



**Eötvös Loránd Tudományegyetem**  
**Informatikai Kar**  
**Információs Rendszerek Tanszék**

## **Portal 2D**

**Témavezető:**

Dr. Gombos Gergő  
adjunktus, Ph.D.

**Szerző:**

Kiss János  
Programtervező informatikus BSc.

Budapest, 2020

## Tartalomjegyzék

1.	Bevezetés .....	4
1.1	Rövid ismertető.....	4
1.2	Motiváció.....	4
2.	Felhasználói dokumentáció .....	5
2.1	A játék telepítése és telepítési előfeltételei.....	5
2.2	Szerver.....	5
2.2.1	Konfigurálása.....	5
2.2.2	Elindítása.....	6
2.3	Kliens.....	7
2.3.1	Elindítása.....	7
2.3.2	Használat.....	7
3.	Fejlesztői dokumentáció .....	12
3.1	Szerver.....	12
3.1.1	Fizika.....	13
3.1.1.1	Entity .....	13
3.1.1.2	Ball .....	14
3.1.1.3	Wall .....	16
3.1.1.4	RoundWall .....	16
3.1.1.5	Portal .....	17
3.1.1.6	Fizika motor .....	19
3.1.2	Játéktér.....	27
3.1.2.1	Pályák .....	27
3.1.2.2	mapLoader .....	29
3.1.3	Kommunikáció.....	30
3.1.3.1	Server .....	30
3.2	Kliens.....	30
	Hivatkozások.....	30



## 1. Bevezetés

### 1.1 Rövid ismertető

Szakedolgozatom egy a Portal [1] című játékhoz hasonló 2 dimenziós többszemélyes játék megvalósítása webes alkalmazásként. A játék fő eleme, hogy a pályák során különböző fejtörőket kell megoldani, teleportáció segítségével, portálokat lehet falakra helyezni. A játék fizikai rendszere szerint amilyen lendülettel áthalad egy tárgy az egyik portálon, az olyan lendülettel távozik a másikon, ez fontos szerepet fog játszani az egyes feladatok megoldásánál. A játék fizikai rendszerét magam valósítottam meg, ezzel egy egyedi fizikai szimulációt létrehozva. Az alkalmazást javascript nyelvben valósítottam meg, a többszemélyességet pedig TCP alapú web socketet [2] alkalmazva Nodejs [3] szerverrel. Többszemélyesség a játékban oly módon nyilvánul meg, hogy együttesen próbálhatják megoldani a fejtörőket a játékosok.

### 1.2 Motiváció

A 2007-ben megjelent Portal című játékkal még általános iskolában ismerkedtem meg és első pillanattól kezdve el voltam ámulva tőle. Egy számítógépes játékban valódi fizikát láttam méghozzá olyan sci-fi elemekkel, mint a teleportáció egybevetve. Abban a pillanatban tudtam, hogy aki egy ilyen dolgot meg tud valósítani az bármire képes, persze ekkor még nem tanultam programozni, így el sem tudtam képzelni, hogyan lehetne valami hasonlót megalkotni. Most, hogy egyetemi tanulmányaim vége felé közeledem egy személyes próbatételként élem meg ezt a témát a többszemélyesség megalkotása pedig csak hab a tortán, hiszen soha nem dolgoztam még sem fizikával, sem többszemélyes játékkal.

## 2. Felhasználói dokumentáció

Ez a fejezet bemutatja a játék indításának előfeltételeit, rendszerkövetelményeit, az elindításához szükséges lépéseket és további instrukciókat, majd részletesen leírja a játék működését és használatát.

### 2.1 A játék telepítése és telepítési előfeltételei

A program egy webalkalmazás, így nincs szükség külön telepítésre, viszont szükségünk van a kliens oldalon egy modern böngészőre (pl.: Google Chrome, Microsoft Edge) valamint szerver oldalon Node.js [3] keretrendszerre. A szerver és a kliens természetesen lehet azonos eszközön, viszont, ha a két külön eszközt használunk akkor biztosítani kell a hálózaton való kommunikációt.

Az optimális játékelményért az alábbi rendszerkövetelmények ajánlottak:

- OP. RENDSZER: WINDOWS 10
- PROCESSZOR: 2 GHZ
- MEMÓRIA: 2 GB RAM
- TÁRHELY: 50 MB SZABAD HELY

### 2.2 Szerver

A játék működéséhez egy dedikált szerverre van szükségünk, amit futtathatunk a saját gépünkön vagy akár egy távoli gépen is, amit elérünk az interneten keresztül.

#### 2.2.1 Konfigurálása

A szerver konfigurálása a config.js fájlban történik.

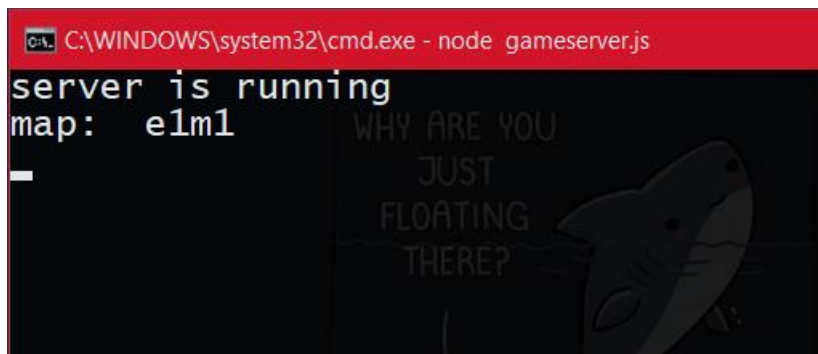
Paraméterei:

- server\_host: segítségével megadható, hogy a szerver milyen ip cím alatt fusson. Ha helyi szervert szeretnénk futtatni akkor ez nyugodtan lehet a 'localhost' érték
- server\_port: segítségével megadható, hogy a szervert melyik porton szolgáljuk ki. Itt érdemes egy 1024-nél

nagyobb számot megadni, hisz az ez alatti portok a rendszer számára kitüntettek portok.

### 2.2.2 Elindítása

A szerver elindításához Node.js [3] keretrendszerre van szükségünk, ami letölthető innen: <https://nodejs.org/en/>. Ha már rendelkezünk Node.js-el a gépünkön akkor a szerveret indíthatjuk manuálisan parancssorból: a szerver főmappájába navigálunk és beírjuk a következő parancsot: „node gameserver.js”, vagy Windows 10 alatt használhatjuk a local\_server.bat scriptet, ami ugyanezt eredményezi.



1. ábra: sikeres szerverindítás képe

Ha a szerver nem dobott hibát (1. ábra) akkor készen áll a kliensek fogadására.

