Beágyazott rendszerek szoftvertechnológiája

Házi feladat fejlesztői dokumentáció

*2022 Tavasz*

**KURVE FÍVÖR**

**Készítők: Hantos Márton, Harsányi Dániel, Fritsch Balázs**

**Konzulens: Naszály Gábor**

Tartalomjegyzék

[Hálózat és a menü 3](#_Toc105978351)

[Client 3](#_Toc105978352)

[Server 4](#_Toc105978353)

[Menu 6](#_Toc105978354)

[Játék logika 6](#_Toc105978355)

[Vector2D 7](#_Toc105978356)

[CurvePoint 7](#_Toc105978357)

[Curve 7](#_Toc105978358)

[Board 7](#_Toc105978359)

[Player 8](#_Toc105978360)

[ServerSidePlayer 8](#_Toc105978361)

[Game 8](#_Toc105978362)

[boolean[] detectCollisions() 9](#_Toc105978363)

[Grafikus interfész (GUI) 12](#_Toc105978364)

[ScreenManager 12](#_Toc105978365)

[GameScreen 14](#_Toc105978366)

[GamePanel 15](#_Toc105978367)

[InfoPanel 15](#_Toc105978368)

[MenuScreen 16](#_Toc105978369)

[EndGameScreen 17](#_Toc105978370)

# Hálózat és a menü

A menu úgy értendő, hogy a többi programrészt összefogó, és irányító kód, ami a GUI-tól kapott információk és események alapján kezeli az egész játékot. Erre szolgál a Menu osztály, aminek a run metódusa mag a fő függvény. Létezik egy Main osztály is, de az csupán meghívja a Menu run metódusát.

A játék kommunikációs szempontból egy kliens-szerver típusú, ahol a szerver egy speciális kliens. A játék tick alapú, azaz minden lépések szerint történik, ahol egy lépés egy fix időlépést reprezentál.

A Game osztály felelős a játéklogikáért (Curve-ok irányítása, ütközés detektálás, pontszámítás stb.). Ennek az osztálynak egyetlen egyede van, az pedig a szerverben. A Game-en belül van egy Board osztály. Ez tartalmazza a játék állását (Eddig lerajzolt Curve-ok, pontszámok), ez a fő játék tábla, ez ebben tárolt adatokon alapszik az új pozíciók kiszámolása, az ütközésdetektálás, és a pontszámítás. Amikor megtörtént egy lépés kiszámítása akkor (csak) az új pozíciókat elküldjük a klienseknek a pontokkal és a játék állapotával együtt. A kliensek ezekből a folyamatosan érkező pontokból a saját Board példányaikban felépítik a Curve-öket, aktualizálják a pontszámot stb.

## Client

Tárolja tagváltozóként a játékos nevét, egy Board példányt, a szerver IP-címét, és a socket-et és az adatküldésre használt object stream-eket. A kommunikáció 3 részre tagolódik. A játék elején inicializáció, játék közbeni kommunikáció, és a játék végi esetleges újrakezdés megvalósítása.

ReceiveFromServerInit()

Létrehozza a socket-et, és az ObjectInputStream-et valamint ennek az Output párját, majd blokkolva várakozik egy String-re a szerver felől. A szerver akkor fogja küldeni az első stringet, ha az összes kliens csatlakozott. Miután megkapta az első stringet, utána elküldi a szervernek a helyi játékos nevét, majd pedig vár egy InitPackageS2C objektum megérkezésére a szervertől. Ha ezt megkapta akkor az abban lévő adatok alapján létrehozza a kliens lokális Board-ját.

sendToServer()

Ez a függvény már a játék közbeni működéshez tartozik. Blokkolva várakozik, hogy a szerver küldjön egy kérelmet (újabb string), majd erre válaszolva elküldi a játékos irányítás input-ját, amit a gui-tól kap.

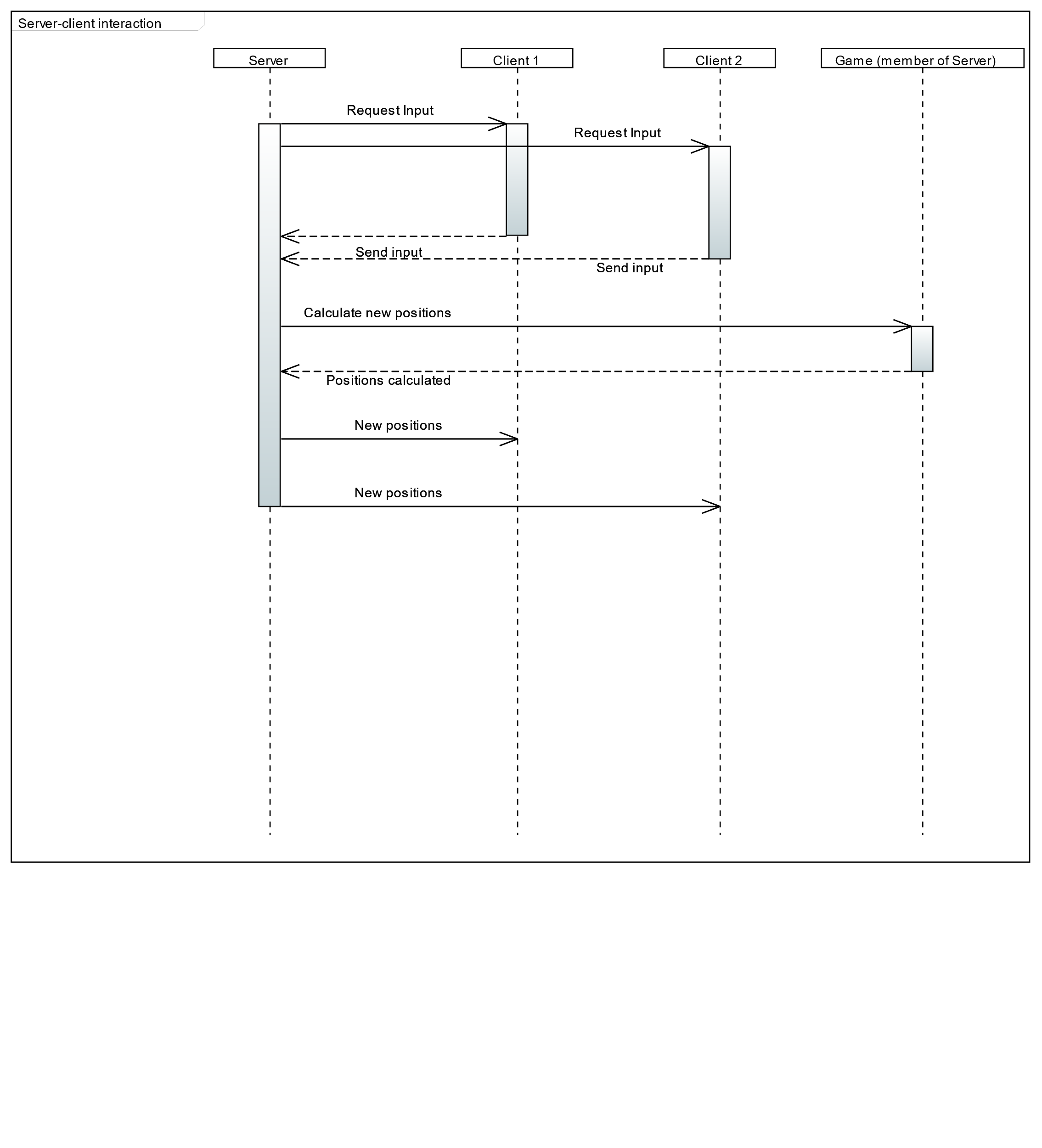
receiveFromServer()

