játéktér irányítóobjektuma. Leírása nem különbözik a 38. oldal 6-os pontjában megfogalmazottaktól, de adattagjai többsége részletezést igényel.
    a. *timer*: 39. oldal 6-os pont a. része.
    b. *vert_links*: Logikai értékeket tároló nyilvános kétdimenziós adattag. Ebben a változóban vannak a játéktér függőleges vonalai. A kicsi, 4x4-es pályán 5 sorból és 6 oszlopból, míg a nagyon, tehát az 5x5-ösön 6 sorból és 7 oszlopból áll. Ha az y. oszlopban, x. és x+1. sorokban lévő pontokat kötöm össze vonallal, akkor a *vert_links[x,y]* elem hamisról igazra vált.
    c. *hor_links*: Logikai értékeket tároló nyilvános kétdimenziós tömb. Hasonlóan az előző változóhoz, ebben a vízszintes vonalakat raktározzuk. A kicsi pályán 6 sorból és 5 oszlopból, míg a nagyon 7 sorból és 6 oszlopból áll. Ha az x. sorban, y. és y+1. oszlopban lévő pontok közé húzok vonalat, akkor a *hor_links[x,y]* elem hamisról igazra vált.
    d. *player_one_score*: Egész típusú nyilvános mező. Gépi ellenfél ellen ez az emberi játékos pontszáma. Értéke a pontszámokat jelző textúrák számosságától függ.
    e. *player_two_score*: Egész típusú nyilvános adattag. Gépi játkos ellen játszva ez ellenfelünk pontszámát tartja nyilván. Értéke attól függ, hogy hány gépi ellenfelünk pontszámát szimbolizáló textúra van a játéktéren.
    f. *first_selected*: Az elsőnek kiválasztott pont azonosítója. A játék alapmechanikáinak egész típusú segédváltozója. Ennek és ezutáni adattagókkal képesek vagyunk könnyedén ellenőrizni egy lépés helyességét, azaz hogy van-e már azon a helyen vonal vagy éppen egymás melletti-e a két pont, melyet össze szeretnénk kötni.
    g. *second_selected*: A másodiknak kiválasztott pont azonoítója. Egész típusú változó. Funkcionalitása ugyanaz, mint az előzőnek.

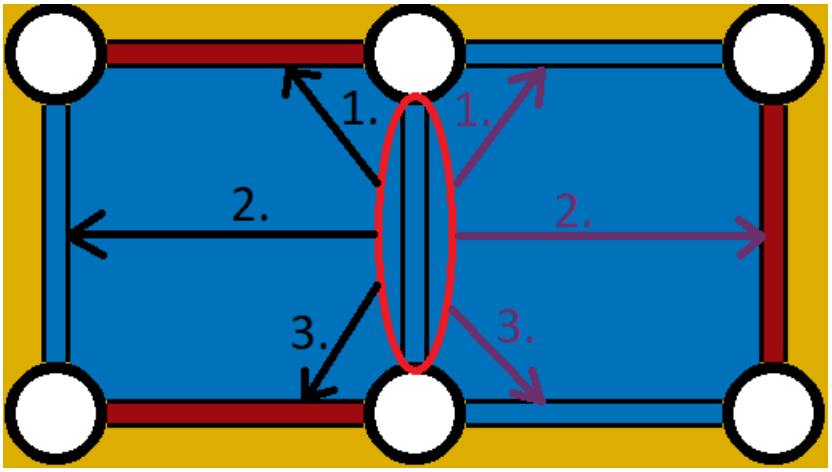
h. *scr_simple_cnt_neighbors_hor(int,int,boolean[][], boolean[][])*: Privát számoló függvény, mely egy egész számmal tér vissza 0 és 4 között. Argumentumai közül az első kettő az éppen létrehozott vízszintes vonal koordinátái. Harmadik argumentuma egy függőleges vonalakat, míg negyedik egy vízszintes vonalakat tároló kétdimenziós tömb. Nem biztos, hogy mindig a *vert_links* és *hor_links* lesznek ezek, mivel sokszor le fogja másolni játékterünket a gépi játékos, hogy szimulálhasson helyzeteket. A függvény segítségével képesek vagyunk megszámolni, hogy az argumentumba adott koordinátákon lévő vízszintes vonalnak az utolsó két tömbbel szemléltetett játéktéren hány „szomszédos” vonala van. Itt azokra gondolok, melyekkel négyzetet zárhat be. Ezeket összeszámolja, de külön az alatta lévőket, külön a felette lévőket és ezek maximumát adja vissza. Ha mindkét szám 3, akkor 4-el tér vissza, ami azon szituációt fogja jelenteni, hogy a vízszintes lépésünk két négyzetet is bezárt, így két ponttal gazdagított minket.



3.2.2.1.2. ábra: A pirossal bekarikázott vízszintes vonal "szomszédos" vonalai. Feketével a felette, míg lilával az alatta lévők.

i. *scr_simple_cnt_neighbors_vert(int,int,boolean[][],boolean[][]):* Privát számoló függvény. Hasonlóan az előzőhöz, ez is egy vonal szomszédjait számlálja meg az utolsó argumentumokba adott játéktérben, csak e vonal

ebben az esetben függőleges. Ekkor nem a felette, illetve alatta lévő szomszédokat, hanem a tőle balra és jobbra lévők létezését vizsgálja meg.