Lehetséges hibák:

Hibaüzenet	Kiváltó ok
'node' is not recognised...	Nincs Nodejs feltelepítve a gépünkre
address already in use ...	A portot már egy másik alkalmazás foglalja

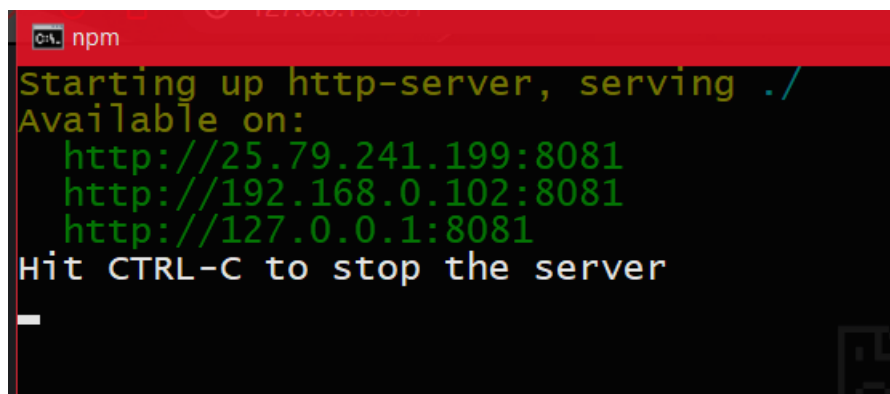
## 2.3 Kliens

### 2.3.1 Elindítása

A kliens elindításához ki kell szolgálnunk a client mappa tartalmát egy http-szerveren amit egyszerűen elvégezhetünk az npm http-server moduljával, amit itt szerezhethünk be: <https://www.npmjs.com/package/http-server>

Ha rendelkezünk, az előbb említett http-kiszolgálóval akkor a Windows 10 alatt a client mappába lévő local\_server.bat scriptet futtatva elindíthatjuk a kiszolgálót, vagy konzolból az „npx http-server” paranccsal.

Sikeres futtatás után a következőt (2. ábra) kell látnunk a konzolban:



```
npm
Starting up http-server, serving ./
Available on:
  http://25.79.241.199:8081
  http://192.168.0.102:8081
  http://127.0.0.1:8081
Hit CTRL-C to stop the server
```

2. ábra: Sikeres kliens indítás a 8081-es porton

Ha elindítottuk a kliens, akkor egy modern böngészőt megnyitva (pl. Google Chrome, Microsoft Edge) alapértelmezetten a localhost:8080 címet felkeresve, elérhetjük az applikációt.

### 2.3.2 Használat

Az oldal betöltése után a következő képnek (3. ábra) kell fogadnia minket:

## Portal 2D

### About this project

This project is my Bachelor thesis for my university studies.

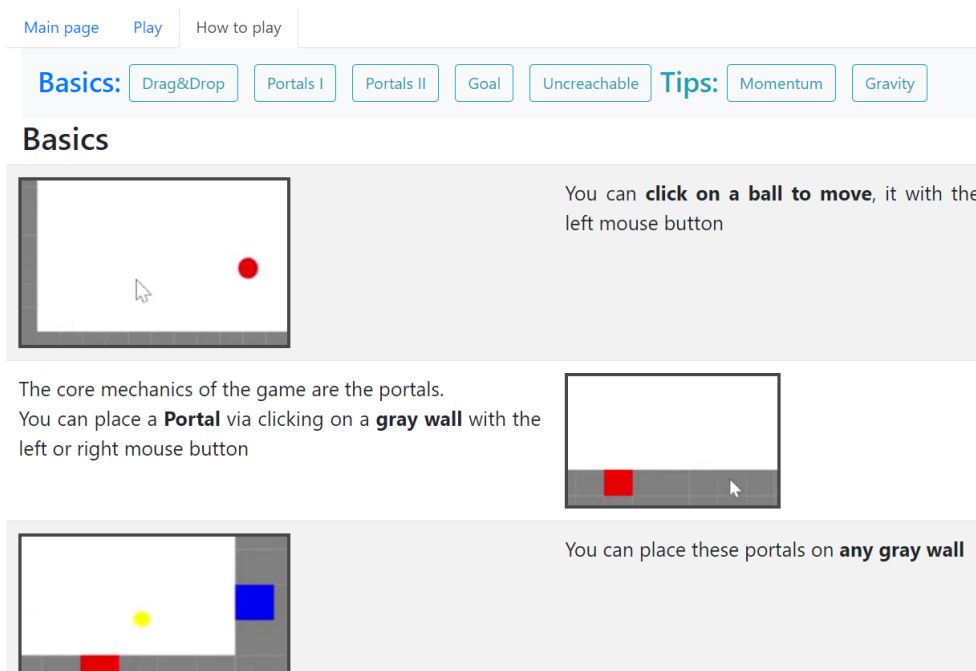
### Welcome

this is a multiplayer game based on the famous [Portal](#) game. You can play at the **Play** tab above or you can check out the tutorials at the **How to play** tab.



3. ábra: kezdőoldal

Itt a felső menüsávban válthatunk az oldalak között. Külön oldal van a játék elmagyarázására és külön magára a játékra. Ha szeretnénk megismerkedni a játék menetével és szabályaival akkor a „How to play” menüpontra kell kattintanunk, ami elnavigál minket a következő (4. ábra) oldalra:

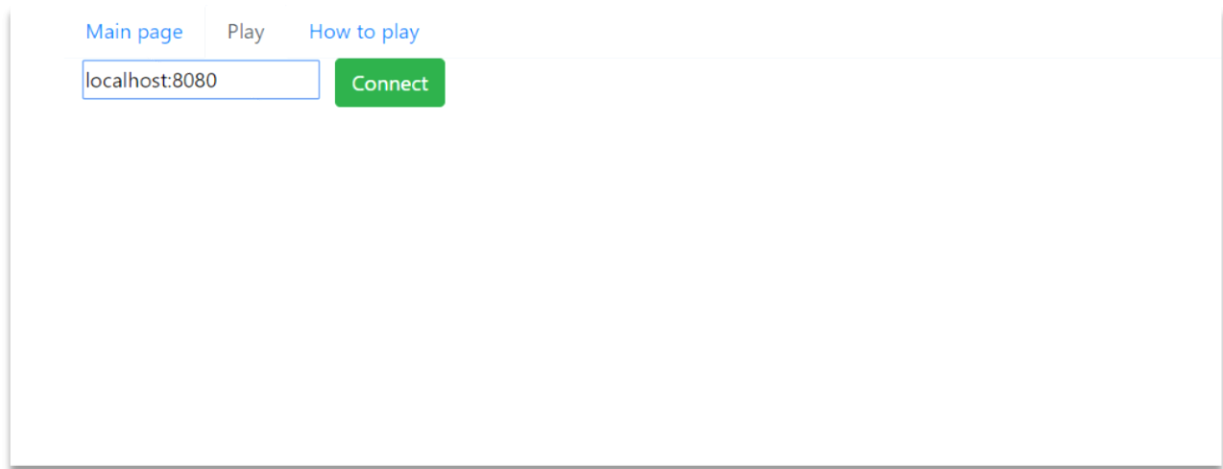


(4. ábra: betanító oldal)