Miután elküldte az aktuális ControlState-et az előző függvény, ez a függvény várakozik a szerver válaszára, amiben az újonnan kiszámított CurvePoint-ok, pontszámok, és játék állapot érkezik. Ezeket egy PackageS2C osztály szállítja.

sendReplayRequest() és waitForReplayMsg()

A játék végén az újrajátszás indikálásra szolgálna. Az első egyszerűen elküld egy stringet, a második pedig blokkolva várakozik, hogy a szervertől kapjon egy stringet, ami a sikeres újrakezdést jelzi. (Azaz, hogy minden játékos rányomott az újrakezdésre)

Példa a szerver kliens kommunikációra játék közben, 2 kliens esetén (Tehát összesen 3 játékossal), egyetlen tick (ciklus) ábrázolva:



## Server

Ő felelős a kommunikáció másik oldaláért, illetve ő kezeli a fő játék táblát. Kommunikációs szempontból jelentős különbség, hogy neki akár 3 klienssel is szükséges lehet egyszerre kommunikálni. A játék tick-et is ez az osztály kezeli, egy időzítő segítségével. Az időzítő megadott időközönként meghívja a runServer() függvényt.

Ahhoz, hogy több klienssel párhuzamosan lehessen kommunikálni, a konkrét adatküldést végző függvényekhez privát részosztályokat hoztam létre, amelyeket a runnable osztályból örököltettem, így thread-eket tudok majd belőlük készteni.

Főbb metódusok:

acceptConnections()

Blokkoló módon megvárja amíg az összes kliens csatlakozik. Az osztály konstruktorában meg kell adni a játékosok számát, aminél egyel kevesebb lesz a kliensek száma.

Setupgame()

Minden kliensnek elküldi a request-et, amire a kliensek a saját játékosuk nevét küldik vissza. Miután begyűjtötte az összes játékos nevét, létrehozza a Game-et és main Boardot a szerveren. Ezek létrehozásánál megtörténik a színek véletlenszerű kiosztása a játékosok között. Ha kiosztotta a színeket, megvannak a nevek akkor összeállítja az InitPackageS2C csomagot, és kiküldi minden kliensnek. Ha ez sikeresen megtörtént akkor elindítja a timer-t.

runServer()

Fő metódus. Ezt hívja meg a timer fix időközönként. Először elküldi a kérést az összes kliensnek, hogy küldjék el az aktuális inputot, ha megérkezett az összes akkor meghívja az új adatokkal a Game osztály runGame() metódusát, majd az újonnan kiszámolt CurvePointokat elküldi a klienseknek, a pontokkal együtt egy PackageS2C objektum segítségével.

GetControl és SendPoints alosztályok

Ez a két alosztály hivatott a tényleges kommunikáció lebonyolítására. A felépítésük egyszerű. A GetControl elküld egy konstans stringet egyetlen kliensnek, majd vár a válaszára blokkoló módon. A send points csak kiküldi a fő osztályban lévő és runServer által előkészített PackageS2C objektumot egy kliensnek. Mind a kettő a beépített Runnable osztályból származik. (implements)

requestInput()

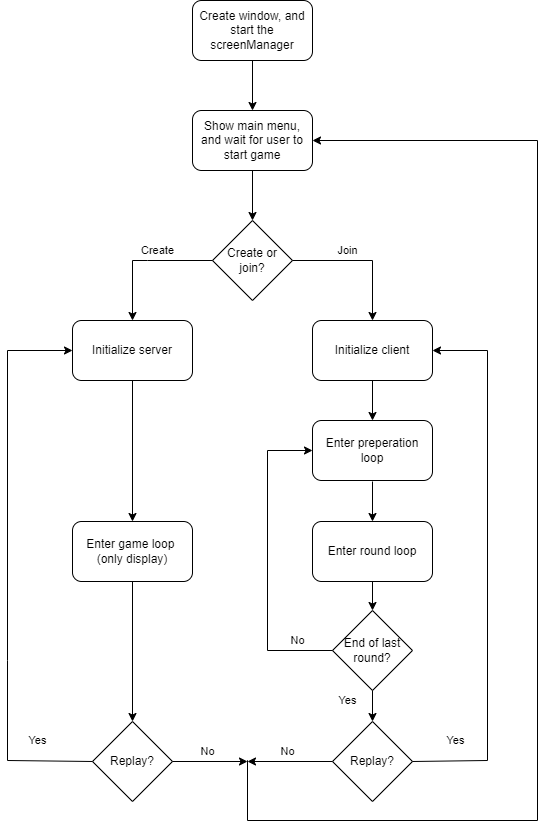
Létrehozz a GetControl futtatható osztályokból csinál thread-et és mindegyiket elindítja. Ezek után vár, hogy mindegyik szál befejezze a futást.

waitForReplayMsgs()

Az újra játszásnál van szerepe. Hasonlóan, mint a SendPoints-nál itt is egy külön runnable alosztályt hoztam létre. Ez a WaitForStringFromClientRunnable amiből ez a függvény hoz létre példányokat, majd azokból szálakat, és indítja el. Az adott szál blokkolva várakozik, hogy a hozzá tartozó kliens küldjön egy újra játszási kérelmet. Itt azért van szükség külön szálakra, mert nem tudjuk, hogy az egyes játékosok milyen sorrendben fognak rányomni az újrajátszás gombra, így nem lehet blokkolva várakozni. Ha a ez a függvény lefutott akkor tudjuk, hogy az összes játékos rányomott az újra játszásra.

## Menu

A Menu osztály felelős a program összefogásáért. A runMenu() metódus a fő függvény, ezt hívja meg a main függvény. Ezen kívül csak adat másoló függvények vannak az osztályban, amik a GUI felé továbbítják a megjelenítendő dolgokat. A program fő logikáját szemléleti az alábbi ábra:



A szerver oldalon azért szerepel az only display a game loopnál, mert maga a számítás, mint már írtam a Server osztályban történik. A szerver csak egy flag-gel jelzi a menü osztálynak, ha elkészült egy adott tick számításával, ekkor a Menu elindítja a kijelzés frissítését, majd várakozik a következő tick-re.