3.2.2.1.3. *ábra: A pirossal bekarikázott függőleges vonal "szomszédjai". Feketével a baloldali, míg lilával a jobboldali.*

j. *step():* Privát eljárás. Ezen segítségével váltakoznak a körök, gondolkodik a gépi játékos és teszi meg lépését. Gyakorlatban ez természetesen a *step* esemény.

k. *draw_GUI(): //ezt ki kell venni az UML-ből*

8. *obj_controller_big:* Az irányítóobjektum egyik speciális leszármazottja. Ez a példány a nagy pályán jön létre, mivel más méretű tömbökkel dolgozik majd két függvénye.

   a. *scr_cnt_neighbors_big_hor(int,int):* Privát, egésszel visszatérő függvény. Hasonlóan az *scr_simple_cnt_neighbors_hor*-hoz, ez is az argumentumba adott számokon, mint koordinátákon lévő vízszintes vonal „szomszédjait” számolja meg, azzal a nagy különbséggel, hogy e függvény minden esetben az éppen folyó játszma játékterét szimbolizáló (*hor_links, vert_links*) kétdimenziós tömbökön vizsgálódik. Ez azt eredményezi, hogy ezzel nem tudunk szimulációkban számolni. Ennek az az oka, hogy ezen alprogrammal csak akkor szerettem volna tevékenykedni, ha élesben valamely játékos lépett. Így ez a függvény lefutás után, ha úgy számolt, akkor még a pontokat szimbolizáló textúrárat is létrehozza a megfelelő helyeken.

b. *scr_cnt_neighbors_big_vert(int,int):* Ez a függvény ugyanazon funkciókat látja el, mint az előző, csak függőleges vonalak esetén.

9. *obj_controller_small:* Az irányítóobjektum speciális leszármazottja, melyet csak a kis pályán hozunk létre. Ez megint azért fontos, mert függvényei kisebb tömbökön számolnak majd.

    a. *scr_cnt_neighbors_small_hor(int,int):* Teljesen ugyanaz, mint az *scr_cnt_neighbors_big_hor,* csak kisebb, a 4x4-es pályához tartozó tömbökön vizsgálja az argumentumnak adott vízszintes vonal szomszédjait.

    b. *scr_cnt_neighbors_small_vert(int,int):* Ugyanaz, mint az előző függvény, csak függőleges vonalra tervezve.

10. *obj_btn_reset:* 42. oldal 8-as pontja.

11. *obj_dot:* A pályákon lévő pontok objektuma, melyek közé vonalat lehet húzni. Információkkal látja el az játszmaszoba irányítóobjektumát. Ezek a saját x és y koordinátái mellett a példányazonosítója is, melyeket majd az *obj_controller_start* példány *first_selected* vagy éppen a *second_selected* változói fogják megkapni értékül.

    a. *is_selected:* Logikai típusú példányváltozó. Ennek segítségével oldottam meg a játék azon alapmechanikáját, amellyel ki tudunk jelölni egy kattintással egy pontot, majd ha elkattintunk, akkor már nincs kijelölve. Ez a példányváltozónak hamis a kezdőértéke, majd ha rákattintunk igaz lesz. Utóbbi esetben az *obj_dot* adott példányának textúrája fekete színnel telítetté változik, ezzel jelezve azt, hogy éppen azt választottuk ki.

12. *obj_btn_back:* 42. oldal 9-es pontja.

3.2.2.2 Akadályok és megoldásiak

Hasonlóan az előző játékhoz, ennél is fontosnak érzem bemutatni az egyes, nem szokásos akadályok megoldási módszereit.

- Probléma: Mik lesznek a fontos információk, amelyek alapján tároljam a játék állását? Ezt milyen adatszerkezetben tegyem?

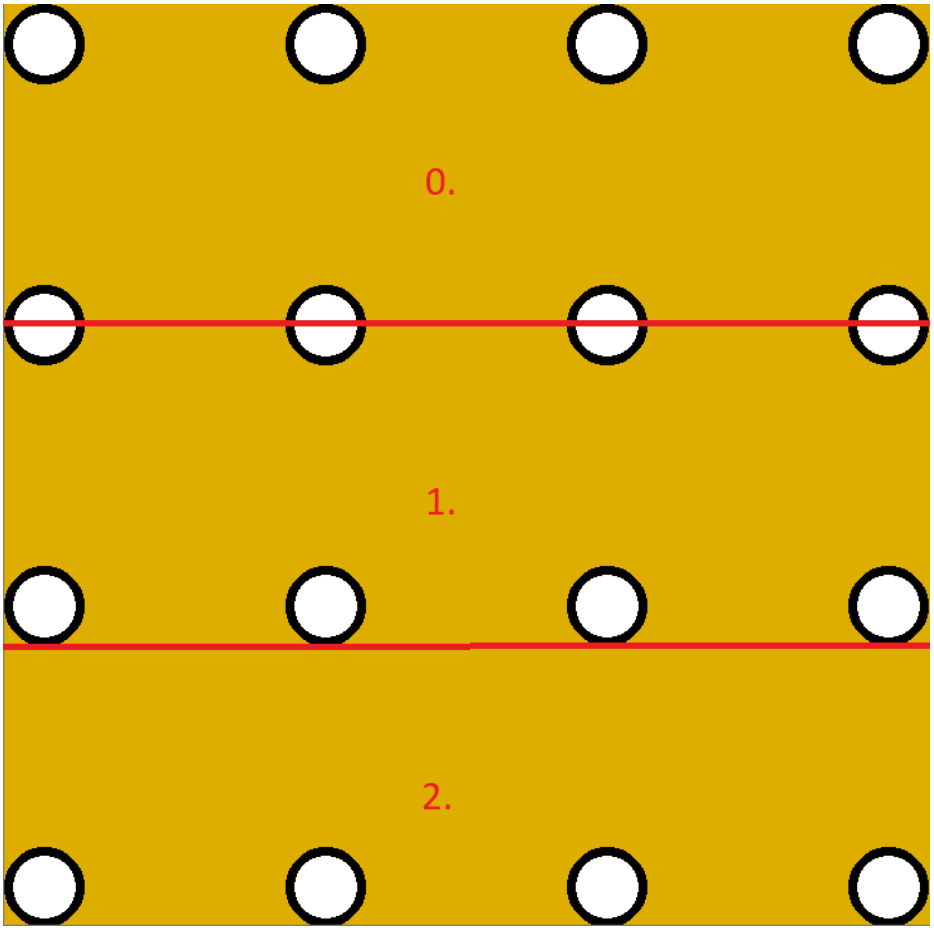
Megoldás: Persze egyértelmű volt, hogy a pontokat összekötő vonalakat kell tárolnom, viszont nagyon sokat kellett gondolkodnom, hogy ezt hogy tegyem. Ezen töprengésben eszembe jutottak jobb-rosszabb ötletek, egy ponton még azt is mérlegeltem, hogy esetleg a vonalak helyett az azok által bezárt négyzeteket tárolja majd a program. Ekkor olyan tömböt képzeltem el, amely például a 4x4-es táblán kétdimenziós, 3 sorból és 3 oszlopból áll. Viszont ekkor nem logikai értékeket tárolt volna, hanem a négyzeteket határoló vonalakat valamilyen módon. Például minden elem lehetett volna egy logikai tömb, melynek első eleme igaz, ha a négyzet felett van vonal. Ezt a logikát folytatva lehetne egy adattárolót konstruálni. Ezzel a megoldással az volt a problémám, hogy amikor belegondoltam milyen módon gondolkodjanak a gépi játékosok, akkor azokat a módszereket nem tudtam volna oly egyszerűen implementálni. Ezen kívül a gép a legkönnyebben a logikai változókkal számol. Így maradtam inkább az egyszerűbb, logikai elemeket tároló kétdimenziós tömbnél. Ennek a döntésnek viszont az lett a következménye, hogy két tömbre lesz szükség: egy a vízszintes, egy a függőleges vonalakhoz.