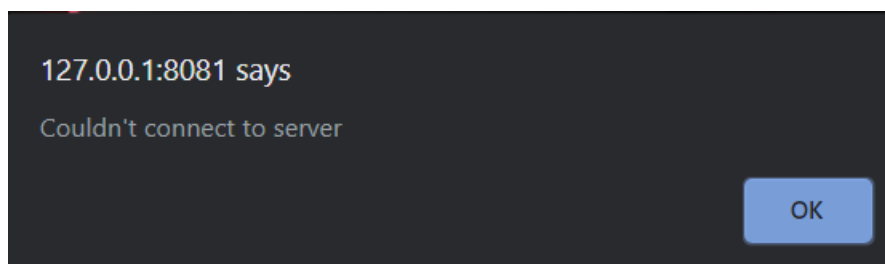
Ezen az oldalon leírásokat találunk a játék mechanikájáról, és a leírásokat videók kísérik a könnyebb megérthetőség érdekében.

A felső menüsávban a „Play” gombra kattintva érhetjük el azt a felületet, ahol a valódi játék történik. Itt egy egyszerű szövegmezővel találkozunk, ahová a szerver elérési útvonalát kell megadnunk, majd a „Connect” gombra kattintva csatlakozhatunk is a szerverhez (5. ábra).



(5. ábra csatlakozás szerverhez)

Sikertelen csatlakozás után a következő hibaüzenet fog várni minket (6. ábra):

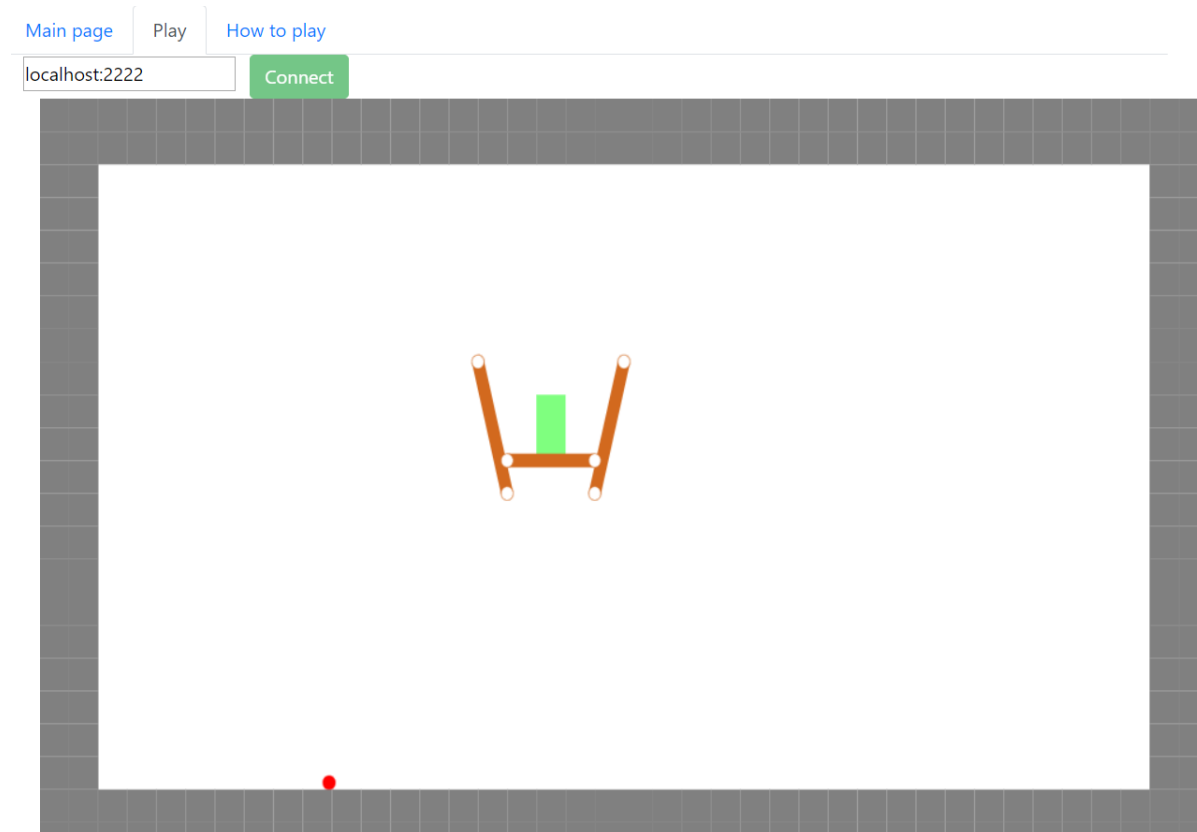


(6. ábra: sikertelen csatlakozás)

Sikertelen csatlakozást 2 hiba okozhat:

- Rossz szerver elérési útvonal
- A szerverre nem tud több kliens csatlakozni

Miután sikeresen csatlakoztunk egy szerverhez, kezdődhet is a játék, az első pálya fog fogadni minket (7. ábra).



(7. ábra: a játék első pályája)

Itt pedig alkalmazhatjuk a betanító oldal által tanult technikákat a pálya teljesítéséhez.

Ha a játék, működése közben valami oknál fogva megszakadna a kapcsolat a szerverrel, a következő hibaüzenet fog fogadni minket (8. ábra):



(8. ábra: szerver elvesztése)

Ha pedig elértük az utolsó pályát, akkor a következő kép fog fogadni minket (9. ábra):



(9. ábra: játék vége)

### 3. Fejlesztői dokumentáció

Ez a fejezet bemutatja a játék megvalósítását és annak eszközeit.

#### 3.1 Szerver

A szerver felel a játék működéséért, tehát a fizikai objektumok viselkedéséért, a játéktér létrehozásáért, elpusztításáért, valamint a kliensek fogadásáért és a velük való kommunikációért.