# Játék logika

A játék logikát megvalósító osztályok közül a Game osztály áll a központi szerepben. A Game osztályban példányosodnak a játék logika elemei, tehát a játékosok, ők pedig a játékban hozzájuk tartozó elemeket tartalmazzák (vonalak, pontszám), illetve a játékteret megvalósító Board osztályt is tartalmazza. A Game osztály az, amelyiken keresztül módosítani lehet a játékteret és a játékosok tagváltozóit, illetve ez az osztály tartalmazza a számolások metódusait is.

A játéklogikához tartozó osztályokat és metódusokat bottom-up stílusban mutatjuk be.

## Vector2D

A Vector2D osztály segítségével írjuk le a pályát, az euklidészi térben ez az osztály hordozza törtszámként a pontok koordinátáit. Az osztálynak csak getterei és setterei vannak.

## CurvePoint

A CurvePoint osztály a Vector2D osztályból leszármazó osztály, az isColored tagváltozója hordozza azt az információt, hogy az adott pontot meg kell-e jeleníteni a játéktéren. Erre később a vonalakban keletkezett lyukak miatt van szükség, a lyukak a játékban olyan pontjai a játékosok által húzott vonalaknak, amiknek ez a tagváltozójának az értéke false.

## Curve

A játékban egy játékoshoz tartozó vonalat a Curve osztály valósítja meg. A Curve osztály CurvePointok összessége. Mivel a vonalak mérete folyamatosan nő, ezért a Java beépített [ArrayList](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html) osztályát használtuk a pontok tárolására. A játékban egyenként a CurvePointok színe nem fontos a logika szempontjából, ezért a teljes Curve hordozza annak a pontjainak a színét. A Curve alap metódusain (getterek, setterek) felül a játék logikát megkönnyítvén implementáltuk az alábbi metódusokat:

* getCurveSize() – nem kap bemeneti paramétert, visszaadja a Curve méretét
* getLastPoint() – visszaadja a Curve legutóbb hozzáadott pontját
* deleteLastPoint() – kitörli a legutóbb hozzáadott pontot

## Board

A Board osztály tárolja a teljes játékteret, illetve a megjelenítéshez szükséges adatokat. Ez az az osztály, amiben megjelenik mind a három nagyobb része a programnak (megjelenítés, játék logika és hálózati kommunikáció). A Board tárolja a játékosok neveit, Curve-jeit, pontszámait. Logikailag inkább a Game osztályhoz tartozna pár tagváltozója, ezek a roundNum, a currentRound, és az előző állapotot tartalmazó prevGameState, de a megjelenítés során a kliens oldalon is ilyen Boardok alapján történik a pálya kirajzolása. A Board metódusai:

* getLastCurvePoints() – visszaadja az összes Curve legutolsó pontját, a hálózati kommunikációhoz szükséges
* addCurvePoints(CurvePoint[] newPositions, boolean[] playersAlive) – hozzáadja az adott játékoshoz tartozó pontot a játékos vonalához, ha az életben van, ezt a második paraméterként kapott boolean tömb határozza meg
* deleteLastCurvePoints() – letörli a legutolsó pontokat a vonalról
* clearBoard() – letisztítja a Boardról a vonalakat

## Player

Ez az osztály tartalmazza egy játékos nevét, illetve az irányításhoz használt ControlState típusú irányítási állapotot, illetve az id-t, ami alapján azonosítva van a szerver oldalon.

## ServerSidePlayer

A ServerSidePlayer osztály a Player osztályból származik le, kibővíti a játék logika szempontjából használt, de a kliens oldali megjelenítés szempontjából nem használt tagváltozókkal. Ez az osztály tárolja egy játékos pontszámát, pozícióját, sebességét és színét, illetve azt, hogy életben van-e. Az alap metódusain felül az alábbi metódusokat valósítottuk meg az osztályon belül:

* rotateSpeed(double angle) – az aktuális sebességvektort elforgatja a paraméterként adott fokkal
* move() – az aktuális irányítási állapot alapján beállítja (elforgatja) a sebességvektort és a pozíciót eltolja a sebességvektorral
* updateScore(double scoreToAdd) – a paraméterként kapott számot hozzáadja az aktuális pontszámhoz

## Game

A Game osztály valósítja meg a játék logikát és tartalmazza az ahhoz kapcsolódó metódusokat. A Game osztály tartalmazza a játékhoz köthető eltárolandó adatokat, mint a körök száma, az aktuális kör, a játékosok (ServerSidePlayer típus), a Board, illetve a játék állapotát. A játék menetét 3 metódus vezérli, ezek a metódusok ciklikusan hívódnak a szerver oldalon.

boolean runGame(ControlState[] Controls, int currentCycle)

A runGame() metódus végzi el a minden játékciklusban meghívandó számoló metódusok hívását, a játék állapot kezelését. A runGame() visszatérési értékként egy boolean-t ad a szervernek, ez az érték a kör legvégén true, egyébként false. Bemeneti paraméterként a szerver oldalról megkapja a játékosokhoz tartozó irányítási állapotokat, illetve egy számláló értéket, amiből kikövetkeztethető, hogy éppen hol tart a játék. A metódus a játék belső állapotától függően működik, a játék állapotgépét a szerver kívülről vezérli.

Prep állapotban az előkészületeknek megfelelően a bemeneti paraméterként kapott irányítási állapotoknak megfelelően változtatja a játékosok sebességét, de nem mozgatja azokat. Ilyenkor a sebesség megjelenítéséhez még egy pont belekerül a játékosokhoz tartozó Curve-ökbe, ez a sebességvektor sokszorosa hozzáadva a pozícióhoz. A Prep állapot végén szerver oldalon törölni kell ezt a pontot. A metódus visszatérési értéke false.

Playing állapotban a játék számoló metódusai hívódnak, amiket az evaluateStep() függvény valósít meg. Ha kör vége van a visszatérési érték true, egyébként false.

Menu állapotban a Game osztály alvó állapotban van, visszatérési értéke false, más nem történik.

boolean evaluateStep(ControlState[] Controls, int currentCycle)

Az evaluateStep() függvény minden ütemben az élő játékosok pontszámát növeli és frissíti a Boardon, meghívja a pozíciók változtatását kiértékelő updatePositions() metódust, illetve a kör végét vizsgálja a még életben maradt játékosok függvényében. A bemeneti paraméterként kapott Controls és currentCyccle változókat az updatePositions() metódus veszi át. Visszatérési értéke true, ha csak egy játékos marad életben, egyébként false.

