Szakdolgozat

Miskolci Egyetem

2D kalandjáték tervezése és fejlesztése Phaser játék motorral

Készítette:

Czikó Tivadar

Programtervező informatikus

Témavezető:

Dr. Árvai-Homolya Szilvia

Miskolc, 2023

Miskolci Egyetem
Gépészmérnöki és Informatikai Kar
Alkalmazott Matematikai Intézeti Tanszék **Szám:**

Szakdolgozat Feladat

Czikó Tivadar (O2IXLB) programtervező informatikus jelölt részére.

A szakdolgozat tárgyköre: 2D játék fejlesztés, JavaScript, Phaser, HTML5.

A szakdolgozat címe: Adventure játék tervezés és fejlesztés Phaser játék motorral

A feladat részletezése:

A hallgató feladata, hogy irodalomkutatást végezzen a modern HTML5 technológián alapúló 2D oldalnézetes HTML5 alapú játékok készítésének téma körében. Tervezze meg a játékot, és valósítsa meg annak a működő prototípusát. Az implementációt JavaScript programnyelven készítse el.

Mutassa be az elérhető technológiákat, hasonlítsa össze őket és válasszon olyan megoldást, amely segítségével egy prototípus alkalmazást fog készíteni.

Tervezze meg a játékot, és valósítsa meg annak a működő prototípusát. Az implementációt JavaScript programnyelven készítse el.

Témavezető: Dr. Árvai-Homolya Szilvia, egyetemi docens, Analízis Intézeti Tanszék
Konzulens(ek): Árvai László, tanársegéd, Általános Informatikai Intézeti Tanszék

A feladat kiadásának ideje: 2022. Március 15.

. . . . . .
szakfelelős

Eredetiségi Nyilatkozat

Alulírott **Czikó Tivadar**; Neptun-kód: `O2IXLB` a Miskolci Egyetem Gépészmérnöki és Informatikai Karának végzős Programtervező informatikus szakos hallgatója ezennel büntetőjogi és fegyelmi felelősségem tudatában nyilatkozom és aláírásommal igazolom, hogy *Adventure játék tervezés és fejlesztés Phaser játék motorral* című szakdolgozatom saját, önálló munkám; az abban hivatkozott szakirodalom felhasználása a forráskezelés szabályai szerint történt.

Tudomásul veszem, hogy szakdolgozat esetén plágiumnak számít:

- szószerinti idézet közlése idézőjel és hivatkozás megjelölése nélkül;
- tartalmi idézet hivatkozás megjelölése nélkül;
- más publikált gondolatainak saját gondolatként való feltüntetése.

Alulírott kijelentem, hogy a plágium fogalmát megismertem, és tudomásul veszem, hogy plágium esetén szakdolgozatom visszautasításra kerül.

Miskolc, ... év . . . . . hó . . . . . nap

. . . . . .

Hallgató

1. A szakdolgozat feladat módosítása szükséges (módosítás külön lapon) / nem szükséges

. . . .

dátum témavezető(k)

2. A feladat kidolgozását ellenőriztem:

témavezető (dátum, aláírás): konzulens (dátum, aláírás):

. . . .

. . . .

. . . .

3. A szakdolgozat beadható:

. . . .

dátum témavezető(k)

4. A szakdolgozat . . . . szövegoldalt

. . . . program protokollt (listát, felhasználói leírást)

. . . . elektronikus adathordozót (részletezve)

. . . .

. . . . egyéb mellékletet (részletezve)

. . . .

tartalmaz.

. . . .

dátum témavezető(k)

5. A szakdolgozat bírálatra bocsátható / nem bocsátható

A bíráló neve: . . . .

. . . .

dátum szakfelelős

6. A szakdolgozat osztályzata

a témavezető javaslata: . . . .

a bíráló javaslata: . . . .

a szakdolgozat végleges eredménye: . . . .

Miskolc, ...

a Záróvizsga Bizottság Elnöke

Tartalomjegyzék

1. fejezet

Bevezetés

Az informatika világa tele van számos kihívással egy ember számára, olyan, mint egy véget nem érő szabaduló szoba, melyben számomra a kreativitás, az alkotás, és a problémamegoldás ötvöződik benne,mint egy végtelen szinttel rendelkező logikai játék, ami számos lehetőséget rejt. Mindig is érdekelt, hogyan működnek a számítógépek, és hogyan lehet őket használni az emberi élet minden területén a fejlődés elősegítésére. Számomra Programtervező informatikusként mind a programok és a számítógépek hardveres része meghatározó bár hozzám a programok világa közelebb áll. Az egészen gyermek koromat végig kísérte mind szórakozásban és a tanulásterén, az olyan programok mind a Manó matek, Manó angol, Comenius Logo, amelyek életbe vágóak voltak, hogy megszeressem ezt a világot, már egészen fiatalon. Éveken át tanultam, fejlesztettem és tapasztaltam az informatika izgalmas világát, most pedig itt az ideje, hogy ezt a tudást megosszam a szakdolgozatomon keresztül. A prog-

ramtervező informatikus szakon szerzett tudásom egészen szerteágazó. Minden ágában kipróbálhattam magam és tapasztalatot szerezhettem. Az algoritmusok, a programozás, a szoftvertervezés, adatbázisok és a hálózati rendszerek fejlesztése és több különböző programozási nyelv tanulása során, mindegyik olyan terület, amely elő segítette szakmai fejlődésemet. Sikerült különböző projekteken dolgoznom, csapatban vagy akár egyedül és megtapasztalni azt, hogy mi fog várni az Alma Materem befejezése után. A szakdolgozatom lehetőséget ad számomra, hogy elmélyítsem és továbbfejlesszem ezeket a készségeket, valamint hogy új területeket fedezzek fel, amelyekben hasznosítani tudom az egyetemen tanult tudásomat. Az informatikai világának gyors fejlődése és folyamatos változása lehetővé teszi számomra,

hogy hozzájáruljak a jövőbeli technológiák fejlesztéseihez. Szeretném megtervezni és létrehozni olyan programokat vagy rendszereket, amelyek megkönnyítik az emberek mindennapi életét, és hozzájárulnak a társadalom fejlődéséhez. Ezenkívül érdekelnek az olyan területek, mint a mesterséges intelligencia és a természetes nyelvek feldogozása, amelyben a szakmai gyakorlatom elvégzése során tapasztalatot szerezhettem, habár a szakdolgozatom témája a játékfejlesztés lesz. A dolgozatom témája egy HTML5 2D oldalnézetes/féloldalnézetes játék

tervezése és fejlesztése, melynek az implementációját JavaScript programnyelven képzeltem el. Több szintes platformer/kaland játék volt az alapkoncepció, melyben a főhősünket irányítva végig kell jutni a pályán, ellenségekkel kell megküzdeni, érméket kell gyüjteni. Hosszas kutatás után úgy döntöttem, hogy a phaser.js segítségével fogom elkészíteni a játékomat, mert számos dokumentáció találató az interneten és a phaser hivatalos is nagyon felhasználó barát, ahol számos segítseget kód részlet segíti a kezdő phaser fejlesztőt. A Phaser.js az egyik leg-

ismertebb JavaScript könyvtár, amely segítnekünk a játékok elkészítésében. Ebben a könyvtárban számos függvény és összetevő megtalálható, ami segíti munkánk előre haladását. A Phaser mint 2D fejlesztő környezet már 10 éve létezik és számos változata jelent meg. A Jelenleg a legújabb a 3.7.0 Legelőször 2013-ban jelent meg és azóta is folyamatosan fejlesztik és frissítik a könyvtárát. Mivel már egy ideje elérhető, azt gondolnánk nem annyira sűrűn frissítik a könyvtárát vagy már meg álltak a fejlesztésében, de ez nem igaz. Munkám kezdetén 3.5.5 verzió számú volt a legfrissebb. A Phaser-re nagyjából féléventi kijön egy nagyobb frissítés, de kisebb frissítések másfél havonta jelennek meg. Ha a legújabb verzióval szeretnénk rendelkezni, akkor, lehet a kódunkat is át kell majd alakítani, hogy újból működőképes legyen a program. Ennek következtében nagyon oda kell figyelnünk, hogy mikori dokumentációkból, fórumokról és kódokból dolgozunk. Szerencsér Phaser oldalán vissza tudjuk nézni a korábbi verziók által elkészt kódokat, hogy még is miben változott az új könyvtár és Példákkal is megmutatja egy felhasználó barát kezelő felületen. Ami számomra további nehézség

volt a phaserrel kapcsolatban, hogy nem találtam refaktorálásra olyan példát, a saját weboldalán ami különböző phaser fájlokat köt össze. Emiatt számos phaser kódott, phaser-rel készült játéknak a kódját néztem át és arra következtetésre jutottam, hogy a phaserben számos különböző módon meg lehet valósítani. Az objektum orientáltság alapelveit. A Phaser-t

mint JavaScript alapú keretrendszert széleskörben használják, számos Facebook-os, egyébb internetes weboldalon találhatunk játékokat amelyeket ezzel a technológiával készültek.

2. fejezet

2D játékok megvalósítása

Ebben fejezetben azt fogom bemutatni, hogy milyen módszerekkel lehet játékot készíteni és milyen technológiákkal, programopzási nyelvekkel lehet megvalósítani.