A szerver mappaszerkezete a következő (10. ábra):



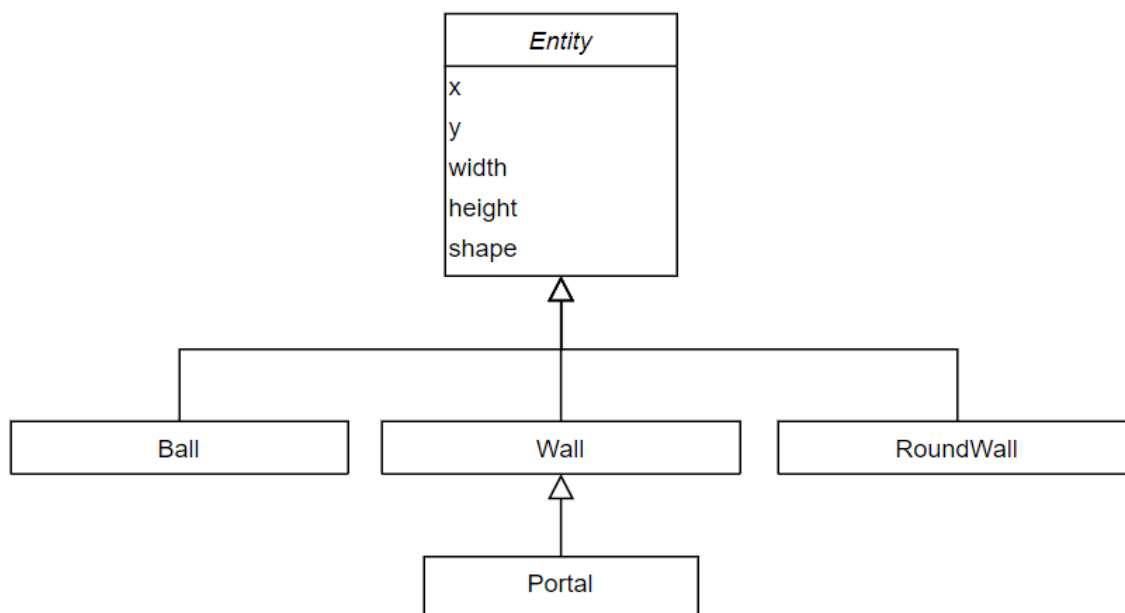
(10. ábra: szerver mappaszerkezete)

A gameserver.js fájl az egész szerver magja, ezt az egy fájlt kell futtatnunk a működéséhez, és nem tesz mást, mint felkészül a kliensek fogadására, betölti az első pályát és elkezdi a fizikai szimulációt, mindezt a server.js,

mapLoader.js és az engine.js fájlok segítségével. Erre a három fő mozgatóelemre épített bontást szeretném bemutatni az elkövetkező oldalakon.

### 3.1.1 Fizika

A fizika néhány általam megvalósított objektumon működik, amelyeket a következő diagram (11. ábra) szemléltet:



(11. ábra objektumok)

#### 3.1.1.1 Entity

Ez a fizikai őszosztály melyből az össze többi származik, ez az osztály van olyan alapvető attribútumokkal és metódusokkal felruházva, amelyekre minden egyéb fizikai szimulációban résztvevő egyednek szüksége lesz.

##### Attribútumok:

- **objID**  
azonosító szám, amely egyedi minden objektumnál
- **x, y**  
Az objektum pozíciója

- **width, height**  
Szélesség, magasság
- **shape**  
Az objektum formája (pl.: négyzet vagy téglalap)
- **vx, vy**  
horizontális és vertikális sebesség
- **mass**  
tömeg, egy képzeletbeli hozzárendelt tömeg az objektumhoz, fontos szerepet játszik például két labda ütközésénél.

Metódusok:

- **getCenter**  
Az objektum középpontját visszaadó metódus
- **getDir**  
ha mozgásban van az objektum akkor visszaadja a mozgás irányát
- **getLeft/getTop/getWidth/getHeight**  
Az objektumot körül határoló téglalap széleinek koordinátáit visszaadó metódusok
- **physicsUpdate(deltaTime)**  
ebben a függvényben kell megadnunk, hogy milyen fizikai változások történjenek az objektummal. És ez a fizikai szimuláció alapvető függvénye, ez fut le minden objektumra amikor egy szimulációs iteráció végbemegy. Van 1 paramétere is a „deltaTime”, aminek segítségével megmondhatjuk, hogy mennyi idejű mozgást kell elvégezni az adott függvényhívás alatt.

### 3.1.1.2 Ball

Ez az osztály valósítja meg a közismert labdákat. Egy labdára hat a gravitáció, ha meglöki elgurul, és a súrlódás hatására lelassul majd megáll. Ha egy labda ütközik egy másik labdával akkor azok dinamikusan ütköznek, ha pedig egy szilárd

felülettel (egyszerű és lekerekített fal) akkor arról visszapattan.

- **Konstruktor** (x, y, r, vx, vy)

Egy labda szabályos létrehozásához az első három paraméterre van szükségünk, amelyek megadják a labda középpontját és a sugarát. Az utolsó két paraméterrel pedig kezdősebességet adhatunk meg a labdánknak, ezek alapértelmezetten 0 értéket vesznek fel.

- **Mozgás**

Egy labda mozgását a fentebb említett physicsUpdate metódusában valósítottam meg a következő módon (12. ábra):

```
physicsUpdate(deltaTime) {  
  let gravityForce = engine.gravity * deltaTime;  
  this.ax = -this.vx * engine.friction;  
  this.ay = -this.vy * engine.friction * 0.5;  
  this.vx += this.ax * deltaTime;  
  this.vy += this.ay * deltaTime + gravityForce;  
  this.x += this.vx * deltaTime;  
  this.y += this.vy * deltaTime;  
  
  if (Math.abs(this.vx) < 0.01)  
    this.vx = 0  
  if (Math.abs(this.vy) < 0.01)  
    this.vy = 0  
}
```

(12. ábra: egy labda mozgása)

Itt a függvény első sorában kiszámoljuk mekkora gravitációs erő hason a labdára, a következő két sor megmondja, hogy a sebessége mennyivel változzon. Majd az elkövetkezendő 4 sorban alkalmazzuk is a kapott értékeket. Ez mind az eltelt idő „deltaTime” függvényében történik, és a függvény végén található 2 elágazás, amelyek segítenek abban, hogy a nagyon kicsi mozgásokkal ne törődjünk, azokat úgy sem tudjuk megjeleníteni.

Ez a mozgásmegvalósítás nem valódi fizikai képletek alapján működik, csak egy látszólagos hihető közelítése.

Ha kívülről szeretnénk befolyásolni egy adott labda mozgását akkor csak a labda  $v_x$  és  $v_y$  attribútumait kell módosítanunk a kívánt értékekre.

### 3.1.1.3 Wall

Egyszerű téglalap alakú fal megvalósítása, amelyen a labdák nem tudnak áthaladni, hanem visszapattannak róla.

- **Konstruktor** ( $x$ ,  $y$ ,  $w$ ,  $h$ )

Első két paraméter a fal bal felső sarka, majd a szélesség és magasság. Ha nincs magasság megadva akkor alapértelmezetten négyzetnek fogja megvalósítani a fal alakját.

- **getCorners**

Egy segédfüggvény amely egy A,B,C,D négyessel tér vissza, amelyek rendre a fal sarok koordinátáit jelölik jobb felülről kezdve óramutatóval ellentétes bejárással.