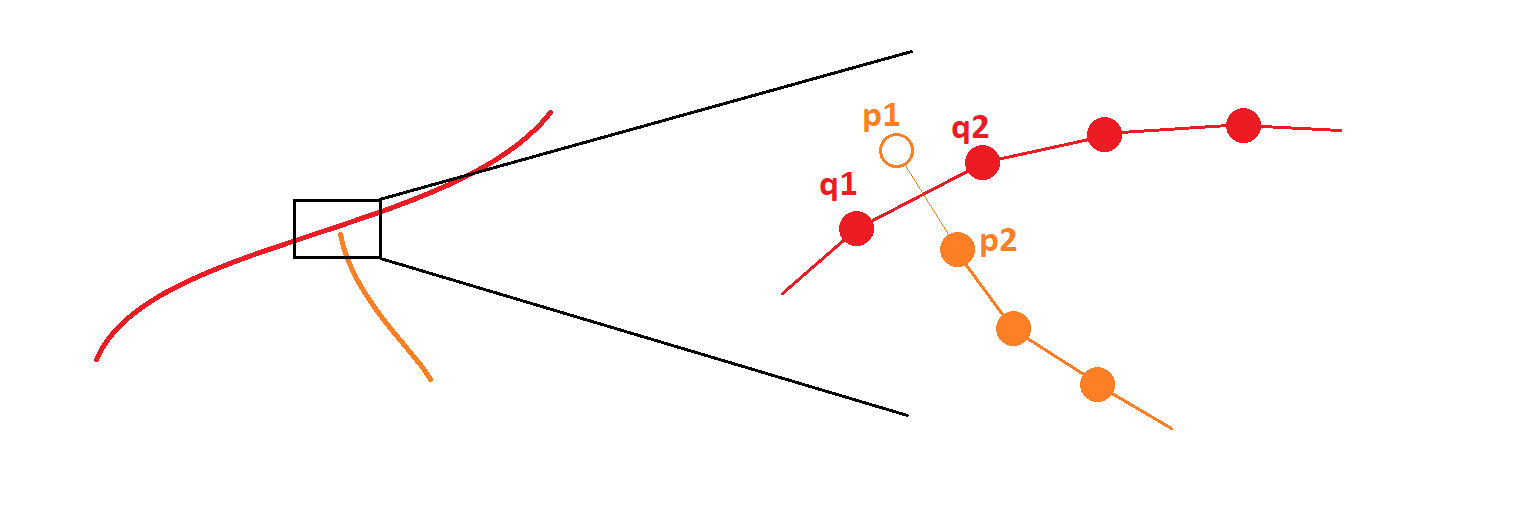
void updatePositions(ControlState[] Controls, int currentCycle)

Az updatePositions metódus végzi a játék során minden ütemben a játékosok mozgatását, ütközések kiértékelését, illetve az ütközéseknek megfelelően a játékosok és a Board állapotait beállítja. Ez a metódus hívja a bemeneti paraméterként kapott ciklusszámláló alapján a random lyukakat generáló függvényt is, így a Curve-ök bővítésénél a CurvePoint láthatóságát már beállítva adhatók hozzá a játékosok Curve-jeihez az új pontok.

### boolean[] detectCollisions()

A játéklogika legkomplikáltabb eleme a játéktéren az ütközések vizsgálata volt. Az ütközések vizsgálata során a játékosok Curve-jeit kell vizsgálni az adott játékoshoz tartozó vonal legelejével (hiszen csak a „kukac feje” ütközhet neki a vonalaknak). A metódus visszatérési értékeit tároló ideiglenes boolean tömböt feltölti a legelején false értékekkel, és a későbbiekben az adott játékosnak megfelelő indexű elemét true-ba állítja ütközés esetén.

A metódus végig iterál minden élő játékosnál a pályán található összes Curve-ön (a sajátját is beleértve), és ezeknek a Curve-öknek a pontpárjainak a segítségével meghatározza, hogy keresztezi-e egymást a Curve feje és bármelyik pontpár a Curve-ökön belül. Ezt a keresztezés detektálást a [geeksforgeeks implementációját](https://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/) felhasználva végzi el a metódus a szakaszpárokra. A metódus működéséhez készítettünk egy kis illusztrációt:



boolean[] setHoles(int currentCycle)

A játékmechanikához hozzátartozik az is az izgalom fokozása érdekében, hogy a játékosok által húzott Curve-ökbe viszonylag random időzítésekkel lyukakat helyezzen a logika. Ehhez a játék ciklusszámlálóját használtuk fel, minden játékosnak van egy olyan tagváltozója, amit állítva a ciklusszámláló alapján generálhatóak a random lyukak.

A lyukak fix hosszúak, a lyukakat követően pedig egy fix ideig biztosan vonalat fognak húzni a játékosok. A fix idő elteltével egy rövid, de random időszakasz határozza, meg, hogy pontosan mikor keletkezik lyuk a játékos vonalában. A metódus visszatérési értéke egy boolean tömb, azt adja vissza, hogy az adott játék ütemben az egyes játékosok új pontjai láthatóak, vagy nem (tehát az új pontok isColored tagváltozója ez alapján kerül beállításra).

void updatePlayerScores()

A kör végi pontszámításokat ez a metódus végzi el. A játék futása során egy kör szempontjából ideiglenes tömbökbe (Game tagváltozói) tárolódnak el a halálozások sorrendje, illetve a körben szerzett pontszámok. A specifikációban már ismertetett módon a metódus a körök végén egyszer lefut és elvégzi a szükséges pontszámításokat a körben szerzett pontszámok, illetve a halálozások sorrendje alapján.

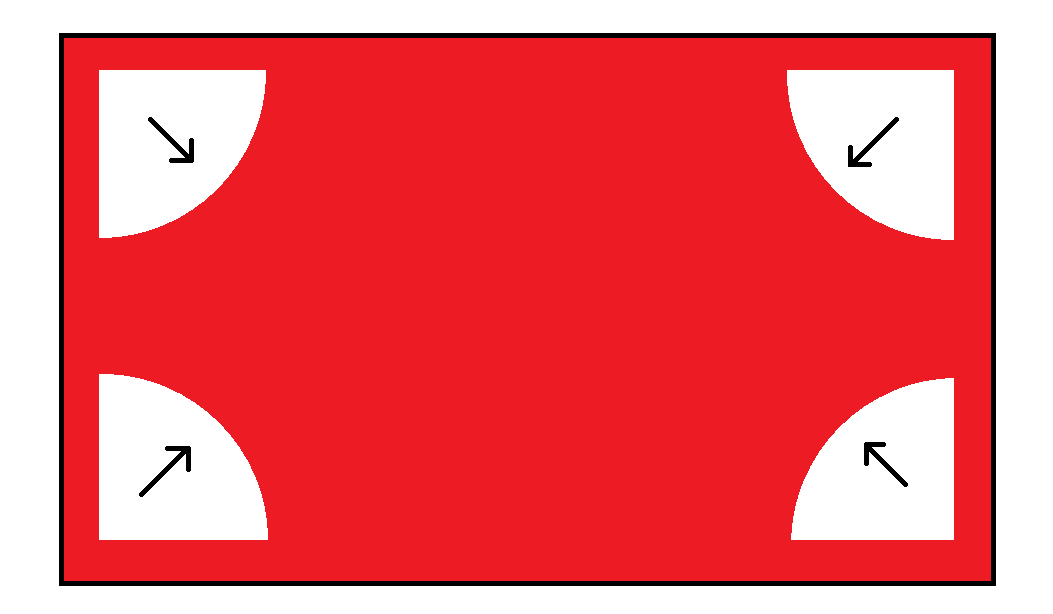
void initGame()