Megjegyzem, amikor eszembe jutott az, hogy a vonalak szomszédjait kell majd megvizsgálni minden lépés után a játékban, hátha négyzetet zár be, akkor egy átlagos méretű tömbben mindig ellenőrizni kéne, hogy netán a szélén vagyunk-e valahol, mert például ha a jobb szélén vagyunk, akkor nem vizsgálhatjuk a bevitt függőleges vonal jobb szomszédjait, hiszen nincsenek. Ez túl műveletigényes és bonyolult lett volna, így kitoldottam, azaz keretet adtam a tároló tömböknek, végül 5x6-os illetve 6x5-ös lett a kis pályán, 6x7-es és 7x6-os a nagyon.

Az előző játékhoz hasonlóan, gyorsan és könnyedén képes voltam e szoftvernek is az alapmechanikáit implementálni, játszmaszobáját berendezni. Ekkor „már csak” az volt a feladat, hogy minden lépéskor a megfelelő tömb elemét igazra állítsam. Először is el kell dönteni, hogy az adott lépésünk vízszintes vagy függőleges vonal. Ezt az akadályt is, hasonlóan az előző játékhoz, beépített változókkal és egy kicsiny koordinátageometriával oldottam meg. Így jönnek a képbe a *first_selected* és *second_selected* változók. Az előbbibe az elsőnek kiválasztott pontnak, utóbbiba a másodiknak kiválasztott pontnak az azonosítóját tárolom el. Ekkor, ha mindkét változó értéket kapott, azaz megvan, hogy mely pontok közé kíván az adott játékos vonalat húzni, akkor több dolgot is ellenőrizni

kell. Van-e a kettő között már vonal? A két pont egymás melletti? Két különbözőről van szó biztosan? Ezen feltételek eldöntéséhez újra a fejlesztői környezet beépített függvényeit hívtam segítségül. Első ilyen amit sokszor használta az *instance_position_meeting(x,y,obj)* logikai értékkel visszatérő alprogram. Első két argumentuma a koordináta, ahol ellenőrzi, hogy a harmadik argumentumba adott példány létezik-e. Második nagyon fontos beépített egész számmal visszatérő függvény a *point_distance(x1,y1,x2,y2)*. Az első két, illetve utolsó két argumentum egy-egy pontot határoznak meg. E pontok távolságát adja vissza ezen alprogram. Ezek után már könnyedén tudjuk ellenőrizni az adott lépés helyességét. Ezek után el kell döntenünk, hogy a vonal, melyet behúztunk függőleges vagy vízszintes, mivel ezen múlik tárolásának helye. A beépített változók ekkor sem hagynak cserben, mivel ha a két kiválasztott pontnak x koordinátája egyenlő, azaz first*_selected.x == second_selected.x,* akkor függőleges, egyébként vízszintes a vonalunk. Így már készen is állunk az adat bevitelére. De hogyan is állapítsuk meg egy lépés után a vonal helyét? A tömböt már tudjuk, most annak sorát és oszlopát kell kiszámolnunk egy pici koordináta geometriával. Ehhez ismernünk kell pontosan a pálya fontosabb méreteit. Két pont között 160 pixelnyi hely van, és egy pont mérete 64x64 pixel. Ekkor, hasonlóan az előző szoftverhez, megint a játéktér képzeletbeli feldarabolásával voltam képes a tárolók megfelelő elemének megváltoztatására, azaz a vonal elraktározására. Csak a függőleges esetet mutatom be, a vízszintes teljesen analóg módon működik, csak fordított koordinátákkal. Először is a sort kell meghatároznunk. Ehhez elég volt a pályát vízszintesen 256 pixelesével feldarabolni. Így a sor koordinátája függőleges esetben:

floor(link_pos_y/256)+1;