### 3.1.1.4 RoundWall

Lekerekített fal, a sima téglalap alakú fallal megegyezően viselkedő fal rész, annyi különbséggel, hogy a végpontjai le vannak kerekítve és állása lehet ferde is (13. ábra).



(13. ábra: egy példa a lekerekített falra)



- **Konstruktor** (sx, sy, ex, ey, r)
  - o sx és sy a fal kezdetének x és y koordinátája
  - o ex és ey pedig a fal végének a koordinátái
  - o r pedig a fal szélessége

### 3.1.1.5 Portal

A portál a játék fő eleme, ez egy olyan objektum, amelyekből egyszerre 2-öt tudunk falakra (Wall) helyezni és ha az egyikbe „belemegy” egy labda akkor az a másikon kijön megfelelően elforgatva a sebességvektoraival.

- **Konstruktor** (x, y, w, playerID, color)

Egy portál kinézet ügyileg megegyezik egy négyzetes fallal, így az első 3 paraméter megegyezik a Wall konstruktorával, ezen kívül megadható, hogy melyik játékos birtokolja, és hogy milyen színű legyen a portál. A játékban most egyelőre csak a kék és a piros színű portálok játszanak szerepet, ők alkotnak egy part, ha a ugyan azon játékoshoz tartoznak.

A portál konstruktorában hozzá rendelünk egy kisebb négyzetet, ez lesz az a négyzet, amivel vizsgálni fogjuk, hogy találkozik-e vele labda, ha igen akkor úgy tekintjük, hogy az belement a portálba. Ennek az attribútumnak a neve „portalRect”.

Ezután a „portalWall” attribútumban elmentjük azt a falat amire rá lett rakva a portál és megváltoztatjuk az alakját, hogy most már ne ütközessenek vele a labdák.

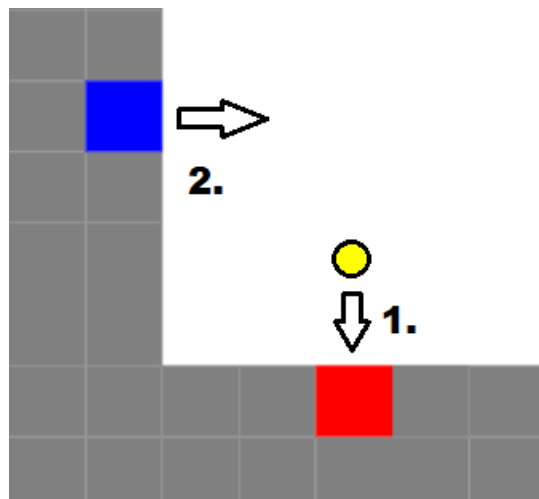
Ezután pedig eldöntjük, hogy merre van a „kijárat” a portálból, oly módon, hogy megnézzük hol vannak szomszédos falak a portál mellett és ha csak az egyik oldalt nincs fal akkor az az oldal a kijárat.

Ha ez utóbbi feltétel nem teljesül, tehát nincs egyértelmű kijárat akkor a portál nem is fog működni, ezzel próbáltam elkerülni a nemdeterminisztikus viselkedést.

- **physicsUpdate**

Egy objektum sajátos viselkedését az objektum „physicsUpdate” függvényében írhatjuk le, ez rendszeresen le fog futni minden egyes alkalommal amikor fizikát számolunk.

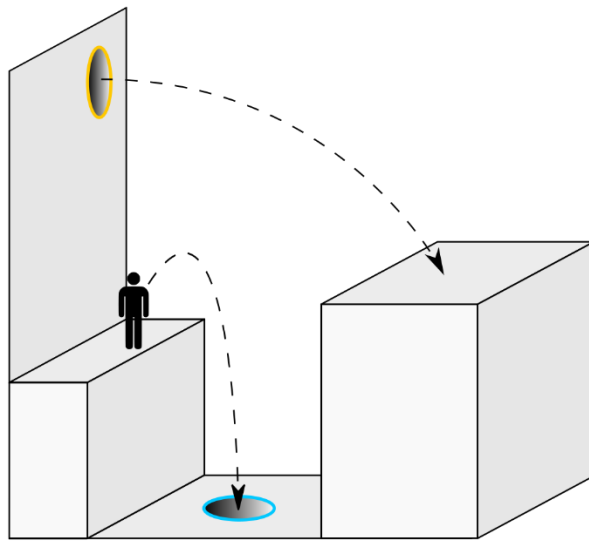
A Portálok sajátos viselkedése, hogy ha egy labda belemegy a portálba akkor azt át kell teleportálni a portál párjához és a labda sebességvektorait megfelelően elforgatni. Megfelelő elforgatás alatt pedig azt értem hogyha például mindkét portál a padlóra van elhelyezve, akkor amikor beleesik egy labda az egyikbe, akkor mikor átteleportáljuk a másik portálhoz, a vertikális sebességét az ellenkezőjére kell állítani. Ha pedig az egyik portál a talajon a másik pedig az egyik oldalsó falon fal, akkor a vertikális és horizontális sebességeket fel kell cserélni, és úgy fordítani, hogy a kijárat felé mozogjon a labda. De nézzünk meg egy példát (11. ábra) a könnyebb érthetőség végett.



(11. ábra: példa a portál viselkedésére)

Itt tegyük fel, hogy a sárga labdát csak a gravitáció húzza lefelé, így a horizontális sebessége 0 a vertikális pedig egy pozitív szám. Amikor beleér a piros portálba el kell mozgatnunk a kék portál közepéhez képest egy kicsit jobbra, hiszen arra van a kijárat. Az sebességei pedig

úgy módosulnak, hogy a horizontális sebessége lesz az eddigi vertikális, a vertikális sebessége pedig az eddigi horizontális. Így amikor átteleportáltuk, a labda jobbra fog mozogni, mint ahogy neki kell is. Ily módon a portálok megtartják az objektum momentumát, csak az irányát forgatják el mindig a kijáratnak megfelelően, így használhatjuk az eredeti „Portal” játékban látott (12. ábra) logikai fejtörők megoldásához.



(12. ábra: fejtörő a Portal játékból)<sup>1</sup>

#### 3.1.1.6 Fizika motor

A játék fizikai szimulációja saját kezűleg lett megvalósítva a fentebb említett engine.js fájlban.