2.1. Különböző technológiák bemutatása

Amint a bevezetésben is említettem a szakdolgozatom témája egy 2D kaland játék elkészítése. Több féle technológia közül választhatunk. A legnépszerűbbeket, leggyakrabban használtakat említeném meg, melyek az Unreal Engine, Unity, Godot Engine, GameMaker Studio, Phaser és Cocos2d-x. Az elöbb említettek különböző előnyekkel rendelkeznek. A felsoroltak

küzöl és egyben a legnépszerűbb fejlesztő környezet a Unity, ami híres a felhasználóbarát oldaláról és könyű használható. Majd nem összes létező platfomra lehet vele játékott készíten, mint például Windows-ra, Androidra, webre vagy akár Apple TV-re is. Kezdők számára a legjobb választás, nagyon sok forrás elérhető. A másik, szélleskörben elterjedt technológia

az a Unreal Engine, ami többségében 3D játákok készítésébenterjedt el, de készíthető vele 2D-s játék is. A legismertebb játékok, ami Unreal Engine-nel készültek az a Fortnite, Hogwarts Legacy és a We Happy Few nevezetű játékok. A következő Godot Engine, ami 2D

játék fejlesztésben a legnépszerűbb. Ez egy nyílt forrású játékfejlesztő motor, könyen használható gördülékenyen tanulható és széleskörben támogatott. A GameMaker Studio sok min-

denben hasonlít a előzőekben felsoroltakban, amiben kiemelném, hogy a felhasználók nem igényelnek programozási ismereteket, mivel egy saját nyelvet használ, a GameMaker Language (GML)-t, amely egyszerű és könnyen tanulható. Amikor 2D-s játék fejlesztésről van szó

nem hagyhatjuk szó nélkül a Cocos2d-x-et. Ez egy olyan eszköz amely különös előnyökkel jár a fejlesztők számára, mivel ingyenes és nyílt forrású kódú keretrendszer, aminek számos előnye van ammiért jó választás lehet. A Cocos2d-x c++ programozási nyelvet alapúl ami lehetővé teszi felhasználók számára a közvetlen hardveres hozzáférést. Ennek eredményeként kiváló teljesítményt nyújt, ami kulcsfontosságú a gyors és sima játékélmény elérésében. Az

én választott technológiám a Phaser ami kiemelkedő helyet foglal el, különösen a 2D játékok terén, amelyeket HTML5 és JavaScript technológiákkal hoz létre játékot. Ez a keretrendszer nem csupán egy eszköz, hanem egy teljes környezet, amely lehetővé teszi a fejlesztők szá-

mára a kreativitás kibontakoztatását és rugalmas játékok létrehozását. Ezeknek a fejlesztő

környezeteknek megannyi különböző előnye van, a felsoroltakon kívűl és számottevő hasonlóság van bennük. Az választási szempotnjaiban jó néhány tulajdonságot szem előtt tartottam. A phasernél találtam meg mind azt a tulajdonságot ami miatt, a legkézenfekvőbb választás ami a 2.1. táblázatnál látható.

2.1. táblázat: Szempontok összehasonlítása

•	Phaser	Unity	Unreal E.	GMS	Cocos2D
JavaScript	✓	✓	×	×	×
2D játék	✓	✓	×	✓	✓
HTML alapú	✓	×	×	✓×	✓×
ingyenes	✓	✓×	✓×	×	✓

2.2. Phaser bemutatása

A választott technológiám a fentiekben említett a Phaser lett. Mivel számomra megannyi szempotból megfelelt. Mivel olyan programozási nyelvet használ amit már egyetemen folytatott tanulmányaim során már tanultam.

- A Phaser egy HTML5 és JavaScript alapú keretrendszer, amely kifejezetten 2D játékok fejlesztésére lett tervezve. Továbbá egyesíti az egyszerűséget és a hatékonyságot, ennélfogva kiváló választás lehet kezdők és tapasztalt fejlesztők számára is. Ezt a játék fejlesztési környezetet széles körű támogatása miatt gyorsan elsajátítható a keretrendszerének használata.

- A dokumentáció részletes és jól strukturált, és a fejlesztők számára könnyűvé teszi a belépést a keretrendszer világába, amit a Phaser hivatalos oldalán szintén tapasztalhatunk. A szakdolgozatom elkészítése során többnyire a phaser weboldaláról tájékozóttam.

- Az alkalmazások készítése HTML5 és JavaScript felhasználásával lehetővé teszi a keresztplatformos kompatibilitást. Ez azt jelenti, hogy a Phaserrel készített játékok könnyen futtathatók böngészőkben, mobil eszközökön és más platformokon is.

- Ugyanakkor kiváló választás a prototípuskészítéshez. Az előre elkészített funkciók, például az animációk, a fizika és a hangkezelés könnyen integrálhatók, gyorsítva ezzel a fejlesztési folyamatot. Ezen felül még számos beépített funkcióval is rendelkezik, amelyek segítik a fejlesztőket a játékuk különböző részeinek kezelésében. Emellett a keretrendszer moduláris, így könnyen kiterjeszthető saját egyedi igényekhez.

- Mind emellett élénk fejlesztői közösséggel is rendelkezik, amely aktívan részt vesz a keretrendszer továbbfejlesztésében és a felhasználók támogatásában. Ez a közösség ösztönzi az ötletek megosztását, a problémamegoldást és a fejlesztés folyamatos fejlődését.

- A Phaser tehát nemcsak egy hatékony eszköz a 2D játékok fejlesztéséhez, hanem egy olyan közösségi keretrendszer is, amely támogatja a fejlesztőket és lehetővé teszi számukra, hogy kreatív elképzeléseiket valósággá formálják. A Phaser használatával a

játékfejlesztés egy midenki számára elérhető és élvezetes kihívás, amely mind az amatőrök, mind a szakértők számára megnyitja a játék fejlesztés világának kapujait.

- Az egyetemi tanulmányaink során számos hasznos készséget sajátítottunk el, köztük az alapvető HTML és JavaScript ismereteket, amelyekre példaként a webtechnológiák nevű tantárgyban összpontosítottunk.

2.3. Webes játékok előnyei

- Webes felületre játékokat fejleszteni számos előnnyel járhatnak, amelyek között szerepel a hozzáférhetőség, a könnyű elérhetőség, a platformfüggetlenség és a közösségi interakció.

- A webes játékokhoz való hozzáféréshez csak internetkapcsolatra és egy böngészőre van szükség. Ez lehetővé teszi, hogy a felhasználók bárhol és bármikor játszhassanak, ami növeli a felhasználói bázist.

- A szórakozni kívánó egyének bármilyen eszközről játszhatnak, amely támogatja a böngészőket, beleértve számítógépeket, táblagépeket és okostelefonokat is. Ezekhez nem szükséges speciális hardver vagy operációs rendszer.

- Továbba könnyen terjeszthetők és megoszthatók az interneten, ami lehetővé tette a széles körű elterjedését és a gyors népszerűség Ráadásul lehetőséget ad arra, hogy a játékotfejlesztők csökkentsék a fejlesztési költségeket és egyszerűbben kezeljék a szervereket, mivel a játékokat közvetlenül a böngészőből futnak.

3. fejezet

Tervezés

3.1. Játék bemutatása

A játék amit csináltam egy HTML5 2D oldalnézetes/féloldalnézetes játék, melynek az implementációját JavaScript programnyelven alapúl, Phaser könyvtárral kiegészülve. Egy több szintes platformer/kaland játék volt az alapkoncepció. A témáját a Narutó nevezetű japán animációs rajzfilm sorozat, a Super Mário nevezatű videó játék és a Doodle jump nevű mobiltelefonos játék ihlette ami a következő a 3.1 látható. A következőképpen lehet játszani főhősünket kell irányítva ki kell kerülnünk a repülő késeket,közben érméket kell gyüjtenünk. Minden 12 érme után plusz egy kés jelenik meg a platfomunkon ami a pálya elemeiről vissza pattan, ezzel megnehezítve az érmék összegyüjtését. A főhősünket a nyilakkal lehet irányitani, jobbra, ballra lehet vele menni és ugrani a fel fele nyillal. A játékhoz egy Menü felületet

3.1. ábra: Doodle Jump mobil telefonos játék.

szeretnék készíteni ahonnan el tudjuk indítani a játékot, távábbá elérni tudjuk az irányítás és a beállítások felületet. Az Irányítás felületen meg tudjuk nézni mivel lehet irányítani a főhősünket, majd onnan úgyszintán eltudjuk indítani a játékot és vissza is tudjunk lépni a Menübe. Az beállítások felületen a játék hangját tudjuk állítani egy csuszkával. Majd inne úgyszintén el tudjuk indítani a játékot. A játék felületen is gombok segítségével tudjunk állítani a hangerőt, azonfelül még meg is lehessen állítani és elindítani is a játékot. A vége lett a játéknak legyen lehetőség az újra indításra és, hogy valamilyen módon legyen jelezve, hogy a játék véget ért.

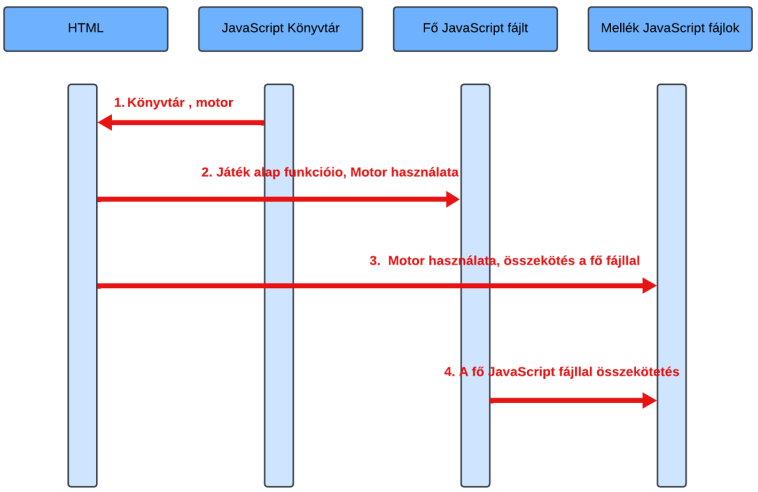
3.2. ábra: Az elképzelt játékmenű látható.

3.2. Előkészületek

3.2.1. A fejlesztő környezet kritériumai

A játék elkészítéséhez ki kellett választanunk a számomunkra legoptimálisabb Játékfejlesztő motort. Ehez több különbözőtt letöltöttem és kipróbáltam. Részemről ilyenkor az döntött melyiket lehet a legkönnyebben megtanulni, és olyan programozási nyelvet használjon amit már ezelőtt használtam. Továbbá, hogy elérhető legyen tömérdeknyi tanitó jellegű dokumentáció és hogy ingyenes legyen. Ha ezzel megvagyunk akkor kezdhetjünk megtervezni a programunk stuktúráját.

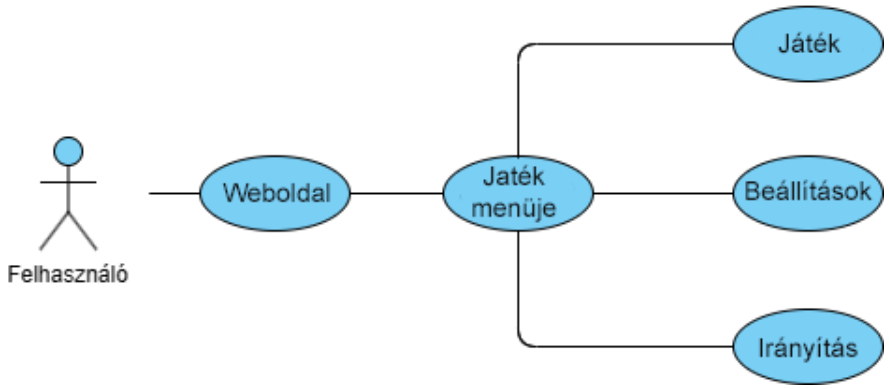
Amennyiben a Phaserrel szeretnénk 2D játékot készítenünk, akkor először egy index.html fájlt kell létre hoznunk, majd ezt kötjük össze phaser.js fájllal. Ezeltál tudjuk használni a phaser könyvtárát. Ezek után szüksége egy fő.js fájlra, ahol a játék alapját kreáljuk, mint például a gravitáció és az ablak mérete. Következő lépésként össze kell kötnünk a mellék.js fájlokkal amik egyenértékűek és több is lehet belőlük. A Phasernél fontos a fájlok helyes sorrendben való meghívása. Ezt a felépítést a következő 3.3 ábrán láthatjuk.