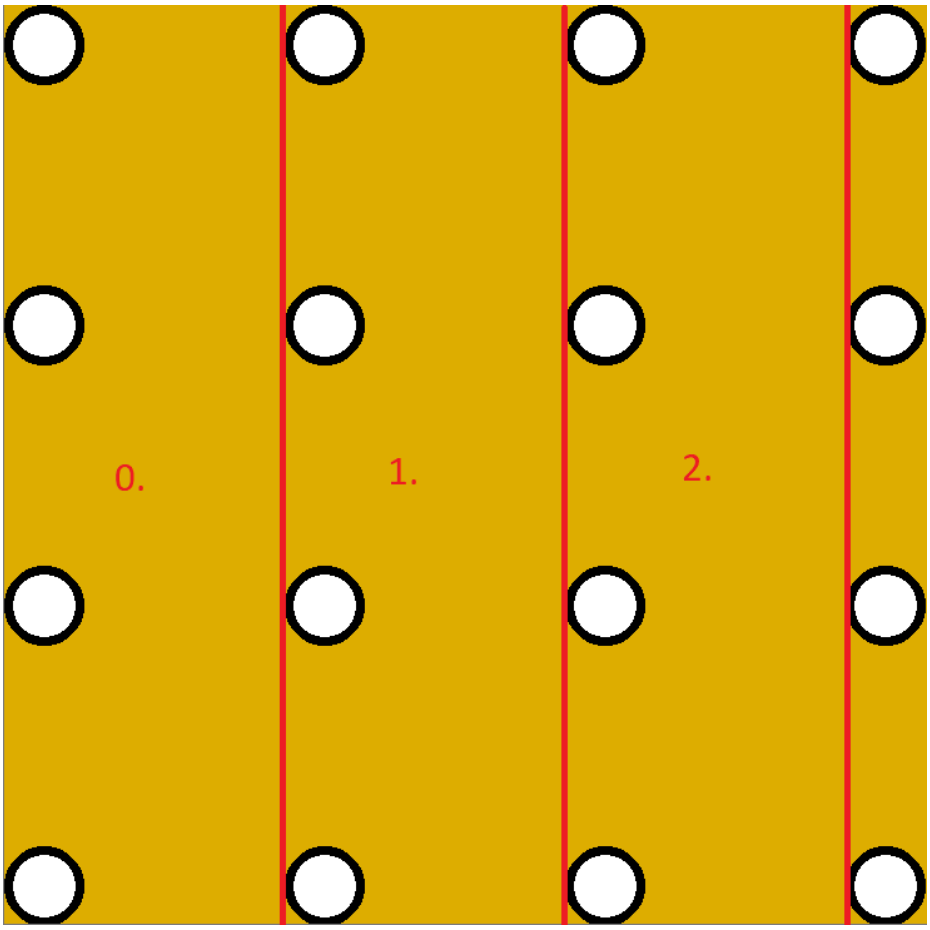


3.2.2.2.1. ábra: A sor meghatározása függőleges vonal behúzása esetén

, ahol a *link_pos_y* a két kiválasztott pont függőleges koordinátájának átlaga. Egyet azért kellett hozzáadni, mivel kerete van a tömbnek. Az oszlopot ehhez hasonlóan a

$$floor(first\_selected.x/224)+1;$$

képlettel kapjuk, ahol 224 pixelesével daraboltuk pályánkat függőleges vonalakkal képzeletben.



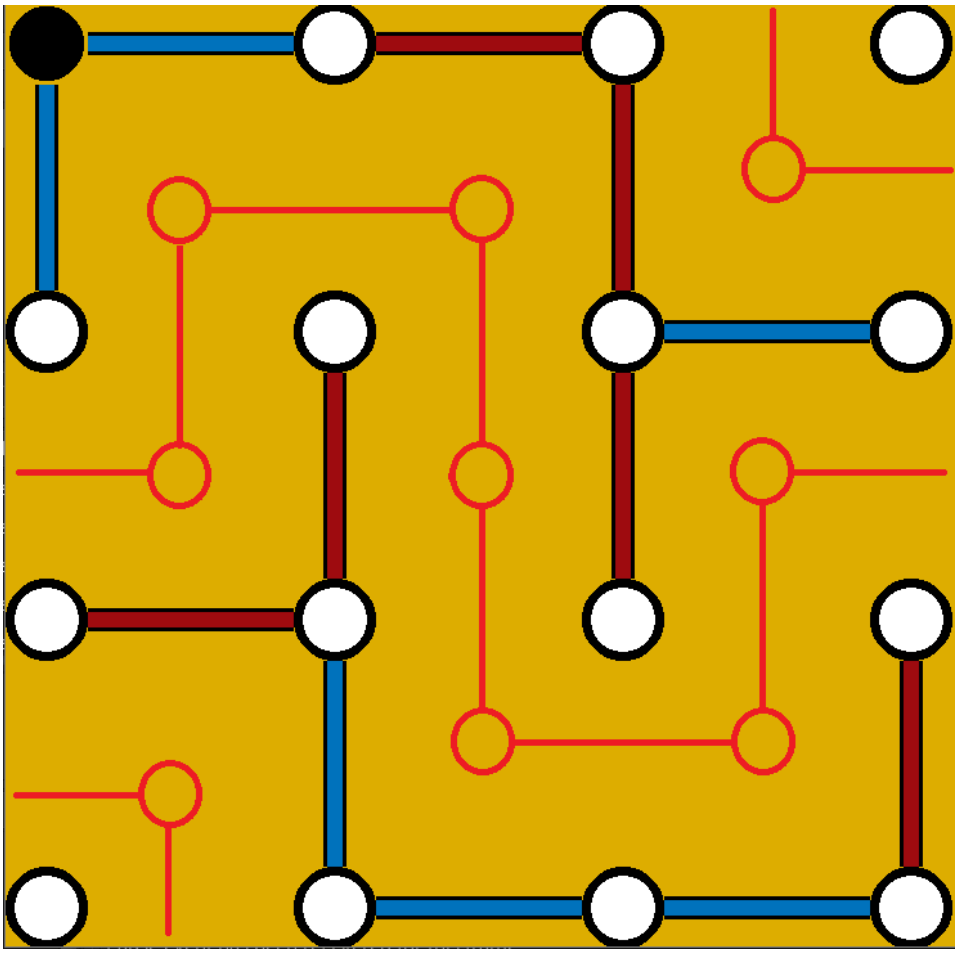
3.2.2.2.2. ábra: Az oszlop meghatározása függőleges vonal behúzása esetén

Ezek után készen is volnánk. Megjegyzem, hogy a nagyobb pályán annyi ezen feldarabolási módszereknek az eltérése ettől, hogy kevesebb pixelenként osztjuk fel a játékteret.

- Probléma: Mik legyenek az egyes gépi játékosok taktikái? Meg lehessen verni a legerősebbet is úgy, hogy hibázunk? Mennyit számít a nehézségben a kezdési lehetőség, illetve a pálya nagysága? Mit tudunk tanulni az egyes szintek elérésével?