Az engine.js fájl néhány beégettet paraméterrel (13. ábra) kezdődik, amelyek teljes mértékben meghatározzák a fizika viselkedését a szimuláción belül:

---

<sup>1</sup> [https://upload.wikimedia.org/wikipedia/commons/thumb/c/c0/Portal\\_physics-2.svg/1024px-Portal\\_physics-2.svg.png](https://upload.wikimedia.org/wikipedia/commons/thumb/c/c0/Portal_physics-2.svg/1024px-Portal_physics-2.svg.png)

```

var engine = {
  lastTick: performanceNow(),
  deltaTime: null,
  physicsPrecision: 1 / global.physicsPrecision,
  gravity: 0.8 * 1000,
  friction: 0.9,
  index: {},
  indexSize: 256,
  shapes: {
    CIRCLE: "circle",
    RECTANGLE: "rectangle",
    ROUNDRECTANGLE: "roundrectangle",
    SQUARE: "square",
    DESTINATION: "destination",
    PORTAL: "portal",
    UNREACHABLE: "unreachable",
  },
};

```

(13. ábra: a fizika paramétereit)

Néhány fontosabb paraméter elmagyarázása:

PARAMÉTER	HATÁS
gravity	milyen gyorsan esnek le a labdák
friction	Milyen gyorsan vesznek el a horizontális sebességüket
physicsPrecision	Másodpercenként hányszor számoljon fizikát

A többi paraméterre később kerül sor használat közben a könnyebb érthetőség végett.

A motor fő mozgatórugója a „simulatePhysics” nevű függvény, amelynek egy paramétere azon objektumok tömbjét várja, akikkel szeretnénk fizikát szimulálni.

- **simulatePhysics**

a függvény az első pár sorában megvizsgálja, hogy mennyi idő telt el mióta utoljára meg lett hívva, és az eltelt idő függvényében állítja a „loopindex” nevű változót, ami azért felel, hogy a két függvényhívás között bármennyi

idő teljen is el, mindig azonos precizitású legyen a szimulációnk, ezzel elérve, hogy ha egy gyengébb eszközön futtatjuk a szerveret, akkor sem fog furcsán viselkedni a fizikai szimuláció.

Ezután a paraméterében kapott összes objektumra, meghívja a `physicsUpdate` nevű függvényt, ami az objektumok sajátos viselkedésükért felel, majd megvizsgálja, hogy bármely két objektum ütközik-e és ha igen leszimulálja a megfelelő viselkedésüket.

Az ütközésvizsgálat egy térbeli indexeléssel kezdődik, amelyet a „`createIndex`” metódus valósít meg.

- **`createIndex`**

A térbeli indexelés célja, hogy ne kelljen minden objektum minden másik objektummal való ütközését vizsgálni, mert az nagyon költséges sok objektum esetén, hanem vizsgáljuk csak az objektum környezetébe eső objektumokat.

Ennek eléréséhez számon kell tartani, hogy mely objektum a játéktér melyik részén helyezkedik el. Ezt úgy valósíthatjuk meg, hogy felosztjuk az egész játékteret négyzet alakú cellákra, amelyek mérete az engine „`indexSize`” változójában állítható. És minden egyes objektumról feljegyezzük, hogy mely cellákba lóg bele, így amikor ütközést vizsgálunk elég megnézni azon objektumokkal való ütközést, amelyekkel van közös cella.

Fontos a cellaméret jó megválasztása hiszen túl kicsi cellaméret esetén az objektumok nagyon sok cellába belelőgnak és keresésnél ez sok cella vizsgálata, túl nagy cellaméretnél, pedig egy cellában túl sok objektum lesz. A számunkra megfelelő méretet próbálgatással megtalálhatjuk, én a 256-os méretet megfelelőnek találtam tesztjeim során.

A „`createIndex`” függvény egy olyan objektummal tér vissza, amelynek vagy egy hívható metódusa, a „`query`”

melynek segítségével visszacapjuk azon objektumok halmazát, amelyek metszik az általunk paraméterben megadott objektumot.

- **query**

ez a metódus a „createIndex” által létrehozott térbeli index segítségével végighalad az összes cellán, amelyet érint a paraméterben megkapott objektum, és minden talált objektumra eldönti ütköznek-e és ha igen az ütköző elemeket egy halmazba rakja és visszatér vele. Az ütközés eldöntése a „doEntitiesIntersect” függvényben történik, amely esetszétválasztással pontosan el tudja dönteni, hogy két síkbeli test ütközik-e.

- **doEntitiesIntersect**

Ez a függvény felelős eldönteni, hogy két objektum ütközik-e. A két vizsgálandó objektumot a függvény paraméterként kapja meg, és az objektumok „shape” attribútuma segítségével eldönti, hogy hogyan kell ütközést vizsgálni. Két tetszőleges téglalap esetén a „rectangleIntersect” -nek adja tovább a két objektumot, két labda esetén a „circleIntersect” -nek, labda és téglalap esetén a „rectCircleColliding” -nak és labda és lekerekített fal esetén pedig a „roundRectCircleIntersect” nevű függvénynek adja át a két objektumot.

- **rectangleIntersect**

Egy egyszerű eldöntés, hogy a két téglalapnak van-e metszete

- **circleIntersect**

Megvizsgálja, hogy a két kör középpontjának a távolsága kisebb-e, mint a két kör sugarának összege.

- **rectCircleColliding**

Megvizsgálja, hogy egy téglalap és kör metszi-e egymást. A függvény első két sora kiszámolja az  $x$  és az  $y$  koordinátáik eltérését majd, ha valamelyik nagyobb, mint a kör átmérője, akkor egyértelmű, hogy nem metszik egymást.

Ezután, ha a távolság kisebb, mint a téglalap fele és a sugár összege, akkor biztosan metszik egymást. Ezután már csak azt kell megvizsgálni, hogy a sarkokat metszi-e a kör, ezt pedig a Pitagorasz-tétellel egyszerűen ellenőrizzük az utolsó sorban.

- **roundRectCircleIntersect**

Ez a függvény pedig a lekerekített fallal való ütközését vizsgálja a labdának, oly módon, hogy megkeresni a labdához legközelebbi pontot a fal középegynesén, és azon a helyen megvizsgálja, hogy egy labda, amelynek sugara megegyezik a fal szélességével arra a helyre elhelyezve, találkozna-e az mi körünkkel, ha igen akkor a fal is ütközik a körrel. Ezt a kódrészletet Javidx Github oldalán találtam [4].

Miután megtörtént az ütközésvizsgálat, le kell szimulálnunk az ütközéseket, amikért az engine „handleCollision” metódusa a felelős.

- **handleCollision**

A „handleCollision” függvény a „doEntitiesIntersect” függvényhez hasonlóan az objektumok alakjainak függvényében esetszétválasztással kezeli le az ütközéseket. Ha két labda ütközik akkor a „ballBallCollision” fut le, ha a labda egy téglalap alakú fallal ütközik akkor a „rectBallCollision”, ha a négyzet alakú fallal akkor „squareBallCollision”, ha pedig lekerekített fallal akkor a „roundRectBallCollision” fog lefutni.