3.3. ábra: Program stuktúrája.

3.2.2. A programom felépítésének koncepciója

A felhasználó szempontjából a játék akkor tud elérhető lenni, ha a játékunkat feltöltjök egy az internetre egy weboldalra. Szerencserére phaser-rel készült játékok százai a facebookon, ami a világ egyik legismertebb platformja. Az 3.4 ábrán a menete látszódik hogy a felhasználó hogyan is tudja elérni a játékot. Legelőször meg kell jutni a weboldalra ahova fel lett töltve, onna elindul a játék. Majd a játék menülével találkozik, ahol gombokkal találkozik, ami segítségével el tudja érni a játék, a beállításokat és az irányítást ahol megtudja nézni mivel kell irányítani a játékot.



3.4. ábra: felhasználói diagram.

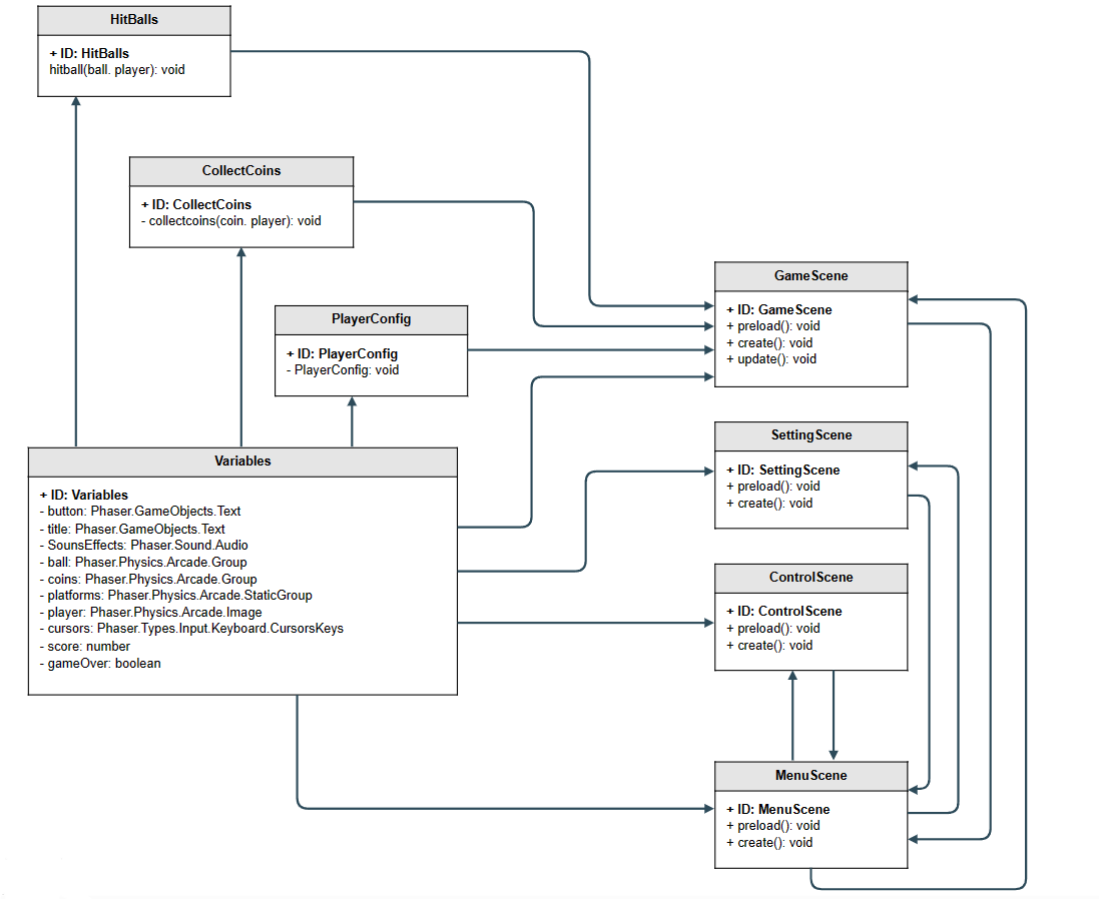
A következő **??** osztály diagramon a játék struktúrája látható, hogy a különböző osztályok miként fogják elérni egymást. Ezek azt osztályok mint kiterjesztett osztályok, tehát a `config.js` fájl és a `index.html` miatt eltudják érni egymást. Először is a variables osztállyal kezdeném ahol az osztályon kívűl deklaráltam számos változót, egyedet. Az osztályon belül meg egy konstruktor található, amivel az egyedim kulcsot fogjuk hozzáadni, hogy összeköttetésben legyen a többi osztállyal. Ehez hasonló a PlayerConfig, a Hitballs és a CollectCoins osztályok.

A **PlayerConfig** az osztályon kívül a **player** egyednek a fizikai tulajdonságai vannak beállítva, hogy milyen gyors legyen a karakter mozgása, ami playerconfig változó segítségével lett megvalósítva. A HitBalls osztályban az egy hitballs függvény található, ami a **balls** és a **player** egyednek ütközések észlelésének hatásáért felelős. A függvényben található egyedek miatt vannak összekötetésben a Variables osztállyal.

A **CollectCoin** osztályban is egy függvény található, ami a **coin** és a **player** egyednek egymás átfedéséért felelős. Ezekere az osztálykra nem feltétlenűl van szükség, a bennük lévő objektumok akkár a **MenuScene** osztályban is lehetnének. Én azért raktam ezeket külön osztályokban, hogy módosítások miatt gyorsan megtudjam őket találni. Ha az osztályokon belűl lennének deklarálva akkor csak az osztályon belül lennének elérhatőek ezek a globális változók és függvények. A **Variables** osztályban deklarált változók az összes osztály számára szükséges az elérésük a benne lévő változók miatt.

A **GameScene** osztály használja a **HitBalls**, **CollectCoins** és **PlayerConfig** osztálynál deklarált függvényeket, objektumokat, emiatt összekötetésben vannak. Továbbá ez a osztály összeköttetésben van a **SettingScene**, **MenuScene** nevű osztálkyokkal is mivel egy gomb segítségével összekötetésben leszek, mivel a gombbal tudunk majd az osztályokban létre jött helyszinek között váltani. Ezen felül még a Háttérzene miatt is kapcsolatban lesz mind a két osztállyal, mivel a MenuScene osztályban lesz elindítva a háttér zene a `play()` objektum által és ennek a hangerejét a **ControlScene** és a **GameScene** osztályban létre hozott csuszkával

lehet állítani. A **MenuScenen** osztály továbbá kapcsolatban van a **ControlScene** osztályal is.



3.5. ábra: Osztály diagram.

4. fejezet

Megvalósítás

4.1. Előkészületek

- Mivel a Phaser egy HTML5 játékkeretrendszer, ami lehetővé teszi egy böngészőben futó játék elkészítését. Ennek használatához számos előkészületre van szükség. Az alábbi lépések fognak nekünk segíteni.

- Első lépésként szükségünk lesz egy webszerver feltelepítése például a Node.js vagy használhatjuk a beépített webszervereket. Számomra a Node.js telepítése elkerülhetlen volt, mivel globális telepítésel tettem fel a phaser-t, a könnyebb használhatóság érdekében és a Node.js nélkül nem tudjuk ezt megcsinálni. A Node.js feltelepítésére előbb szükségünk van Node Package Manager telepítésére, ha nem lenne rajta számítógépünkön. A letöltése és telepítése a következő kódót kell beírnunk Windows Operációs rendszerünk parancssorába.

```
npm install -g npm
```

  Ha ezzel meg vagyunk akkor a kezdhetjük a Node.js telepítését ami úgy tudunk meg tenni, hogy letültjök a wwww.nodejs.org weboldalról, majd a letöltött dokumentumot teleptjük. A sikeres telepítést a Windows parancssorban a

```
node -v vagy npm -v
```

  lehet leellenőrizni, ami meg mutatja a feltelepített npm verzió számát.

- Második lépésként elkezdhetjük a Phaser Globális telepítését. Amire azért van szükség mivel A globális elepítés során a Phaser keretrendszer fájljai a rendszerünkre települnek, és elérhetők lesznek minden olyan projekt számára, amelyben a Phaser-t használjuk. Ez számos elönnyel jár.

  1. **Könnyebb projektmegosztás:** Ha másoknak is meg kell dolgozniuk a projektel, vagy ha egy másik gépen szeretnénk folytatni a fejlesztést, akkor nem kell minden alkalommal a phaser.min.js fájlt másolnunk vagy letöltenünk. A globális telepítés révén a Phaser minden projekt számára elérhetővé válik a rendszeren.
  2. **Frissítések kezelése:** A globális telepítés lehetővé teszi, hogy könnyen frissítsük a Phaser verzióját. Egyszerűen frissíthetjük a globális telepítést a legújabb verzióra, és minden projekt, amely a rendszeren belül hivatkozik a Phaser-re, automatikusan az új verzióval fogjuk használni.

3. **Rendszerintegráció:** A globális telepítés lehetővé teszi a rendszer integrációját, például a parancssorban való hívást vagy más automatizált folyamatokat.

A globális telepítést úgy tudjunk meg tenni, hogy a Terminálunkba bele írjuk a következő kódot.

```
npm install -g phaser
```

Majd létre kell hoznunk egy új munkakönyvtárat, ahol Phaser projektetket fogunk tárolni. Következőképpen el kell navigálnunk a munkakönyvtárba a paracssorral. Ezután be kell írnunk a

```
phaser init jatekunkneve
```

A jatekunk neve helyett tetszőleges nevet is írhatunk. Ha ezzel megvagyunk akkor parancssorral el kell mennünk a projektunk mappájába és beírni azt hogy

```
npm install
```

miután feltelepítődött, akkor a

```
npm start
```

kódot kell beírni. Ez elindítja a fejlesztői szervereket és a játék elérható lesz a böngészőnkben.

- Harmadik lépésként szükség lesz egy szöveg szerkeztő programhoz. Én Visual Studio Code-ot használtam mivel számos kiegészítő elérhető, ami megkönnyebbíti munkánkat.