A játéklogika feladata az inicializáció, tehát a játékosok számának a függvényében azoknak egy véletlenszerű szín, pozíció generálása, illetve a pozíciónak megfelelően egy fix nagyságú kezdősebesség megadása. Az initGame() metódus foglalja össze ezeket az inicializációs metódusokat.

void initPositions()

A metódus inicializálja a játékosok pozícióját és sebességét. A játékosok négy pozícióban kezdhetnek, ez a négy pozíció a játéktér négy sarkában található, a pozíciónak megfelelően a játéktér közepe felé irányul a kezdősebesség. A pozíciók generálása egy negyedkörön belül random, de túl közel nem lehet a játéktér széléhez.

Ezt az elrendezést a kezdősebességekkel az alábbi ábrával szemléltettük (a piros területre nem kerülhet játékos):



A generált pozíciók a játékosok számától függ, hogy hova kerülhetnek, ezeket a random pozíció-kezdősebesség párosokat a metódusban összekeveri a játékosok között. Ahhoz, hogy a keverésnél a kezdőpozíciók és a sebességek konzisztensek maradjanak, egy segéd osztályt használtunk, aminek a két Vector2D típusú tagváltozója pozíció és a sebesség értékeket hordozza, így egy ArrayList és annak a [shuffle](https://mkyong.com/java/java-how-to-shuffle-an-arraylist/) metódusának a segítségével kevertük össze a játékosok között a kezdőpozíció-kezdősebesség párosokat.

Vector2D generateRandomPosition(playerPositions playerPosition)

Az egyes játékosok random pozícióját ez a metódus generálja a fentebbi ábrán látható fehér területre. A metódus bemeneti paramétere egy enum, aminek értékei lehetnek: TOP\_LEFT, TOP\_RIGHT, BOTTOM\_LEFT és BOTTOM\_RIGHT, azt határozza meg, hogy melyik sarok közelébe helyezze el a véletlenszerű pozíciót a metódus. A visszatérési értéke a metódusnak a generált pozíció.

void initColors()

A játékosokhoz tartozó színeket is előre meghatározó színekből választja ki a metódus és rendeli a játékosokhoz. Ezt a jó láthatóság és elkülönítés jegyében valósítottuk meg így. A már előző metódusnál ismertetett Arraylist shuffle metódusát használtuk erre a feladatra is, az előre meghatározott tömb elemét összekeveri metódus, és az első X számú értéket a neki megfelelő indexű játékosnak osztja ki.

void initBoard()

Végül pedig a játéktér inicializálását végző metódus a már beállított játékos paraméterek alapján elhelyezi őket a Board-on és beállítja a hozzájuk tartozó isAlive tagváltozók értékét true-ra.

# Grafikus interfész (GUI)

A megjelenítési funkciókat igyekeztünk teljes mértékben elkülöníteni a többitől, így külön csomagba került (gui). A csomag feladata a játék megjelenítésén kívül a felhasználók perifériákkal való interakcióinak detektálása és kezelése. Ezekre a feladatokra jellemzően a [swing](https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html) és [awt](https://docs.oracle.com/javase/7/docs/api/java/awt/package-summary.html) könyvtárakat használtuk. Először igyekszünk bemutatni a működés architektúráját, majd egyesével a megvalósító osztályokat.

A gui a swing könyvtár beépített grafikus komponenseit használja, gyakorlatilag az összes főbb építőelem a [JPanel](https://docs.oracle.com/javase/7/docs/api/javax/swing/JPanel.html) osztályból van leszármaztatva. Ezeket a főbb alkotóelemeket Screen-eknek nevezzük és ezeken a paneleken történik a megjelenítés, illetve a felhasználói interakciók kezelése. A program főbb állapotaihoz tartozik egy Screen (JPanelből leszármaztatott osztály):

* MenuScreen
* GameScreen
* EndGameScreen

Ezeket egy ScreenManager nevű, szintén JPanel-leszármazott osztály fogja össze és a játék állapotától függően jeleníti meg őket.

## ScreenManager

Mint az említésre került, ez az osztály felel a különböző képernyők összefogásáért. Ez bővebben azt jelenti, hogy közvetíti számukra a megjelenítéshez szükséges információt, illetve egy állapotgép alapján mindig a megfelelő Screen-t jeleníti meg. Az állapotgép a gui package-en kívüli programState enum-ot használja. Ezen tagváltozót használja fel a programvezérlő logika is: Ha a logika megkívánja, akkor egy setter-rel állítható, az osztályon belül pedig a különböző felhasználói interakciók következtében lehet szükséges változtatása. Ezen kívül fontos tagváltozók még az említett képernyő osztályok és azok a változók, melyeket a felhasználó akciói által (pl.: gombnyomás) kerülnek beállításra.

Mint az említésre került, ez az osztály felel a különböző képernyők összefogásáért. Ez bővebben azt jelenti, hogy közvetíti számukra a megjelenítéshez szükséges információt, illetve egy állapotgép alapján mindig a megfelelő Screen-t jeleníti meg. Az állapotgép a gui package-en kívüli programState enum-ot használja. Ezen tagváltozót használja fel a programvezérlő logika is: ha a logika megkívánja, akkor egy setter-rel állítható, az osztályon belül pedig a különböző felhasználói interakciók következtében lehet szükséges változtatása. Ezen kívül fontos tagváltozók még az említett képernyő osztályok és azok a változók, melyeket a felhasználó akciói által (pl.: gombnyomás) kerülnek beállításra. Ezen akciók észlelésének céljából az osztály implementálja az [ActionListener](https://docs.oracle.com/javase/7/docs/api/java/awt/event/ActionListener.html) és [KeyListener](https://docs.oracle.com/javase/7/docs/api/java/awt/event/KeyListener.html) interfészeket.