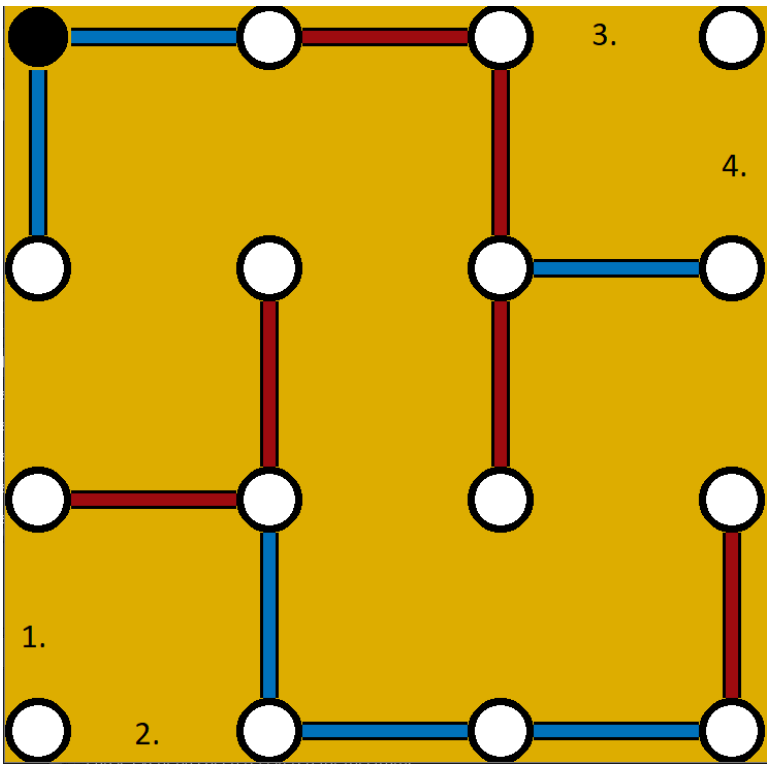
  Megoldás: Úgy gondoltam, hasonlóan az első játékhoz, ebben is a leggyengébb gépi játékos szimuláljon egy teljesen kezdőt. Lépjen mindig teljesen véletlenszerű helyre, kivéve, ha van négyzet, amit befejezhet egy vonallal. Minden körében végigfut a játéktéren és megszámolja az összes vonal szomszédjait az *scr_simple_cnt_neighbors_vert()* függvénnyel, amelyben az éppen folyó játszma tárolóit adjuk meg neki, azaz nem szimuláció, hanem élesben számol. Ekkor, ha talál olyat, aminek három szomszédja van, azaz ha azt behúzná, akkor pontot

csinálna magának, akkor azt választja, egyébként marad a véletlenszerű játéknál. E gépi játékos ellen megtanulhatjuk észrevenni a befejezetlen négyzeteket, jobban hozzászokhatunk a játéktérhez, elkezdünk különböző taktikákon gondolkodni, hogy ne adjunk ingyen pontokat az ellenfélnek. Közepes szinten már nehezebb dolgunk lesz. Ez az ellenfél már nem csak azt vizsgálja, hogy egy vonalnak három szomszédja van, hanem azt is észreveszi, hogy ha kettő. Ez azért fontos, mivel ha ilyen helyre lépne, akkor nekünk ingyen pontot adna, hiszen akkor valahol kialakulna három szomszédos, még be nem húzott vonal. Így e vetélytárs ellen elsajátíthatjuk a tökéletes játékot egy ideig. Ha mi sem adunk ingyen pontot, akkor egy idő után már nem marad lehetőségünk olyan helyre lépni, ahol utána ne tudna ellenfelünk pontot csinálni. Ekkor figyelhetjük meg először az úgynevezett folyosók kialakulását.



3.2.2.2.3. ábra: A folyosók kialakulása. Ha a piros vonalakat ebben a szituációban valamely játékos megszakítaná egy lépéssel, akkor azon folyosón lévő összes pontot begyűjtheti ellenfele. Ezek az üres piros körök.

Egy folyosó olyan részek összessége, melyek nincsenek elvágva, viszont határolva vannak vonallal és nyitott a két végük. Ezek sorsdöntő részei a játéknak, mivel a legjobb taktika ezek tudatos kialakítása, majd befejezése. Ezalatt azt értem, hogy ha ellenfelünk egy ilyen részbe húz akárhova vonalat, akkor mi végig be tudjuk gyűjteni az itt elhelyezkedő pontokat. Így mikor a közepes gépi játékos egy ilyen sorsdöntő szituációban véletlenszerű helyre lép, akkor picit kiábrándulunk belőle. De nem baj, hiszen már el is sajátítottuk a legjobb taktika alapját. A legnehezebb fokozaton annyi a különbség, hogy a sorsdöntő pillanatban nem véletlenszerűen, hanem logikusan dönt. Ez azt jelenti, hogy ha kialakulnak a folyosók és muszáj neki valamelyiket elvágnia, akkor a legrövidebbet fogja választani, hiszen akkor azt mi befejezzük, viszont ezek után nekünk muszáj a fennmaradtak közül valamelyiket, de az vagy ugyanolyan hosszú, mint amit nekünk adott a gépi, vagy hosszabb, így rosszul jövünk ki. Ezt egy szimulációval éri el versenytársunk. Lemásolja a játékteret, majd ezeken megnézi, hogy milyen szituációba keveredne minden egyes lépést kipróbálva. Egy ilyen próbalépés után megnézi, hogy hány pontot lenne képes begyűjteni és a legvégén ezen számokat minimalizálja. Ha több minimális van, akkor véletlenszerűen választ azok közül. Így képes a legkisebb áldozatot hozni egy rossz szituációban. Egy ilyen taktikás