- Negyedik lépésként el kell látogatnunk a phaser.io című weboldalára, ahol le kell tölteni `phaser.js` vagy a `phaser.min.js` fájlt. Mivel a globális telepítés önmagában nem elegendő, mert a phaser.js és phaser.min.js tartalmazza a keretrendszer magját.

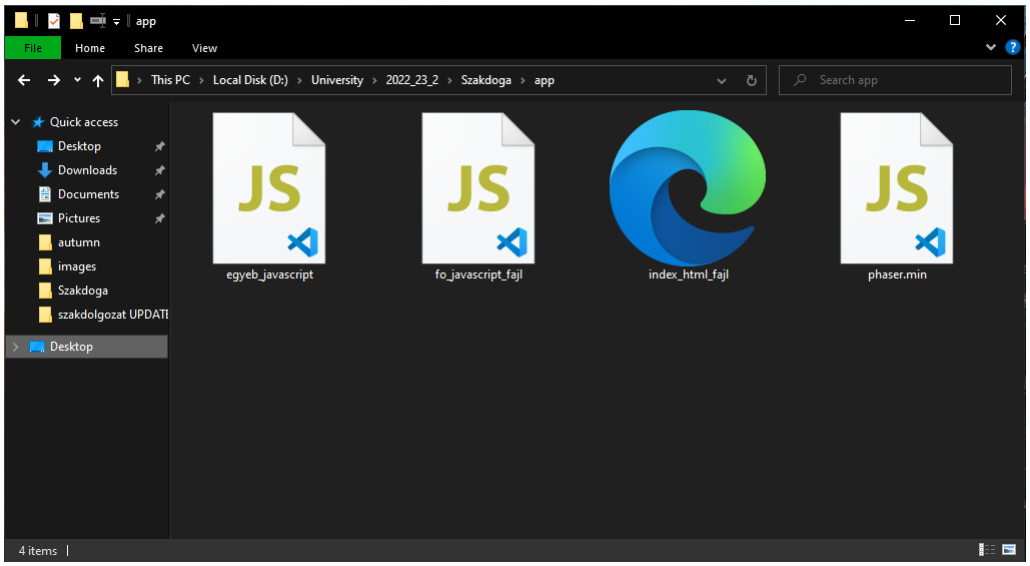
  A Weboldalon több különböző opció közül választhatunk, hogy melyiket szeretnénk letölteni. Le lehet tölteni phaser 2.0 verzióját és a phaser 3.7 és az egyel korábbi verziót a 3.6-t is. Én a zip-ben tömörített formáját töltöttem le, ezután a benne lévő phaser.min.js bele kell másolnunk a projektünk mappájába. Ez a fájl teszi lehetővé, hogy a Phaser-t tudjuk használni.

- Hatodik lépésként szükségünk lesz egy `index.html` nevű fájl létre hozására a projektünk mappájában.

- Hetedik lépésként létre kell hoznunk, a projektünk mappájában egy fő JavaScript fájlt, ami az én projektemben a config.js. A fő javascript fájlunk mellé létrehozhatunk több másik JavaScript fájlt is.

- Nyolcadik lépésként ezeket a fájlokat valamilyen módon össze kell kötni. Erre több különböző módszer létezik, amit a fejlesztés során tapasztaltam is.

- Ha mind ezt sikerült megcsinálnunk, akkor a mappabán a következős fájlokat kellene látnunk, `index.html`, `phaser.min.js`. és egy vagy több JavaScript fájlt, amit a 4.1 szemléltet.



4.1. ábra: Projekt felépítése.

4.2. Munka környezet összerakása.

Először is a fájlok összekötésének szeretném bemutatni, mivel e-nélkül programmunk használhatatlan. Az `index.html` fájltban hozzá kell adnunk a phaser.min.js fájlt amit a 4.1 programkódon láthatunk.

```
<!DOCTYPE html>
<html lang="hu">
<head>
    <script src="phaser.min.js"></script>
</head>
<body>
    <script src="GameScene.js"></script>
  <script src="MenuScene.js"></script>
    <script src="Config.js"></script>
</body>
</html>
```

Programkód 4.1. index.html fájl

Amelyet a html-en bellül a head-ben vagy a body-ban kell elhelyeznünk. Fontos, hogy felülről lefelé haladva a phaser.min.js legyen az első és csak utána jöjjön a többi fájl, ugyanis, ha felette lennének azok a fájlok, amik a phaser könyvtárát használnák, akkor nem tudnák elérni a keretrendszer magját. A html fájl-t sikeresen összekapcsoltuk a phaser.min.js fájllal, meg a config.js fájllal, de a többi JavaScriptfájllal még nem teljes a kapcsolat, ugyanis még össze kell kötnünk ezeket a config.js nevű fájllal, ahol játékkonfiguráció objektum valósul meg, mivel index.html fájl csak betöltésért felel. Amit a 4.2 programkódon látható kóddal.

Ezeknél a JavaScrit fájloknál a jobból ballra menő sorrendiség számit. Felmerűlhet az a kérdés, hogy miért nem elég HTML fájlban össze kapcsolni. Azért nem mert a HTML fájlnak nincsen információja a játék logikájáról vagy, hogy a JavaScript fájlok hogy kapcsolódnak egymáshoz. Ezzel ellenszemben a config.js fájlnak van hiszen itt található játékkonfigurációs objektum az, ahol megadhatjuk a játék beállításait, és a játék részét képező további fájlokat. Amikor fájlokat adunk hozzá a main.js fájlhoz, azt mondja a Phasernek, hogy ezek azok a fájlok, amelyek a játék részét képezik, és a Phasernek kell kezelnie őket. A config.js fájlban

valósúl meg a játékkonfiguráció objektum, ami 4.2 programkódon látható.

```
var config = {
    type: Phaser.AUTO,
    width: 800,
    height: 600,
    physics: {
        default: 'arcade',
        arcade: {
            gravity: { y: 350 },
            debug: false
        }
    },
    scene: [MenuScene, GameScene]
};
var game = new Phaser.Game(config);
```

Programkód 4.2. fájlok összekapcsolása JavaScripttel

A fájlok összekapcsolása még nem ért véget, hiszen még az összekötendő kódban is jelölnünk kell, hogy tudjunk rá hivatkozni, más fájlban is. a 4.3 programkódnál látható, hogy a MenuScenek egy kiterjesztett osztálynak kell lennie, hogy összekötettésben legyen más fájlokkal, ahoz, hogy hivatkozni is tudjunk rá. egy construktor segítségvel egy szuper kulcsot kell írnunk a kiterjesztett osztályon belül. Az osztályon kívűl globális változókat deklarálhatunk. A változók egy részét egy variable.js fájlban deklaráltam.

```
class MenuScene extends Phaser.Scene {
    constructor() {
        super({ key: 'MenuScene' });
    }
}
```

Programkód 4.3. kiterjesztett osztály és Szuper kulcs használata.

4.3. Játék Felépítése

Minden Scene ugyanazon a logikán épül fel. Egy scene-nek tartalmaznia kell egy `preload`, `create` és egy `update` függvényt.

```
class MenuScene extends Phaser.Scene {
    preload() {

    }
  create() {

    }
  update() {

    }
}
```

Programkód 4.4. Scene-k függvényei.

A `preload` függvény felel a látvány elemek hozzáadásáért és betöltésért felelős. Itt tudunk hozzáadni spritesheet-et, képeket és hang fájlokat. A kód tartalmazza a fájlok származási utját

és még egy egyedi azonosítóval is el vannak látva, amivel tudunk rájuk hivatkozni. A sprite sheet-eknél még pluszban meg kell adnunk a frame hosszát és szélességét is.

```
class MenuScene extends Phaser.Scene {
    preload() {
        this.load.image('bgMenu', 'assets/uchiha_d.jfif');
        this.load.audio('music', ['assets/backgroundmusic.mp3']);
    this.load.spritesheet('dude', 'assets/sasuke.png', { frameWidth: 32,
    frameHeight: 48 });
    }
}
```

Programkód 4.5. preload függvény.

A `create` függvényben tudjuk hozzá adni a `preload` függvényben hozzáadott hang és kép fájlokat megjeleníteni, elindítani, elhelyezni, formázni és objektumokra hivatkozni vagy akár deklarálni, létre hozni. A 4.6 progam kódba látható pár példa, hogy mennyi minden lehet kreálni a `create` függvényben

```
class MenuScene extends Phaser.Scene {
    create() {
    bgMusic.play();
    const rectangleBG = this.add.rectangle(400,300, 800,600, 0x00000,
    0.20);
    this.add.image(0,0,'bgMenu').setOrigin(0.075, 0).setScale(1.55,1.8);
    this.physics.add.collider(player, platforms);
    this.physics.add.overlap(player, coins, collectcoin, null, this);
    }
}
```

Programkód 4.6. create függvény.

Az `update()` függvény a Phaser keretrendszer egy fontos része, és erre azért van szükség, mert a játék logikáját, állapotait és megjelenését frissíti minden egyes képkockában. A keretrendszer működése a következőképpen épül fel.

1. Eseménykezelés: A különböző események, például billentyűleütések vagy egér mozdulatainak, eseményére reagálva a játék változásokat hajt végre.

2. Update ciklus: A játék minden egyes képkockában frissül, és az `update()` függvény hívódik meg. Ez lehetőséget ad arra, hogy módosítsuk a játék állapotát, ellenőrizze a kollíziókat, vezérelje a karakterek mozgását és egyéb dinamikus elemeket.

3. Renderelés: Miután az `update()` függvény lefutott, a játék kirajzolja az aktuális állapotát a képernyőre.

4. Időzített események: Időzített eseményeket, például pontok vagy erőforrások regenerálódását is kezelheti.

```
class MenuScene extends Phaser.Scene {
    update() {
        if (gameOver)
        {
            bgMusic.stop();
            gameOverTxt.visible = true;
            restTxt.visible = true;
```

```
8               restButton.visible = true;
9               return;
10          }
11      }
12  }
```

Programkód 4.7. update függvény.