- **ballBallCollision**

Ez a metódus két labda rugalmas ütközését valósítja meg, melynek képletét (14. ábra) a Wikipédián találtam meg [5].

$$\mathbf{v}'_1 = \mathbf{v}_1 - \frac{2m_2}{m_1 + m_2} \frac{\langle \mathbf{v}_1 - \mathbf{v}_2, \mathbf{x}_1 - \mathbf{x}_2 \rangle}{\|\mathbf{x}_1 - \mathbf{x}_2\|^2} (\mathbf{x}_1 - \mathbf{x}_2),$$

$$\mathbf{v}'_2 = \mathbf{v}_2 - \frac{2m_1}{m_1 + m_2} \frac{\langle \mathbf{v}_2 - \mathbf{v}_1, \mathbf{x}_2 - \mathbf{x}_1 \rangle}{\|\mathbf{x}_2 - \mathbf{x}_1\|^2} (\mathbf{x}_2 - \mathbf{x}_1)$$

(14. ábra: két labda dinamikus ütközése)<sup>2</sup>

De mielőtt a dinamikus ütközést elvégeznénk, el kell végeznünk egy „statikus eltolást” ami azt jelenti, hogy a két labda csak úgy ütközhetett, hogy egymásba lógnak, de ilyen a valóságban nem történhet meg, ezért korrigálni kell, meghozzá úgy hogy vesszük a két labda középpontját összekötő egyenest és ezen az egyenesen mindkét labdát arrébb pozicionáljuk az átfedési távolság felével a saját irányába, így pont ahhoz a ponthoz jutunk ahol a valóságban ütköznének, és ezután már végezhetjük is a dinamikus ütközést.

- **squareBallCollision**

A négyzet alakú fal és a labda ütközésének viselkedését egy általam kitalált módszerrel szimulálom, ami a következő:

A két labda ütközéséhez hasonlóan itt is egy „statikus eltolással” kezdünk, amihez segítségünkre van a „squareBallOverlap” nevű függvény, amely megmondja, hogy

---

<sup>2</sup>

[https://wikimedia.org/api/rest\\_v1/media/math/render/svg/14d5feb68844edae9e31c9cb4a2197ee922e409c](https://wikimedia.org/api/rest_v1/media/math/render/svg/14d5feb68844edae9e31c9cb4a2197ee922e409c)



a labda mely oldalról és mennyivel hatolt bele a négyzetbe. Ez alapján pozicionáljuk a labdát a fal mellé, majd annak függvényében, hogy melyik oldalról ütközik a labda a falnak, megváltoztatjuk a sebességvektorainak irányát. (Például, ha felülről ütközött a falnak, akkor tudjuk, hogy az  $y$  irányú sebességét kell az ellenkezőjére változtatni.) Azt, hogy melyik oldalról ütközött a falnak azt oly módon döntöm el, hogy veszem a labda középpontja és a négyzet középpontja által alkotott vektor irányszögét és megnézem, hogy melyik síknegyedbe esik. Tehát ha a szög  $45^\circ$  és  $135^\circ$  közé esik akkor tudom, hogy felül ütközött a fallal a labda, a többi oldalt ehhez hasonlóan.

- **rectBallCollision**

Nagyon hasonló az előbb kifejtett „squareBallCollision” -hoz annyi különbséggel, hogy most azt, hogy melyik irányból jött a labda nem lehet úgy eldönteni, hogy síknegyedeket vizsgálunk, hiszen a fal már nem szabályos négyzet alakú. Mivel itt már téglalappal dolgozunk, így a  $45^\circ$  helyett ki kell számolnunk a fal középpontja és a jobb felső sarok által meghatározott vektor irányszögét és ez az érték fogja helyettesíteni az előző példában a  $45$  fokot. És ehhez hasonlóan mind a négy sarok irányszögét ki kell számolnunk, és csak ezután tudjuk, behatárolni, hogy a labda mégis melyik oldalról ütközött a falnak.

- **roundRectBallCollision**

Hasonló a „roundRectCircleIntersect” függvényhez. Miután megtalálta az ütközés pontját, úgy tesz mintha abban a pontban találkozott volna egy másik labdával és egy labda-labda ütközés van megvalósítva itt is, ugyanúgy, mint a „ballBallCollision” -ben.

A fizika az eddig említett függvények alapján dolgozik, de a motor még rendelkezik néhány segédfüggvénnyel:

- **nearest** (x, y, obj)

Ez a függvény visszaadja az x, y koordinátához legközelebb lévő „obj” típusú objektumot, ha létezik ilyen. Úgy működik, hogy végig iterál az összes objektumon és ha az adott objektum típusa megfelelő akkor elvégez egy minimum kiválasztást rajtuk.

- **place\_meeting** (x, y, w, h, obj)

Paramétereiben megkapja egy téglalap adatait és egy objektum típusának a nevét, majd megkeres egy ilyen típusú objektumot ezen a területen és visszatér vele ha létezik.

- **point\_meeting** (x, y, obj)

A „place\_meeting” -hez hasonlóan működik annyi különbséggel, hogy most téglalap helyett, csak egy pontban vizsgálja az adott típusú objektum létezését.

Ezzel a játék fizikai motorjának a végéhez értünk, és láthatjuk, hogy egy nagyon kényelmesen használható rendszert sikerült létrehozni. Ha új objektumokat szeretnénk hozzáadni a fizikához egyszerűen csak az eddig említett objektumok tömbjéhez kell egy újat hozzáadni, és kitörölni is olyan egyszerű, mint kivenni ebből a tömbből. Ha pedig egy objektum sajátos viselkedését szeretnénk módosítani, azt maga az objektum „physicsUpdate” metódusában tehetjük meg.

Ha létre hozunk egy új labdát még olyan vizsgálatot se kell elvégezni, hogy az adott pontban van e másik labda, nehogy egymásba rakjuk a kettőt, hiszen a „statikus eltolás” megoldja az elpozícionálásukat. A labdák mozgatásához pedig elég a sebességvektoraikat konfigurálni, és a többit megoldja a rendszer.

### 3.1.2 Játéktér

Hogy a játék játszható legyen szükségünk van pályákra, pályák közti léptetésre és célra. A pályákért felelős a „mapLoader” nevű osztály, míg a pályákon elérendő célt a „Destination” nevű osztály valósítja meg. Ezen kívül a játékban még megjelenik egy piros terület, ahová nem lehet labdákat mozgatni, ezeket a területeket az „Unreachable” osztály valósítja meg.