Előbbi segítségével lehet érzékelni, ha a felhasználó valamilyen swing objektumra (pl.: gombra) kattint. Ez az actionPerformed(ActionEvent e) függvény override-olásával történik meg. Ha valami akció történik egy olyan objektumon, amihez az adott ActionListener hozzá lett adva, akkor ez a függvény kerül meghívásra. Itt egy egyszerű if szerkezet és az ActionEvent.getSource() metódus segítségével megvizsgálásra kerül, hogy melyik képernyő mely objektuma volt a forrás. Ez után kinyerésre kerülnek a megfelelő információk és eltárolásra kerülnek a tagváltozókban, hogy a package-en kívüli osztályok hozzáférjenek. Ezen kívül az állapotgép is frissül, amennyiben ez szükséges. Például, ha a felhasználó a Create Game gombra kattint a MenuScreen-en, akkor a kattintás pillanatában aktuális beállítások (játékos név, játékosok száma, körök száma) kerülnek elmentésre, hogy a vezérlő logika tudja, hogy milyen paraméterekkel kell új játékot létrehozni. Továbbá egy „Waiting for other players” felirat is megjelenítésre kerül a menü képernyőjén. Természetesen ebben a példában a programState állapot MENU-ről IN\_GAME-re vált.

A KeyListener interfész az ActionListener-hez hasonlóan működik, itt a keyPressed(KeyEvent e) és a keyReleased(KeyEvent e) függvények vannak override-olva. A függvények neve kifejező: akkor hívódnak meg, ha a felhasználó lenyomott vagy felengedett egy billentyűt. A Keyevent.getKeyCode() metódussal megkapható a billentyű kódja. Egy egyszerű logika beállít pár segédváltozót, melyeket az evaluateInput() tagfüggvény fog felhasználni a felhasználói bemenetek feldolgozásához.

void evaluateInput()

Ez a függvény a KeyListener interfész függvényei által beállított változók segítségével beállítja a ControlState tagváltozót. Ez gyakorlatilag egy enum, és azt kódolja, hogy a játékos előre megy, jobbra kanyarodik, vagy balra kanyarodik.

update(boolean firstCall)

Gyakorlatilag ez a gui legfontosabb függénye, ezt hívja minden ciklusban a játéklogika, miután frissítette modellt. Egy switch-el van vizsgálva, hogy éppen mi a programState és ennek megfelelően kerül megjelenítésre a megfelelő Screen. Ehhez először meg kell hívni az adott képernyő render függvényét (ezek később kerülnek részletezésre), majd meg kell jeleníteni. Ezt a funkciót a [CardLayout](https://docs.oracle.com/javase/7/docs/api/java/awt/CardLayout.html) osztály teszi lehetővé, ez a layout van beállítva a (JPanelből leszármazó) ScreenManager-nek. A CardLayout úgy működik, hogy egy string-el kódolva container-eket (jelen esetben JPanel-eket, mint Screen-eket) lehet hozzáadni, majd a show method-dal lehet ezeket megjeleníteni, egyszerre egyet. A show method meghívása csak akkor szükséges, ha állapotváltás következett be, egyébként elég csak render-elni. A firstcall paraméter átadásra kerül a meghívott render függvényeknek, innen fogják tudni a Screen osztályok, hogy normál, vagy inicializáló ciklus szerint kell eljárniuk. A függvény végén meghívásra kerül az evaluateInput() is, hogy a következő ciklusban a megfelelő bemenetekkel dolgozhasson a játék logika.

## GameScreen

Ez a képernyő felel a játék állapot megjelenítéséért és szinte minden, a többi Screen-nél is alkalmazott technika megtalálható itt, először ezek kerülnek általánosságban ismertetésre. Alapvetően minden grafikus swing objektumhoz, így a JPanel-hez is tartozik egy paintComponent(Graphics g) metódus, mely meghívásra kerül amikor ki kell rajzolni. Ezek a kirajzolások külön szálon futnak automatikusan, például, amikor átméretezésre kerül az ablak, de manuálisan is ütemezhetők a repaint() metódussal. A paintComponent override-olásával egyedi grafikák is megvalósíthatók, ez több esetben is kihasználásra került a projektben. A container-ek tartalmazhatnak más grafikus objektumokat (JPanel, JTextField, JButton stb.), ezek layout-ok segítségével helyezhetők el dinamikusan és esztétikusan. Legtöbb esetben a [GridBagLayout](https://docs.oracle.com/javase/7/docs/api/java/awt/GridBagLayout.html) és a [BorderLayout](https://docs.oracle.com/javase/7/docs/api/java/awt/BorderLayout.html) volt alkalmazva ezen célokra. Fontos arról is szót ejteni, hogy az egyes Screen-ek hogyan jutnak hozzá a kirajzoláshoz szükséges információhoz. Ezek a tagváltozók a játékot vezérlő főciklusból kerülnek beállításra setter-ek segítségével. A kirajzoláshoz minden információ megtalálható a Board osztályban, innen kerülnek átmásolásra az adatok. Itt figyelni kell arra, hogy vannak olyan adatok (pl.: játékosok nevei), melyeket csak egyszer kell átadni, az inicializáló render előtt, de olyanok is, melyeket minden ciklusban (pl.: játékosok aktuális pontszáma).

A GameScreen osztály további két JPanel-ből leszármaztatott osztályt tartalmaz tagváltozóként, ezek a GamePanel és az InfoPanel. Előbbi a játéktér és a Curve-ök kirajzolására van használva és a GameScreen közepére van pozicionálva (BorderLayout.CENTER), utóbbi pedig a képernyő jobb szélén (BorderLayout.EAST) található és a játékosok pontszámait mutatja élőben, sorrendben.

render(boolean firstCall)

A konstruktoron és a getter/setter függvényeken kívül ez az egyetlen tagfüggvény, aminek az a feladata, hogy a ScreenManager-ből meghívva a firstcall paraméternek megfelelően inicializálja vagy frissítse a tartalmazott GamePanel és InfoPanel osztályokat azokkal az információkkal, melyeket a Board osztálytól kapott meg. A függvény végén meghívásra kerül a reprint(), ami ütemezi az objektum újra rajzolását. Ez azt jelenti, hogy önmagának és a tartalmazott grafikus objektumoknak is meghívódik a PaintComponent függvénye.