4.4. JavaScript fájlok

4.4.1. Config.js

Először is a `config.js` kezdeném, ahol az játék konfigurációjának objuktuma valósúl meg, amit már korábban is említettem, ez azt jelenti, hogy játék ablakának méretét, fizikáját és típusát tudjuk meghatározni. és további fájlokat lehet hozzá adni, ami a 4.2 programkódon látható. A játék fizikájának tulajdonságát `arcade`,-ra van állítva. A phaserben 4 fizikai rendszer érhető el:

- Az arcade physics egy egyszerű és könnyen használható fizikai rendszer, amely tökéletesen alkalmas 2D játékokhoz. Támogatja az ütközésdetektálást, a gravitációt és más alapvető fizikai jelenségeket1. Azonban nem támogatja a komplexebb jelenségeket, mint például a forgatónyomaték vagy a rugalmas ütközések.
- P2 ami egy komplex fizikai redszer ami támogatja a forgató nyomatékot és a rugalmas ütközéseket.
- Ninja Physics ez egy speciális fizikai rendszer amelyet, elsősorban platformjátékokhoz találtak ki.
- Box2D ez egy kereskedelmi ez egy kereskedelmi bővítmény egy eléggé erős fizikai motorral rendelkezik.

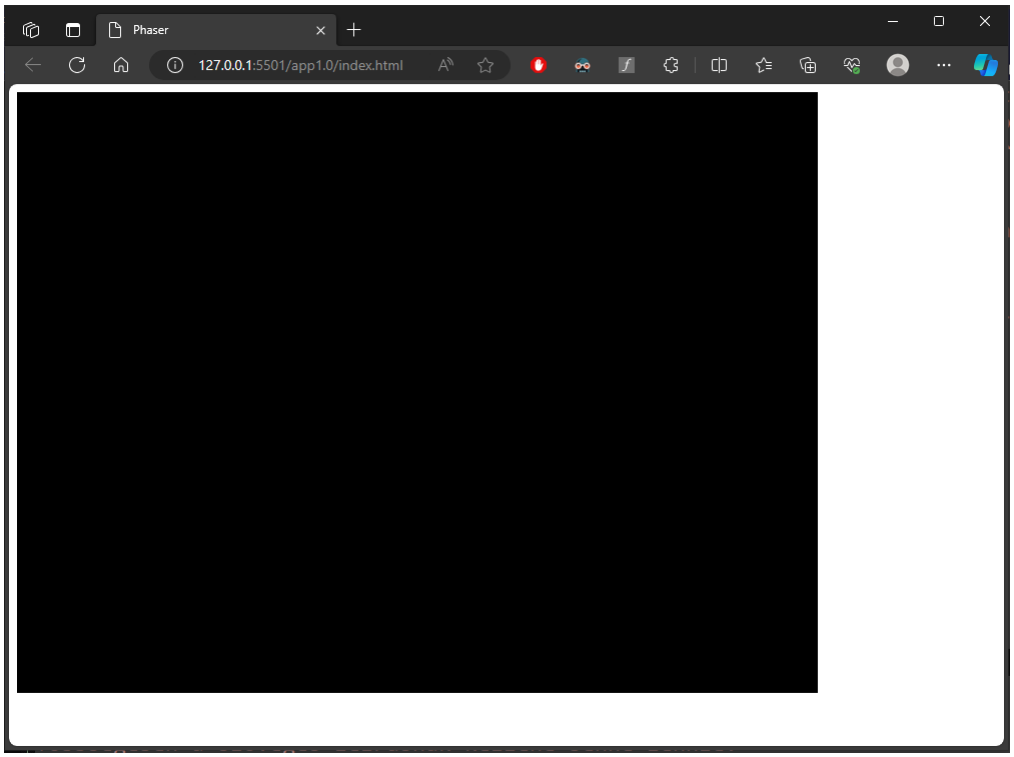
Egy játékobjektum (például egy Sprite) csak egy fizikai rendszerhez tartozhat, de egy játékban több rendszert is aktív használhatunk. Én az számomra a felsoroltak közül az arcade volt a megfelelő.

Ha mindezt beállítottuk akkor a következőt kellene látnunk, amit a 4.2 ábrán látható.

4.4.2. Variables.js és a PlayerConfig.js

Ebben az alfejezetben kettő fájl fogok bemutatni a `variables.js` és `PlayerConfig.js`, mivel ugyan azt a funkciót látják, csak én az átláthatóság miatt külön választottam.

A `variables.js` fájlban a szükséges változókat deklaráltam, amit az osztályon kívűl hoztam létre, mivel így látható lesz a többi osztály számára. Az osztál is egy kiterjeszett osztáály és a konstruktorral van összekötve. A változók típusa `var` és `let`. A két deklarálási típus között az a különbség, hogy a `var` az függvény hatókörű míg a `let` az pedig blokk hatókörül. A függvény hatókörü azt jelent, hogy a változó a függvényen belűl érhető míg ellenben a blokk hatókörü csak a blokkban érhető el, amiben deklarálva lett. A `var` paraméterrel deklarált változók hatókörük tetejére kerülnek, és undefined értékkel inicializálódnak, ellenben `let`-vel deklarált változók szintén a blokk tetejére kerülnek, de nincsenek inicializálva. Ha a deklaráció előtt `let` változót próbálunk használni, akkor hivatkozási hibát fogunk kapni.



4.2. ábra: config.js

Ugyan ebben a hatókörben a `var` lehetővé teszi a változók ujboli deklarálását. Ugyan ezt a `let` nem teszi meg.

A `score` nevű változó 0-ra van állítva, hiszen ez fogja a számolni nekünk a pontokat a játékban. A `gameOver` változó, azért felel, amikor meghallunk a játékban egy szöveg és egy gomb jelenik meg, továbba játék zenéje is megáll. Ez a változó felel azért, hogy csak akkor látszódjanak a hozzá rendelt értékek amikor kell. Emiatt az értéke `false`. Ami a 4.8 programkódban is látható.

```
var score = 0;
var gameOver = false;

class Variables extends Phaser.Scene{
    constructor(){
        super({key: 'Variables'})
    }
}
```

Programkód 4.8. variables.js

A `variables.js` nevű fájlban a többi változó a gombokért, sprite-sheet-ekért és hangokért felelős.

A `PlayerConfig.js` fájlban egyszerűen csak a irányított karakterünknek a tuladonságait lehet beállítani, olyanokat mint a mozgási és ugrási sebesség, ami a 4.10programkódon látható

```
var player_config = {
    player_speed: 500,
    player_jumpspeed: -357,
};
class PlayerConfig extends Phaser.Scene{
```

```
    constructor(){
        super({key: 'PlayerConfig'})
    }
}
```

Programkód 4.9. PlayerConfig.js

4.4.3. HitBall.js

Ebben a fájlban egy ütközés kezelő függvényt deklaráltam, ez azt jelenki amikor az irányított karakterünket el találja egy kés akkor vége lesz a játéknak. Továbbá ebben a függvényben még a játék befejezéséhez effektjei is deklarálva vannak, amik a következők.

- Ez a hangot fájl kezdi lejátszani. Az `erroSound` az egy általam létrehozott változó ami a `variables.js` fájlban van létre hozva.

```
this.errorSound.play();
```

- A következő kód szünetelteti a fizikai rendszert, amire azért van szükség, hogy amikor az ütközés létre jön kifaggyona kép ezzel jelezve, hogy a játéknak vége van.

```
this.physics.pause();
```

- Ez a kód vörösre változtatja az irányított játékosunk sprite-jának színét. Ezzel is jelezve hogy játék végét.

```
player.setTint(0xff0000);
```

- Ez a `turn` kulcsszó megállítja az sprite-sheet animációját.

```
player.anims.play('turn');
```

- Igazra állítja a gameOver változót, ezzel megjeleníte a gameOver változóhoz csatolt, szöveget és gombot.

```
gameOver = true;
```

  A függgényben a player és ball objektumokra van hivatkozva, mivel ha ez a két objektum egymáshoz ér, akkor a függvényben definiáltak, fognak létre jönni.

4.4.4. CollectCoin.js

Ez a függvény a játékban az érmék gyűjtésekor felelős, amikor a felhasználó a karakterrel hozzáér egy érméhez. Az érme eltűnik, a pontszám nő, és ha nincsen több érme a pályán, akkor új érmék jellennek és plusz egy kés add hozzá. Ahányszor fogytak el a pályáról az érmék, azzal az értékkel meg egyező kés van a pályán. Az érméket és a késeket a játékmezőn belül véltlen szerű helyekre helyezi el a platformon. Ebben a függvényben a `player` és a `coin` objektumokra kell hivatkozni.

- Aminkor felszedjük a karakterünkkel az érmét a következő kód felel az eltünésért. A kódban található `true` paraméterek azt mondják a rendszernek, hogy teljesen tiltsa le az érme testét, mind fizikai,mind vizuális szempontból. Ha csak az első `true` lenne használva, akkor a fizikai szimulációban részt venne, de láthatatlan lenne, és ha csak a második `true` lenne használva, akkor látható lenne, de nem venne részt a fizikai szimulációban, tehát mind a kettőre szükségünk van.

```
coin.disableBody(true, true);
```

- A score változó ebben az függvényben növeli értékét, amit korábban 0-ra állítottunk.

```
score += 1;
```

- Ellenőrzi, hogy nincs-e több aktív érme a coins csoportban. Ha az aktív érmék száma 0, akkor a feltétel igaz lesz, és belép a blokkba.

```
if (coins.countActive(true) === 0)
```

- Egy iteráció a `coins` csoport összes gyermekén (azaz érméjén). Az iterált függvényben minden érmet újra engedélyeznek (enableBody(true)), és a jelenlegi adott pozícióját megtartva elhelyezi őket az ablak tetején (`child.x, 0`). Az érmék így, úgymond újraindulnak a pályán.

```
coins.children.iterate(function (child)
{
  child.enableBody(true, child.x, 0, true, true);
});
```

- Kiszámolja az újonnan generált labda `X` pozícióját. A karakterünk aktuális `X` pozíciójától függően az `X` értéket véletlenszerűen választja ki egy alsó és egy felső érték között.

```
var x = (player.x < 400) ? Phaser.Math.Between(400, 800) : Phaser.
    Math.Between(0, 400);
```

- Ez létrehoz egy új labdát a kiszámolt adott pozícióban, 16 pixel magasságban a játékmezőn.

```
var ball = balls.create(x, 16, 'ball');
```

- A következő beállítja a labda visszapattanási értékét. Az 1 azt jelenti, hogy a labda teljesen visszapattan, azaz nem veszít sebességét.

```
ball.setBounce(1);
```

- Itt beállítjuk, hogy a labda ütközzön a platformunk határaival.

```
ball.setCollideWorldBounds(true);
```

- Ezzel a kóddal beállítja a labda kezdő sebességét egy véletlenszerű sebességi értékkel vízszintesen (-200 és 200 között) és függőlegesen (20).

```
ball.setVelocity(Phaser.Math.Between(-200, 200), 20);
```

- Az utolsóval meg kikapcsoljuk a labda gravitációs hatását, tehát a labda nem fog lefelé esni.

```
ball.allowGravity = false;
```

4.4.5. MenuScene.js

A `MenuScene.js`-ben a játékom menüje lett elkészítve, ez a Fő scene amit, látunk akkor, amikor elindítjuk a játékot. A Menüben gomb objkektumok segítségevel a `GameScene.js`, `ControlScene.js` és a `SettingScene.js` fájlokat tudjuk elérni. Erről a felületről indul a játék háttérzenéje is. A `MenuScene.js`-ben a gombok megvalósítását és az effekteket fogom részletesen bemutatni, hogy milyen módon müködnek és benne deklarált változóknak milyen szerepe van. Ebben a scene-ben többségében a `create` függvény fogjuk használni. A `preload` függvényben már a nem rég említett eljárással, adtuk hozzá a szükséges képek és hang fájlokat. Az `update` függvényt ennél a scene-nél nem fugjuk használni.