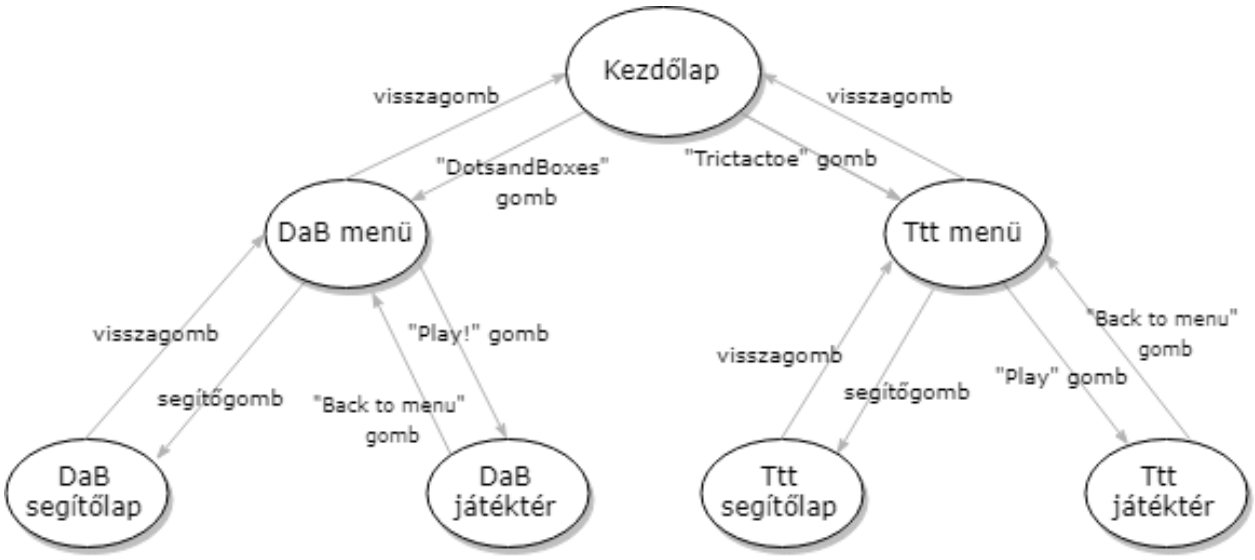
3.2.2.2.4. ábra: A nehéz gépi játékos által preferált helyek egy döntő helyzetben. A legkisebb áldozattal járó lépések helyei számokkal jelezve, ezek közül választ egyet véletlenszerűen.

játékos mellett képesek vagyunk nagyon magas szinten megtanulni játszani e játékot. Itt is megjegyzem, mint az előző játéknál, hogy a gépi játékosok kódjai egy helyen vannak. Ez azt jelenti, hogy minden eszköz, segítség, amivel lépésüket meg tudják határozni, az egyetlen eseményben van, csak különböző blokkokban.

3.3 Tesztelés

3.3.1 Felhasználói felület tesztelése

Tesztelnünk kell, hogy bármelyik ablakból el tudunk jutni akármelyik másikba a megfelelő **navigációs gombok** segítségével.



3.3.1.1. ábra: A program navigációs ábrája. Rövidítések: Ttt = Trictatoe, DaB = DotsandBoxes

Tesztelési terv: A navigációs gombok egy olyan sorozatát keressük meg, amellyel a leghatásosabban lehet tesztelni az ablakok közötti váltások helyességét.

A gombsorozat: „DotsandBoxes” gomb => segítőgomb => visszagomb => „Play!” gomb =>„Back to menu” gomb => visszagomb => „Trictactoe” gomb => segítőgomb => visszagomb => „Play!” gomb

Így minden ablakot bejártunk, leteszteltük a navigációs gombok működésének helyességét.

Ezen kívül tesztelnünk kell még mindkét játéknak a játékterén lévő elemeknek a szabályokból adódó különleges viselkedéseit. A **DotsandBoxes** esetén próbáljuk meg a következőket:

- kössünk össze két nem közvetlen egymás mellett fekvő pontot (például kettővel tőle jobbra lévővel)

- kössünk össze két olyan pontot, amelyek között már van vonal
- húzzunk be vonalat gépi játékos körében
- húzzunk be vonalat átlóban

Ezen eseteket végig próbálva azt tapasztaljuk, hogy a játék **egyszerűen eldobja az elsőnek kiválasztott pontot** és újat kell választanunk. Így előzi meg a szabályokkal ellentmondó hibákat.

A **Trictactoe** játékban próbáljuk megvalósítani a következőeket:

- tegyünk X-et egy halott (szürke) táblára
- tegyünk X-et valahova a gépi játékos körében
- tegyünk X-et olyan helyre, ahol már van

Ezen események után azt tapasztaljuk, hogy lépésünk haszontalan volt: **nem került X sehova**. Tehát nem lehet szabályokba ütköző lépést tenni.

3.3.2 Tictactoe mesterséges intelligencia tesztelése

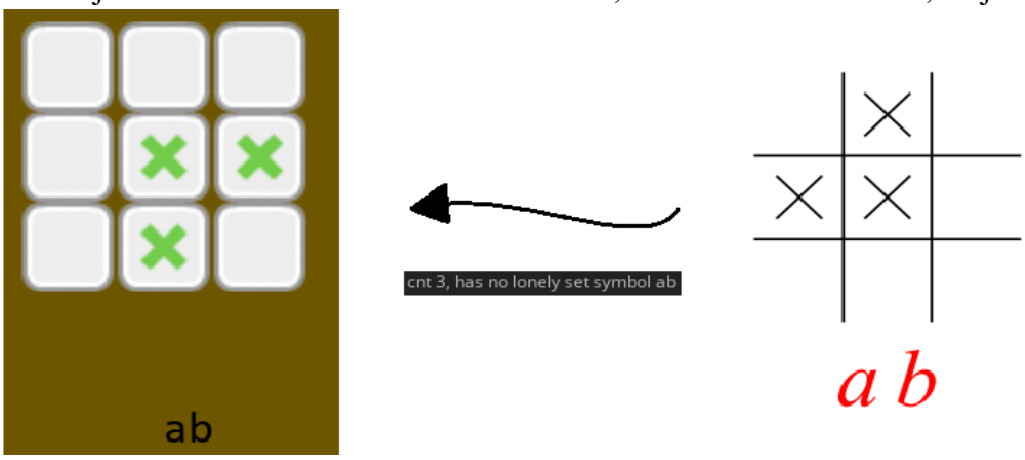
Tesztelési terv: A különböző állapotok változását a konzolba írt üzenetek alapján fogom leellenőrizni. Ezt a *show_debug_message(string)* beépített függvénnyel tudjuk megvalósítani, ahol a *string* egy szöveg típusú változó, melyet ki fog nekünk írni a konzolba. A Trictactoe játékban három fő működési területet kell alaposan letesztelni:

1. **Beállítások.** A főmenüben beállított paraméterek helyesen maradnak meg a szobaváltás után is.
2. **Algebrai módszer.** A táblákhoz rendelt szimbólumok helyesek. Az összesített szorzat megfelelően redukálódott.
3. **Gépi játékosok.** Gondolkodási menetük megfelel az adott szintjükhöz. Lépésük helyes-e az adott szituációkban. Minden alternatívát megfelelően vizsgál és értékel-e ki.

A **beállítások** helyességének vizsgálata roppant egyszerű feladat. Csak ki kell íratni a játéktér irányítóobjektuma keletkezési eseményekor az *act_AI* és *act_style* változókat. Kipróbálva különböző beállításokat a 3.2.1.4.1-es ábrán látható szövegeket olvashatunk a konzolban. Ezek segítségével könnyen látható lenne, ha valami nem helyesen lett volna beállítva.

3.3.2.1. ábra: Példák a konzolban látható ellenőrző szövegek játék indításakor

Az **algebrai módszert** az egyszerűség kedvéért nem a konzolban tesztelem, hanem a játéktérre rajzolom ki a táblák aktuális szimbólumát, az összesített szorzatot, majd csak



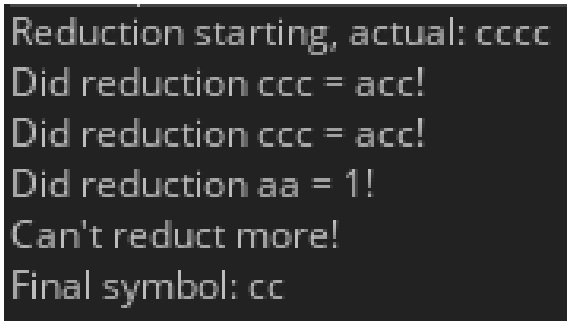
3.3.2.2. ábra: Példa egy szimbólum beállítására. A nyíl alatt a konzolban látható ellenőrző szöveg.

ezek után fogom a konzolban végig követni ezen kifejezés egyszerűsítésének folyamatát. A **szimbólumok helyességét** implementálás közben volt a legegyszerűbb ellenőrizni. Minden újabb csoportosítás és kritérium kódolása után leteszteltem, hogy jó szimbólumot ír-e a program az adott tábla alá. Később kezdtem a konzol segítségével azt is leellenőrizni, hogy nem csak véletlenül helyes-e a hozzárendelt kifejezés, azaz kiírattattam a keresési kritériumokat. Így ha az adott tábla nem felelt meg ezen tulajdonságoknak, akkor tudhatjuk, hogy baj van a programmal, egyébként minden rendben van.

E példában a konzolban látható szöveget úgy kell értelmezni, hogy három darab X van a táblán (*cnt 3*) és nincs „magányos” X valamely csúcsában (*has no lonely*), hiszen nincs is

X egyik csúcsában sem. Ilyen tulajdonságok mellett láthatjuk, hogy a tábla az *ab* szimbólumot kapja (*set symbol ab*).

Kezdjük el vizsgálni az **egyszerűsítés helyességét**. Ezt már csak a konzolban figyelhetjük meg. Először kiírja az összesített szimbólumot, majd hogy milyen szabályok szerint tudott egyszerűsíteni, végül az egyszerűsített alakot is kiírja. Például lássuk a legjobb kezdést, azaz amikor az első lépésünk valamely tábla közepére esik. Ekkor az összeszorzott kifejezésünk c^4. Ezt fogja a program redukálni.



3.3.2.3. ábra: Az egyszerűsítési folyamat a konzolba írva.

Ezt könnyen ellenőrizhetjük az egyszerűsítési szabályokkal akár kézzel is:

$$c^4 = ac^3 \qquad c^3 = ac^2$$

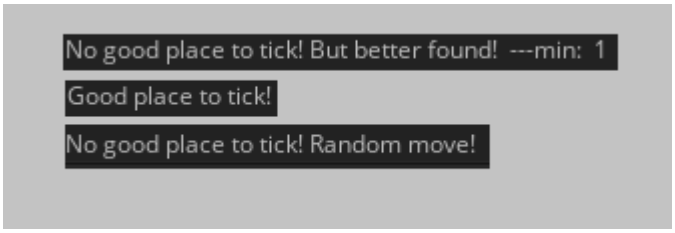
$$ac^3 = a^2c^2 \qquad c^3 = ac^2$$

$$a^2c^2 = c^2 \qquad a^2 = 1$$

A **gépi játékosok** működését is a konzolban követhetjük nyomon. A **könnyű** ellenfélnél csak egyfajta szöveget olvashatunk: „Nincs jó lépés! Véletlenszerűt lépek!” (*No good place to tick! Random move!*). Ez persze csak félig igaz, mivel lehet, hogy lenne ideális lépés, de mivel ugyanezt az eszközt használja a közepes gépi játékos is, ezért az ő esetében tényleg nincs jó lépés. Így ez a szöveg csak akkor jelenik meg, ha a könnyű ellen játszunk, vagy ha a közepesnek nem adunk optimális lépési lehetőséget.

Ha a **közepes vagy a nehéz** talál lehetőséget az előnyszerzésre, akkor a „Talált jó helyet!” (*Good place to tick!*) szöveget olvashatjuk a konzolban.