### GamePanel

Amikor frissítő módban hívódik meg a GameScreen render függvénye, akkor egy, a játékosok számának megfelelő nagyságú, CurvePoint-okból álló tömb kerül átadásra a GamePanel egy tagváltozójának, ami tartalmazza a legutóbb kiszámolt pontot minden játékoshoz. Ezeket kell az osztálynak megjeleníteni úgy, hogy hozzárajzolja az eddigi pontokhoz. Azonban az override-olással egyedi grafikák készítésére alkalmas PaintComponent úgy működik, hogy újra rajzolja az egész objektumot. Ezért először játékosszámnak megfelelő számú pont helyett a teljes Curve-ök átadásával és minden iterációban való újra rajzolásával próbálkoztunk, de ez a körök vége felé nagy késleltetéseket okozott. Így ehelyett egy [BufferedImage](https://docs.oracle.com/javase/7/docs/api/java/awt/image/BufferedImage.html) tagváltozóra rajzoltunk és a PaintComponent-ben ezt a képet rajzoltattuk ki minden iterációban.

void updateBoardImage()

Ez a függvény szolgál a kirajzolandó kép új pontok által történő frissítésére. A BufferedImage.createGraphics() függvény segítségével hozzáadásra kerül egy [Graphics2D](https://docs.oracle.com/javase/7/docs/api/java/awt/Graphics2D.html) objektum a képhez, melynek segítségével rajzolni lehet rá. A képi görbék úgy vannak frissítve, hogy egy for ciklusban az új pontok és az előző iteráció pontjai közt megfelelő színnel egy egyenes kerül kirajzolásra a Graphics2D.draw(new Line2D.Double(params)) segítségével, amennyiben az adott új CurvePoint isColored attribútuma igaz. Az egyenes rajzoló függvény paraméterei gyakorlatilag az egyenest meghatározó négy koordináta: ebből kettő az új pont koordinátái, kettő pedig az előzőé. Az előző pontok egy shift regiszter segítségével vannak eltárolva.

void resetBufferedImage()

Ez a függvény arra való, hogy új játék kezdetekor az eddigi rajzot letörölje a BufferedImage-ről, ezt egy háttér színű, képpel megegyező méretű téglalap rajzolásával teszi meg.

void paintComponent(Graphics g)

Ez az override-olt függvény felel az objektum megfelelő újra rajzolásáért. Itt meghívásra kerül az előbb bemutatott updateBoardImage, majd a frissített kép kirajzolásra is kerül a JPanel közepére a Graphics.drawImage segítségével.

### InfoPanel

E panel feladata játék közben színhelyesen és sorrendben kiírni a játékosok nevét, pontszámát, illetve az aktuális körnek a számát. A GameScreen render függvényének frissítő módban való hívásakor átadásra kerülnek az aktuális játékos pontszámok és az aktuális kör száma. Inicializációkor ezek mellett átküldődnek a játékos nevek és címek, a játékosok száma és az adott játék beállított körszáma is.

void sortScores()

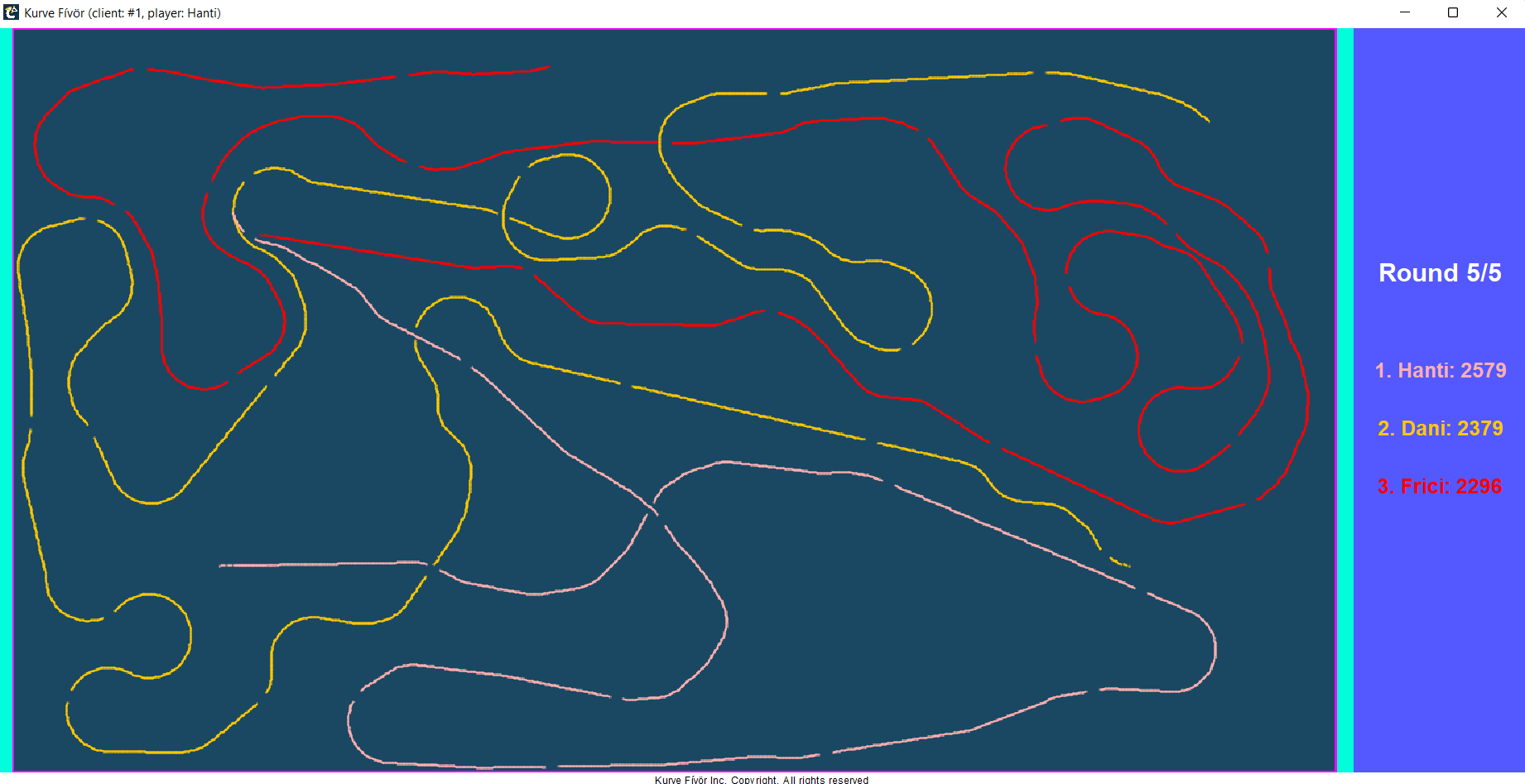
Mivel a kirajzoláshoz szükséges jellemzők fix sorrendben kerülnek átadásra, ezeket külön belső változókban sorrendbe kell rakni, ezt teszi meg ez a függvény. A játékosok pontszámát és ez alapján neveit, valamint színeit teszi sorrendbe egy cserélésre alapuló algoritmussal.

void drawScore()

Ez a függvény felel az információk helyes kiírásáért. Az osztály konstruktora előre meghatározott pozíciókban elhelyezett [JLabel](https://docs.oracle.com/javase/7/docs/api/javax/swing/JLabel.html)-eket, melyekhez szöveget lehet rendelni. A függvény ezeket a szövegeket frissíti, miután meghívta a sortScores-t.

void paintComponent(Graphics g)

Az override-olt függvényben a panel hátterének beállítása után csupán az ismertetett drawScore kerül meghívásra.