- Először is a háttérrel kezdeni, hogy hogyan valósúl meg, további milyen módon alakítottam át az eredeti képet. A következő kód létrehoza kép objektumot ami a játék háttere lesz. A két darabb `0` az X és Y pozícióját jelenti, ami a kép bal felső sarkát helyezi az ablak bal felső sarkához. A harmadik paraméter, `bgMenu`, a kép egyedi azonosítója, amit preload függvényben rendeltünk hozzá, amelyen keresztül a programunk hozzáfér a képhez.

```
this.add.image(0, 0, 'bgMenu')
```

  Az előző kódunkhoz hozzá fűzve a következővel a kép origóját lehet beállítani. Az origó az a pont, amely körül a kép forgatható vagy transzformálható. Az `(0.075, 0)` azt jelenti,hogy az origó 7.5%-kal jobbra és 0%-kal lefelé helyezkedik el a kép bal felső sarkához képest. Ez a beállítás lehetővé teszi az eredeti kép eltolását vagy forgatását az origó körül.

```
.setOrigin(0.075, 0):
```

  Az ezután pluszban hozzá csatolt kód kép méretét állítja be. Az első paraméter, `1.55`, a vízszintes skálázást jelenti, míg a második paraméter, `1.8`, a függőleges skálázást jelenti. Tehát ez a kód megnöveli a kép méretét mind vízszintesen, mind függőlegesen az előbb említett értékekkel.

```
.setScale(1.55, 1.8):
```

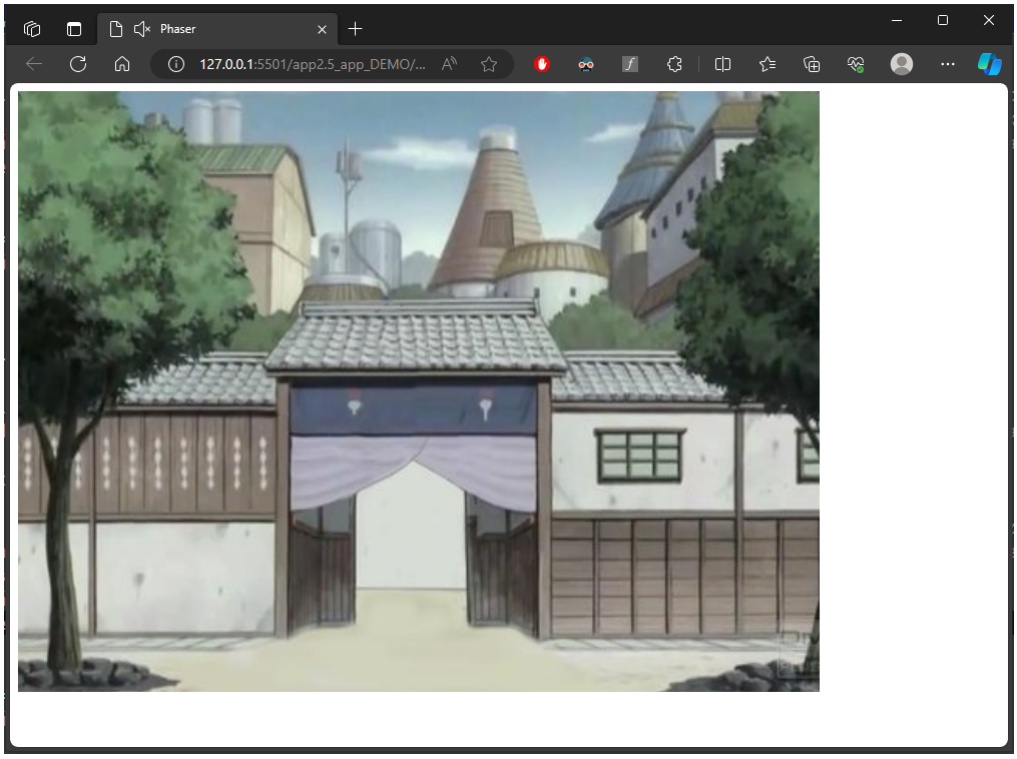
  Ezzekre a módosításokra azzért volt szükség hogy tökéletesen paszoljon, háttérkép az játékunk ablakához.

- A következő kód a játékunk szempontjából elhanyagolható, én csak azért írtam bele, mivel túl világos volt a háttérképnek választott képem, és a következő sor kóddal besötítettem enyhén, hogy a későbbiekben jobban látszódjanak az említett gombok. A kód egy átlátszó fekete téglalap alakú sprite-ot, hoz létre a `rectangle` a szó az objektum

formáját határozza meg. A `400,300` a téglalap központjának pozicióját adja meg. A `800,600` a méretét. A `0x00000` kód a szinét. A `0.20` az átlátszóság százalékos mértéket határozza meg. Fontos tudni, hogy ennek a kódnak csak akkor van értelmi, ha háttérképünket deklarált kód alatt helyezkedik el.

```
1  const rectangleBG = this.add.rectangle(400,300, 800,600, 0x00000,
       0.20);
```

Ezzel létre hoztuk a játékunk menüjének háttérképet, amit a **??** ábrán láthatunk.



4.3. ábra: A játék menüjének háttérképe.

Mivel ez még csak a játék menüje valahogy el kellene érnünk a többi scene-t is amit fizikális vagy virtuális gombok segítségével érhetünk el. Én az utóbbi lehetőséggel oldottam meg.

```
1          const startBT = this.add.rectangle(400,300, 200,50,0xff00ff,
   0.30);
2          startBT.setInteractive();
3          startBT.on('pointerdown', function(){
4              this.scene.start('GameScene');
5              this.sound.add('click').play();
6          },this);
7
8          button =this.add.text(400, 300, 'PLAY',{
9              fontSize: '32px Arial',
10             fill: '#ffffff'
11         });
12         button.setShadow(2,2, '#333333', 2, false, true);
13         button.setOrigin(0.5);
14         button.setTint(0xff00ff, 0xffff00, 0x0000ff, 0xff0000);
```

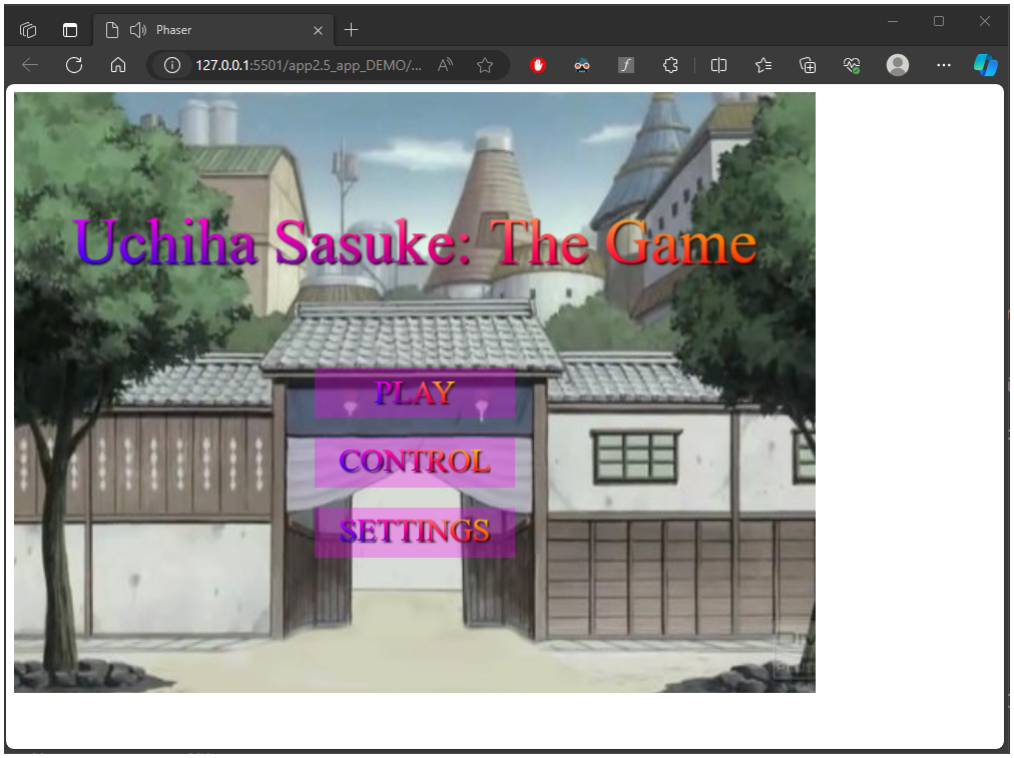
A gombnak egy konstans téglalap objektum van generálva, ami a `.setInteractive()` kóddal interaktívvá tettük a téglalapot, így reagálhat a felhasználói interakcióra, de ettől még nem lett még gomb. Miután hozzá adtuk az eseménykezelőt az objektumhoz, a téglalapra katintva le fut az eseménykezelő. Ebben az esetben elindítja a 'GameScene.js' fájlt és lejátszik egy hang fájlt

A `button` objektummal a gomb felíratát, helyét határoztuk meg. A `fontSize` kulcsszóval a betű méretét, és típusát lehet meg határozni. A `fill` kulszóval pedig a Betűnek a színét. Ezzel a gomb lényeges részét meghatároztuk a továbbiak csak díszítő tulajdonságok. `setShadow()` kód árnyékot ad a szövegnek a megadott paraméterekkel.

- Az első kettő paraméter az árnyék vetületét lehet vele beállítani horizontálisan és hertikálisan, hogy mekkora mértékkel távolodjon el.
- A harmadik paraméter a szinét. A negyedik azt állítja be, hogy mennyire legyen homályos.
- Az ötödik paraméter az árnyék kitöltésért felelős, a `false` azt jelenti, hogy nincs kitöltve.
- A hatodik pedig azért felelős, hogy körvonalként jelenítse meg az árnyékot vagy sem. Itt `true`-ra van állítva ami azt jelenti, hogy körvonalként kerül fel az árnyék.