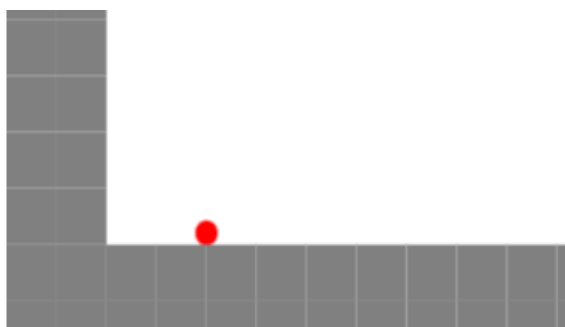
#### 3.1.2.1 Pályák

A pályákat a „maps/maps.js” fájlban tároljuk egy „maps” nevű objektumban, és ezen belül a pályák azonosítói maguk a pályanevek így például az első pályát a „maps.elm1” alatt érjük el. Az elnevezés az „episode 1 map 1” rövidítése és konzisztensen a rákövetkezendő pálya neve „elm2”, majd „elm3” és hasonlóan a többi. Minden pályának a „body” attribútumában van elmentve maga a pálya adatai a következőképpen:

```
[W, W, _ , _ , _ , _ , _ , _ , _ , _ ,  
[W, W, _ , _ , _ , _ , _ , _ , _ , _ ,  
[W, W, _ , _ , _ , _ , _ , _ , _ , _ ,  
[W, W, _ , _ , B, _ , _ , _ , _ , _ ,  
[W, W, _ , _ , _ , _ , _ , _ , _ , _ ,  
[W, W, _ , _ , _ , _ , _ , _ , _ , _ ,  
[W, W, W, W, W, W, W, W, W, W, W,  
[W, W, W, W, W, W, W, W, W, W, W,
```

(15. ábra: pályarészlet)

Az ábrán (15. ábra) egy pályarészlet látható az első pálya bal első sarkáról. Egy több dimenziós tömbben objektumok kezdőbetűi láthatóak: „W” a „Wall” és „B” a „Ball” megfelelője. Így egy grafikus pályaszerkesztőhöz hasonló képet kapunk a keletkezendő pályáról. A keletkezett pálya a következő ábrán látható (16. ábra)



(16. ábra: a 15. ábra eredménye)

(Megjegyzés: a labda azért van a földön és nem a levegőben mert a gravitáció már lehúzta.)

Ez a megvalósítás azért előnyös, mert a program futtatása nélkül már előre láthatjuk, hogy nagyjából hogyan fog kinézni a végeredmény, meggyorsítja és átláthatóbbá teszi a pályák készítését, szerkesztését.

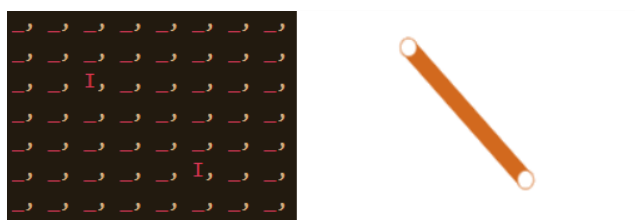
A pályaszerkesztőben a következő rövidítések használhatjuk:

Jelölés	Objektum
—	Üres
<b>W</b>	Sima négyzetes fal
<b>D</b>	Cél
<b>U</b>	Tiltott terület
<b>P,O,I,Z,T,G,H,J,K,L</b>	Lekerekített fal

A Cél a Tiltott terület és a Lekerekített fal elhelyezése eltérő az egyszerű falétól a következőképpen:

- Lekerekített fal

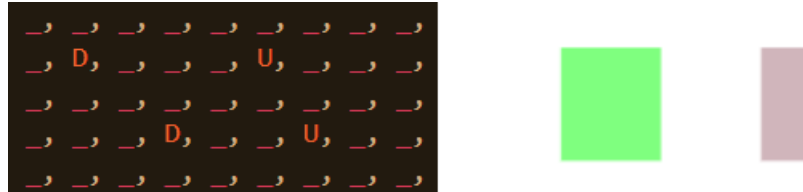
Mivel ezek lehetnek ferde állásúak is így 2 pont jelöli a helyzetüket, meg kell adni egy kezdő és egy végpontot. A következő ábra (17. ábra) ennek a működését mutatja be:



(17. ábra: lekerekített fal elhelyezése és megjelenése)

- Cél és Tiltott terület

Ezen objektumok pedig téglalap alakúak így a téglalap bal felső és jobb alsó sarkát kell megjelölni a következő módon(18. ábra):



(18. ábra: Cél és Tiltott terület)

### 3.1.2.2 mapLoader

Ez az osztály felelős a pályák betöltéséért és a kliensek értesítése a pályaváltásról.

- **konstruktor**

A konstruktor nem csinál mást, mint beállítja a jelenlegi pályát az „elm1” nevű pályára.

A pálya beállítása a „setMap” metódusban történik

- **setMap**

Egy pálya betöltése előtt törölni kell a jelenleg pályán lévő objektumokat, így a függvény első 3 sora ezt is teszi: kiüríti az objektumot tömbjét és értesíti a klienseket is arról, hogy az objektumok törlődtek a websocket „sendRemoveAll” és „clearportals” metódusainak segítségével, ezekről a metódusokról később lesz szó.

Ezután megvizsgálja, hogy a paraméterben kapott pályanév létezik-e a pályák között és betölti, ha nem létező pályanevet adtunk meg, akkor betölti a legelső pályát.

Majd értesíti a klienseket az új pályáról a „sendCreateAll” metódussal, erről is később lesz szó.

A pálya betöltéséért a „processMap” metódus felel.

- **nextMap**

Betölti a következő pályát.

Mivel a pályanevek konzisztensek és csak a nevük végén lévő utolsó számban különböznek így, ha megnöveljük a jelenlegi pálya nevének a sorszámát megkapjuk a következő pályát, vagy ha az utolsó pályán vagyunk akkor érvénytelen pályanevet kapunk, de ezt a „setMap” metódus úgy kezeli le, hogy betölti az első pályát.

- **processMap**

A „Pályák” részben taglalt fájlból betöltött pályát mint több dimenziós tömböt feldolgozza az említettek szerint.

### 3.1.3 Kommunikáció

A kliensekkel való kommunikációt a „server.js” fájlban található „Server” osztály valósítja meg.

#### 3.1.3.1 Server

- Konstruktor

## 3.2 Kliens

## Hivatkozások

- [1] „WebSocket,” 10 05 2020. [Online]. Available: <https://javascript.info/websocket>.
- [2] „Nodejs,” 11 05 2020. [Online]. Available: <https://nodejs.org/en/about/>.
- [3] „Portal,” 11 05 2020. [Online]. Available: [https://en.wikipedia.org/wiki/Portal\\_\(video\\_game\)](https://en.wikipedia.org/wiki/Portal_(video_game)).
- [4] „Javidx,” 20 06 2020. [Online]. Available: [https://github.com/OneLoneCoder/videos/blob/master/OneLoneCoder\\_Balls2.cpp](https://github.com/OneLoneCoder/videos/blob/master/OneLoneCoder_Balls2.cpp).
- [5] „Wikipédia rugalmas ütközés,” [Online]. Available:

[https://en.wikipedia.org/wiki/Elastic\\_collision](https://en.wikipedia.org/wiki/Elastic_collision).