. ábra: GameScreen

## MenuScreen

A GameScreen-hez hasonló BorderLayout elrendezésben van egy JPanel a képernyő közepén, ahol a CreateGame, JoinGame és Exit gombok talalálhatók. Emellett jobb szélen elhelyezésre került egy beállításokat tartalmazó panel is. Mivel ezek a panelek nem tartalmaznak egyedi grafikát ezért nem kellett őket leszármaztatni, csupán tagváltozóként felvenni. A MenuScreen-en sok swing grafikus objektum található, ezek közül azokat kellett tagváltozóvá tenni, melyekhez szükséges hozzáférni a ScreenManager-ből (ActionListener hozzáadása). Ezek közül a fontosabb típusok:

* [JButton](https://docs.oracle.com/javase/7/docs/api/javax/swing/JButton.html): A gombokat megvalósító osztály.
* [JTextField](https://docs.oracle.com/javase/7/docs/api/javax/swing/JTextField.html): A billentyűzettel bevihető adatokat (IP-cím, játékos név, körök száma) ilyen mezőkbe írhatja be a felhasználó.
* [JRadioButton](https://docs.oracle.com/javase/7/docs/api/javax/swing/JRadioButton.html): Az irányítás típusa (A/D vagy jobb nyíl/bal nyíl) választható segítségével.
* [JComboBox](https://docs.oracle.com/javase/7/docs/api/javax/swing/JComboBox.html): Egy legördülő lista valósítható meg vele, amivel a játékosok száma állítható.

Szövegek elhelyezésére JLabel-ek voltak használva, illetve a szebb pozicionálás céljából voltak még alkalmazva további JPanel-ek is, de ezeket nem kellett tagváltozóvá tenni. Mivel a MenuScreen-en nem kellett megvalósítani egyedi grafikát és futás közben sem kell frissíteni tartalmát, ezért a konstruktoron kívül nincsen tagfüggvénye.



. ábra: MenuScreen

## EndGameScreen

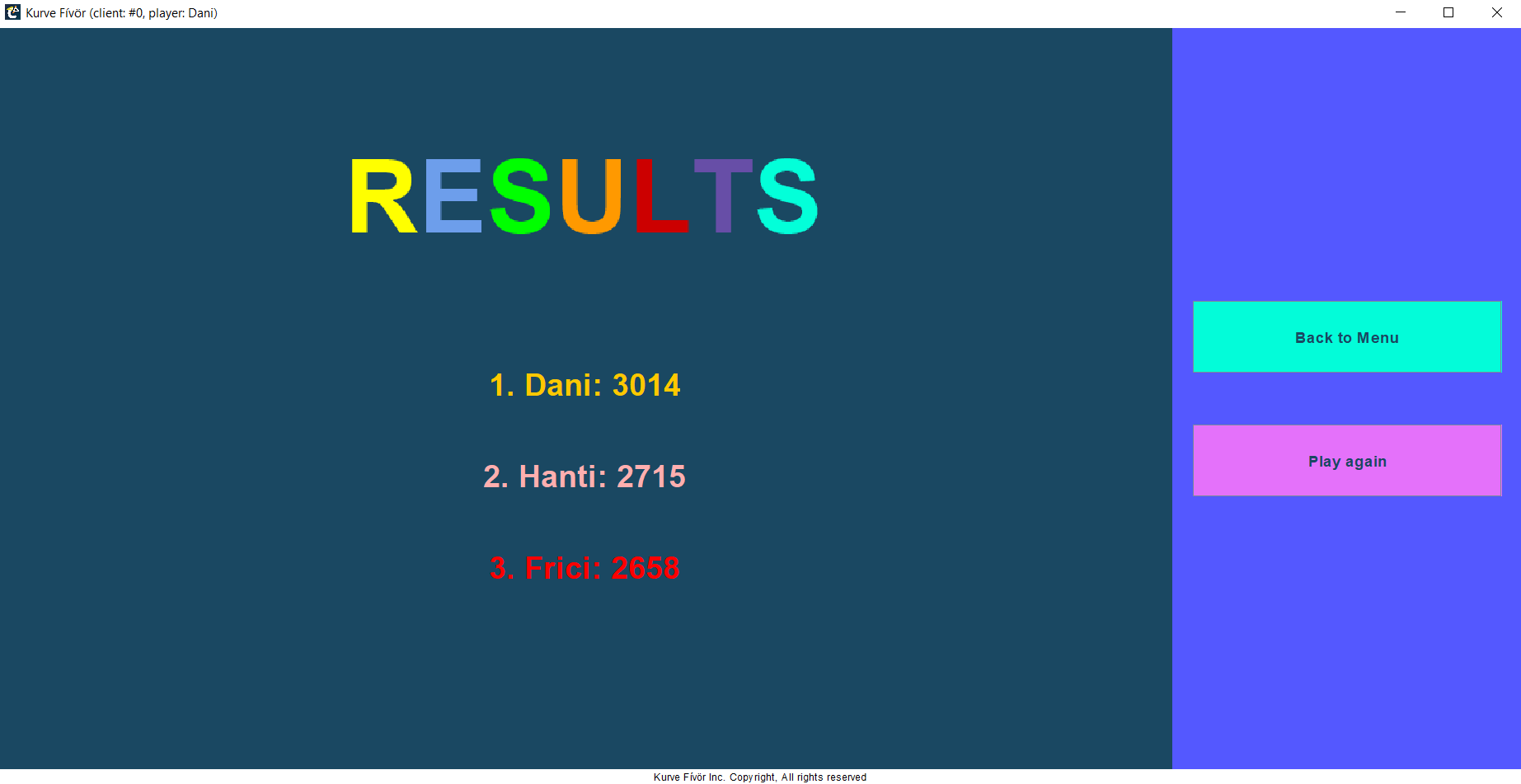
Ez egy nagyon egyszerű képernyő, gyakorlatilag két JPanelt tartalmaz BorderLayout elosztásban, az egyiken a játék végeredménye kerül kiírásra, a másikon pedig két gomb található, az egyik a menübe való visszatérésre szolgál, a másik pedig új játék indítására. A megjelenítéshez szükséges adatokat a többi Screen-nél már ismertetett módon kapja meg.

void sortScores()

A GameScreen InfoPanel-jéhez hasonló működésű sorrendező függvény, ami ahhoz szükséges, hogy a játékosok nevei és pontjai sorrendben és megfelelő színnel kerüljenek kiírásra.

void render()

Az InfoPanel-hez hasonlóan a konstruktorban fix pozícióban felvett JLabel-ek szövegeit változtatja meg, miután lefutott a sortScores függvény.



. ábra: EndGameScreen