`setOrigin(0.5)` középre állítja szöveget. `setTint()` függvénnyel a tónust lehet adni a szövegnek. A hozzá tartozó paraméterek a színezésért felelősek ami az objektum különböző sarkaira vonatkoznak az alábbi sorrendben bal felső, jobb felső, bal alsó, jobb alsó. Ezzel a technikával nagyon szép színátmenetes szöveget vagy akár formát lehet szinezni.

A 4.4 ábrán látható, a gombokkal ellátott menű.



4.4. ábra: A játék menüje gombokkal.

A menühöz még létre létre hoztam egy `title` és egy `bgMusic` objektumot. A `title` egy szöveg objektum, amivel a játék címét írtam ki, ugyan azzal az eljárással mint a gombok szövegeit. Az utóbbi a háttér zenéért felelős.

```
        title = this.add.text(400,150, 'Uchiha Sasuke: The Game', {
            fontSize: '64px Arial',
        });
        title.setTint(0xff00ff, 0xffff00, 0x0000ff, 0xff0000)
        title.setOrigin(0.5);

    //Background Music
        title.setShadow(2,2, '#333333', 2, false, true);
    bgMusic = this.sound.add('music',{
            loop: true
        });
        bgMusic.play();
```

Programkód 4.10. MenuScene.js háttérzene és cím objektum

A háttér zene objektm a `loop` szóval be lehet állítani, hogy a zeneszám, ha végetért, akkor induljon újra. Ebben az esetben `true`-ra van állítva ami azt jelent folyamatos ismétlődni, fog a háttérzenne. A `.play()` függvénnyel indítjuk el a zenét.

4.4.6. ControlScene.js

Az irányítás scene-ben az irányítás tudjuk megnézni, hogy, melyik gombbal tudjuk irányítani a karakterünket. Továbbá 2 darab gomb is található egy `Play` gomb és egy `Back`. Az utóbbival vissza tudunk menni a menűbe. Az előbbivel a játékban tudunk egy ugrani. A sceneben lévő szöveg ugyan azzal az eljárással készült mint a Menűben a felirat. A vissza gomb szintén ugyan úgy oldottam meg mint a Menűben lévő gombokat, csak itt a `MenuScene`-re lép és pluszban, hozzá van adva egy `.stop()`, ami leállítja a háttérzenét, ezt a függvényt azért raktam bele, mivel amikor vissza léptem a menüben, akkor elindította megégyszer a háttérzenét. Hiszen abban a scene-ben van a `.play()` függvény, emaitt ahányszor, váltogadtam a scene-k között mindig elindult, és nem állt le, de ezt a poblémát ki lehet ezzel küszöbölni. A `ControlScene` a következő 5.2 ábrán látható.

4.4.7. SettingScene.js

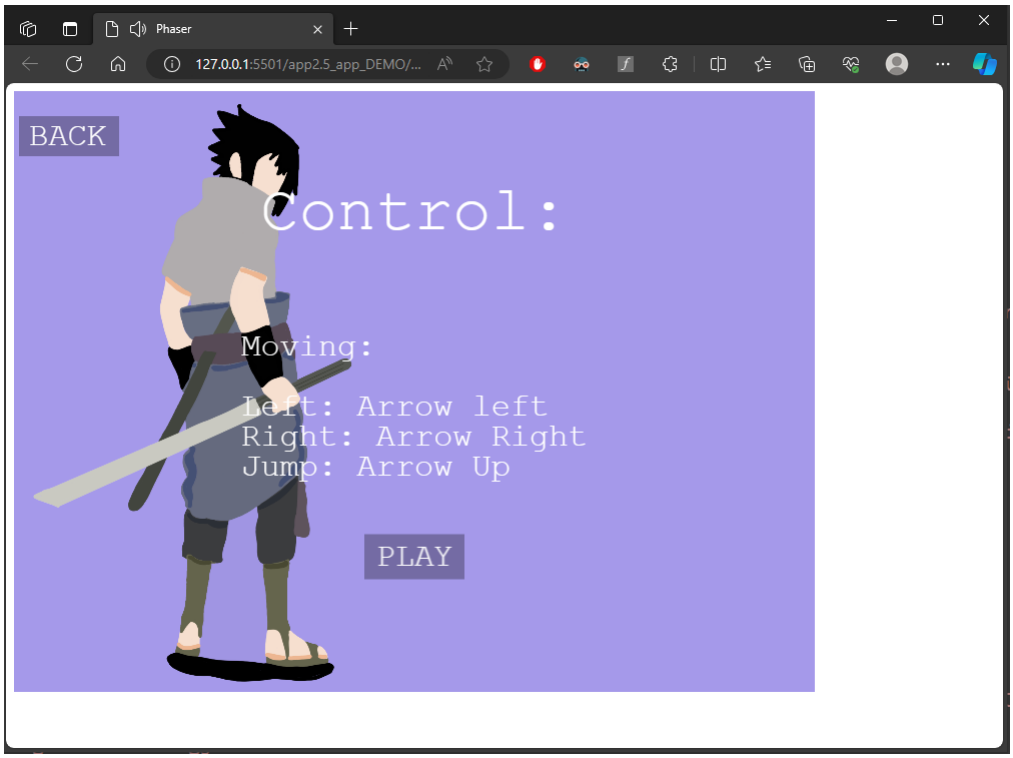
Ebben a menüpontban a háttérzene hangerejét lehet állítani egy csuszka segítségével és még egy vissza gomb és egy játékot elindító gomb található benne, amit a következőképpen valósítottam meg. A gombokat már előzőekben bemutattam, úgy, hogy arra nem fogok kitérni. Ez az fejezet a hang csuszka megvalósítását fogja részletezni.

```
let bar2 = this.add.rectangle(400, 300, 215, 30, 0xffffff, 0.5);
```

Ez egy szimple díszítő téglalap, amely csak dekorációként szolgál, és ez fogja vizuálisan meghatározni a csúszka területét. A téglalap 215 pixel széles, 30 pixel magas, fehér szinű (0xffffff) és az átlátszóság 50%-ra van állítva.

```
const slider = this.add.container(400, 300);
```

Ez egy konténer objektum, amely a csúszka elemeit tartalmazza. A konténer középpontja a képernyő közepénre van állítva.



4.5. ábra: ControlScene.js

```
let bar = this.add.rectangle(0, 0, 200, 16, 0x000000);
```

Ez a csúszka háttere.

```
const control = this.add.circle(0, 0, 12, 0x809779);
```

Ez a csúszka vezérlőpontja, amelyet húzva lehet módosítani a csúszka értékét. A vezérlőpont egy 12 pixel átmérőjű kör, és a színe halvány zöld (0x809779)).

```
control.setInteractive({ draggable: true });
```

Beállítja a vezérlőpontot interaktívvá, és engedélyezi, hogy húzható legyen körünk.

```
        control.on('drag', function (pointer, dragX, dragY) {

            control.x = Phaser.Math.Clamp(dragX, -100, 100);
            var volume = (control.x + 100) /200;
            bgMusic.setVolume(volume);
            // console.log(dragX);
        });
```

Eseménykezelő, amely akkor fut le, amikor a vezérlőpontot húzzák (drag esemény). A függvény a húzási eseményt kezeli, korlátozza az elmozdulást a csúszka területén belül, kiszámolja az aktuális húzás pozíciójából származó hangerőt, majd beállítja a háttérzene hangerősségét az aktuális pozíció alapján.

```
slider.add([bar, control])
```

Hozzáadja a csúszka elemeit a konténerhez.

4.4.8. GameScene.js

Ennél a fájlnál magát a játéknak a szintérét valósítottam meg. Itt valósúl meg a játék mechanikája. Itt is számos gomb található. Többségében ugyan azok mint ez előbb említett sceneekben is fellelhetők, ezért csak a különbségeket fogom részletezni. A `GameScene.js` fájlban deklarálva lett 4 darab gomb és egy hang szabályzó csúszka. Ez mind a képernyő alján látható. A ball felső sarokban egy pont számláló észlelhető. A következő gombok látszódnak, ballról jobbra haladva fogom fel sorolni őket, egy `back` (vissza a menüben), `setting` (SettingScene scene-re ugrik), `pause` (megállítja scene animációit) és `resume` (ami a megfagyasztott scene-t elindítja) gombok. Mindemellett található még egy rejtett gomb ami akkor jelenik meg amikor a karakterünk meghal. Ez a gomb a játék újra indításáért felelős amit `restart` gombnak neveztem el.

A `restart` gombot mutatnám be, ami egy reload gomb, ami a gyakorlatilag újra tölti a weboldalt. Ez **??** programkódon látható.

- A `window` egy globális objektum, amely az aktuális ablakot reprezentálja a böngészőben.
- `location` a window objektum egy tulajdonsága, ami a jelenlegi oldal URL-jét tartalmazza.
- A `reload()` pedig egy metódus ami az oldal újra töltéséért felelős, ami az alábbi program kódon látható

```
  restButton = this.add.rectangle(400, 370, 100, 30, 0x000000, 0.5);
    restButton.setInteractive();
  restButton.on('pointerdown',function(){
  window.location.reload();
  },this);
  restButton.setVisible(false);
    restTxt = this.add.text(400, 360, 'restart',{
    fontSize: '20px',
    fill: '#ffffff'
});
```

Programkód 4.11. Restart gomb

A pause gomb az alábbi módon lett elkészítve.

```
const pauseBT = this.add.rectangle(215, 580, 70, 20, 0x000000,0.50);
   button =this.add.text(190, 572, 'Pause',{
       fontSize: '16px',
       fill: '#ffffff'
   });
   button.setInteractive();
   button.on('pointerdown', function(){
       game.loop.sleep();
       this.sound.add('click').play();
   },this)
```

Ennél a kódnál az előzőekben használt dizájn elemek láthatók, az, ami a különbség, hogy nem egy scene-nel van össze kötve, hanem egy függvény van hozzá rendelve. Amikor a játék fut, a játékmotor ciklikusan futtatja a játék folyamatát egy `loop` segítségével. Ez a hívás azt jelenti, hogy a játékmotornak megtiltjuk a loop futtatását egy időre, vagyis alvásra küldjük

a sleep függvénnyel. Ennek a hatására egy úgy mond megfagyott képernyőt látható. A következő gomb aminek a működését bemutatnám az a resume gomb amivel, a megfagyasztott képernyőt elindítani, ami hasonlóan működik mint a pause gomb csak az ellentetjét valósítja meg a texttt.wake() függvénnyel a .sleep() függvény által megállított scene-t indítja el.

```
    const resumeBT = this.add.rectangle(300, 580, 80, 20, 0x000000,0.50);
    button =this.add.text(272, 572, 'Resume',{
        fontSize: '16px',
        fill: '#ffffff'
    });
    button.setInteractive();
    button.on('pointerdown', function(){
    game.loop.wake();
    this.sound.add('click').play();
    },this);
```

Most, hogy a gombokkal meg vagyunk áttérek a játék mechanikájára, ahol egy platformon egy karakter irányítunk. amivel a érméket kell gyüjtenünk a platformon. Majd megjelennek a késes amihez mi nem érhetünk, ha hozzá érünk a késekhez, akkor a játék véget ér. Ezeknek az elemeknek a ezeknek az elemek kölcsönös kapcsolatáról lesz szó.