Nehéz játékosnál figyelhetjük meg az utolsó féle üzenetet a konzolban: „Nem talált jót, de azokból jobbat talált!”(*No good place to tick, but better found*). Ez akkor jelenik meg, ha a nehéz ellen játszunk és nem adunk neki előnyszerzési lehetőséget. Ekkor eggyel tovább gondolkodik és olyan helyre lép, amellyel utána nekünk a legkevesebb ideális lépéshelyünk marad. Azt is kiírja, hogy hány ilyen helyünk marad nekünk, miután lépett (*---min:x*).



3.3.2.4. ábra: A konzolban látható szövegek a gépi játékosok lépésének ellenőrzéséhez.

Persze e szövegek előtt láthatjuk a szimbólumokat, melyeket a közepes vagy nehéz játékos előre gondolkodása során kiszámolt. Ekkor ugyanolyan üzeneteket figyelhetünk meg, mint amilyeneket egyszerűsítéses részben láthattunk, csak többet egymás után. Ha a nehéz játékost kényszerítjük arra, hogy két lépéssel gondolkodjon előre, akkor még több ilyen szöveget olvashatunk a konzolban. Ezek segítségével le tudjuk ellenőrizni, hogy tényleg volt-e helyes lépés, vagy sem. Ha lett volna, de nem oda lépne a közepes vagy nehéz játékos, akkor kereshetnénk a hibát, de szerencsére ez a helyzet nem áll fönn.

3.3.3 DotsandBoxes mesterséges intelligencia tesztelése

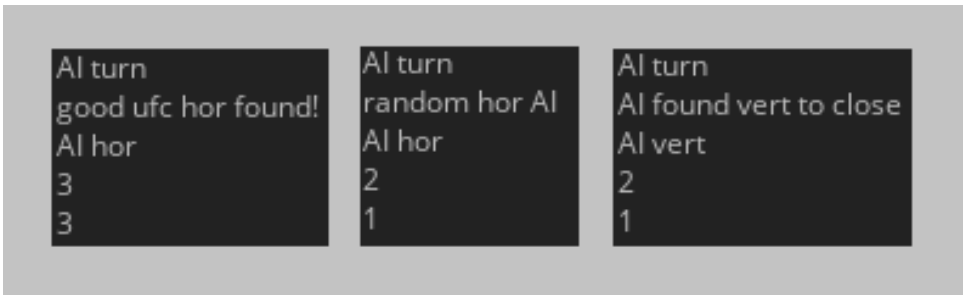
Tesztelési terv: A program állapotainak megfelelő változásait kell megfigyelnünk. Hasonlóan az előző játékhoz, ebben is a konzolra írt üzenetek alapján lehet felülvizsgálni a különböző funkciók működésének helyességét. Két fő területet kell ellenőriznünk:

1. **Beállítások:** a játék főmenüjében beállított paraméterek alapján töltődik be a játszma.
2. **Gépi játékosok:** mindegyik gépi ellenfél a hozzá tartozó kódblokkba lép be és állapítja meg a nehézségének megfelelő taktikával következő lépésének helyét.

A **beállítások** megfelelő működését egy játszma indításakor a konzolban megjelenő üzenetek alapján ellenőrizhetjük. Ezt a játszmaszoba irányítóobjektuma teszi, az ő létrehozási eseményébe van írva.

3.3.3.1. ábra: Példa a beállításokat ellenőrzését segítő üzenetekre

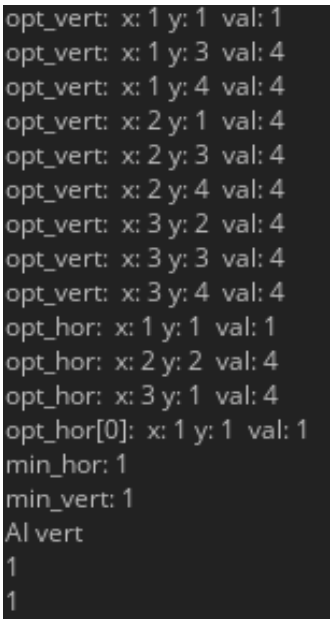
Minden körben, ami nem a játékosé, a „Gépi játékos köre” üzenetet láthatjuk a konzolban. A **könnyű** gépi játékos minden lépésekor véletlenszerűt lép. Ezért ha ellene játszunk, és nem tud pontot csinálni a körében, akkor a konzolban csak „Véletlen



3.3.3.2. ábra: A gépi játékosok lépésének ellenőrzésére szolgáló konzolba írt üzenetek

vízszintes/függőleges vonal” (*Random hor/vert link*) szöveget olvashatjuk. Ugyanezt láthatjuk, ha a **közepes** már nem képes olyan helyre rakni, hogy utána nekünk ne legyen pontunk. Ha **bármely** gépi ellenfél ellen játszunk, és képes pontot csinálni, akkor a „Talált befejésre vízszintes/függőleges vonalat” (*AI found hor/vert to close*) üzenet jelenik meg, attól függően, hogy milyen pozíciójú vonallal volt képes a pontszerzésre. Ha már a **közepes** vagy **nehéz** fokozatú ellen játszunk és képes még olyan helyre rakni, hogy mi utána ne tudjunk pontot szerezni, akkor a „Jó vízszintes/függőleges helyet talált” (*Good ufc hor/vert found*) szöveget olvashatjuk. Ha már nem tud ilyen helyet találni, akkor a **nehéz** játékos belekezd a szimulációjába.

Ezután kiírja a konzolba a fennmaradt helyek tulajdonságait, azaz koordinátájukat, illetve hogy hány pontot lennénk képesek akkor csinálni, ha azt behúzná. Ezek alá kiszámolja mind a függőleges húzások minimum értékét, mind a vízszintesekét. Majd ezen információkból ad egy optimális lépést. Mind ezek után feltünteti a gépi játékos lépésének koordinátáját.



3.3.3.3. ábra: A nehéz gépi játékos szimulációjának ellenőrzésére szolgáló konzoli szöveg