Először is a platform elkészítésével kezdeném, ami nagyon fontos a játék készítése sorám, hiszen, ha nincs platform akkor nem tudjuk leellenőrizni, hogy jól működik-e gravitáció. A platform elkészítését a preload() függvényben valósítjuk meg az 4.12 programkódon látható. A platforms egyed variables.js fájlban lett deklarálva.

Ezzel az egyeddel egy statikus csoportot hoztunk létre a .staticGroup(); metódus segítségével. Ezek olyan csoportok, amelyek a játék világban lévő fix elemeket, platformokat vagy falakat tartalmazák, amelyekkel az objektumok tudnak ütközni.

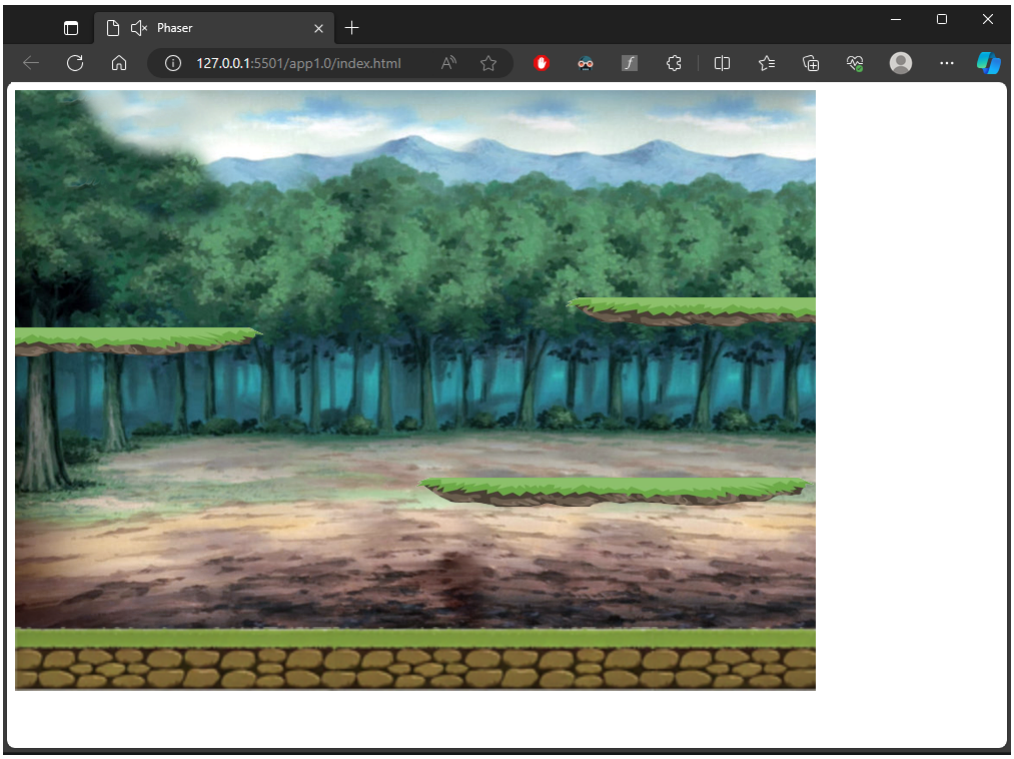
```
    platforms = this.physics.add.staticGroup();
  platforms.create(400, 568, 'ground').setScale(2).refreshBody();
    platforms.create(600, 400, 'block2');
    platforms.create(50, 250, 'block2');
    platforms.create(750, 220, 'block2');
```

Programkód 4.12. GameScene.js platform megvalósítása

A statikus csoporttal az elemeket a create metódussal lehet létrehozni. A zárójelen belüli rész tartalmazza az X és Y kordinátákat tartalmazza és egy kép fájlnak az azonosítóját. A refreshBody() egy metódus, amely egy objektum fizikai testének frissítését végzi el. Ezt a metódust akkor használjuk amikor egy objektum pozícióját vagy méretét programatikusan módosítjuk, és azt szeretnéd, hogy a fizikai testének tulajdonságai is ezen változásoknak megfeleljenek. Ebben az esetben a platformunk alja. Az elkészült platformunk a 4.6 ábrán látható.

Most hogy készen lett a platfomunk már csak a egy karakter kell, hozzá, amit tudunk mozgatni. A karkaterünk amit mozgatunk az nem csak egy kép, hanem egy sprite-sheet. A sprite-sheet az egy olyan kép, amely több kis grafikai elemet, úgynevezett sprite-ot tartalmaz. Ezek a sprite-ok gyakran egyetlen képkereten belül vannak elrendezve, ami optimalizált megjelenítést tesz lehetővé a játékprogramok és animációk számára. Azért használunk sprite-sheet-et mert egyetlen kép betöltése és megjelenítése hatékonyabb, mint sok különálló kis kép betöltése, különösen, ha a játék sok kis grafikai elemet használ.

A következő programkódokban fogom részletes magyarázni, hogyan is működik. Először is a preload függvényben hozzá adjuk a spritesheet kulcs szót, ami segít, sprite-sheetként használni a kép fájlunkat. Ezek után a szokásos módon, hozzá adjuk az egyedi azonosítót ami



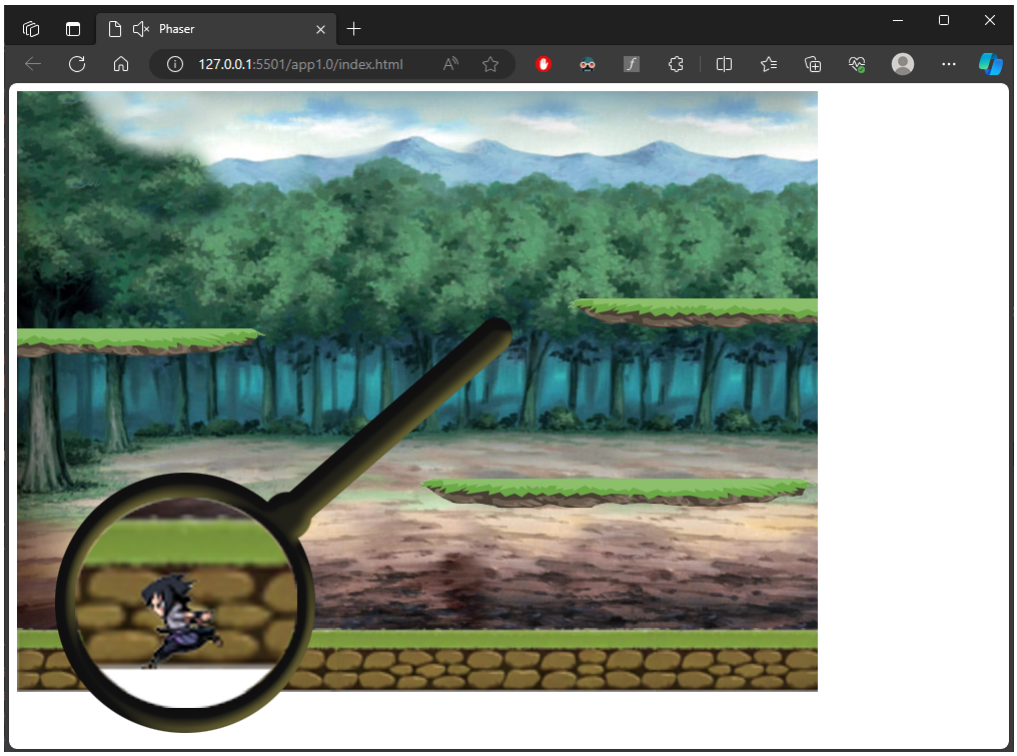
4.6. ábra: .

jelen esetben a dude szót kapta. Majd megadjuk a fájl elérési utvonalát. Miután ezekkel meg vagyunk meg kell adnunk a képkockák méretét és magasságát, ami jelen esetben 32 és 48.

```
preload()
{
this.load.spritesheet('dude', 'assets/sasuke.png', { frameWidth: 32,
 frameHeight: 48 });
  }
```

A következő programkódban fogjuk hozzá adni a játékhoz a sprite-sheet-ünket. Ezeket a kódokat a `create()` függvényben fogjuk megvalósítani. Először is hozzáadjuk a játék fizikájához a következő program kóddal, a `player` egyed segítségével. A `sprite()` metódussal a zárójellel közre fogott részben található az `X`, `Y` koordinátán való elhelyezése és az sprite-sheet egyedi azonosítója. A `setBounce(0.2)` metódus beállítja a karakter ugynevezett ”bounce” vagy ugráló tulajdonságát. A 0.2 értékre állítottuk ami azt jelenti, hogy a karakternek 20%-os pattogó hatása van, amikor ütközik valamivel. Ez a funkció lehetővé teszi, hogy a karakter kis mértékben visszapattanjon az ütközéseknél. A `setCollideWorldBounds(true)` metódussal be tudjuk állítani, hogy a játékos karakter ütközzön-e a játék világhatáraival vagy sem. Ha az érték `true`, akkor a karakter a világhatárokhoz fog ütközni, és nem tud túl menni azokon. Ez megakadályozza, hogy a karakter a játék világából kilépjen, ami az 4.7 ábrán van szemléltetve.

```
  create()
{
player = this.physics.add.sprite(100, 450, 'dude');
  player.setBounce(0.2);
player.setCollideWorldBounds(true);
}
```



4.7. ábra: Spritesheet hozzáadása a játék világához.

Ezenfelül, hogy karakterünk tudjon mozogni a platformon szükségünk lesz egy kollízió érzékelésre, ami azt jelenti, hogy a fizikai motor, fogja tudni kezelni a karakterünk és a platform közötti ütközéseket. Például, hogy tudjuk mozgatni karakterünket a platformon. A 4.13 programkódon látható a kollízió érzékelő metódus, ezáltal teljes lett a fizikai csoporthoz való hozzáadás. Az 4.8 árbán látható, az ütközés érzékelése a két elem között.

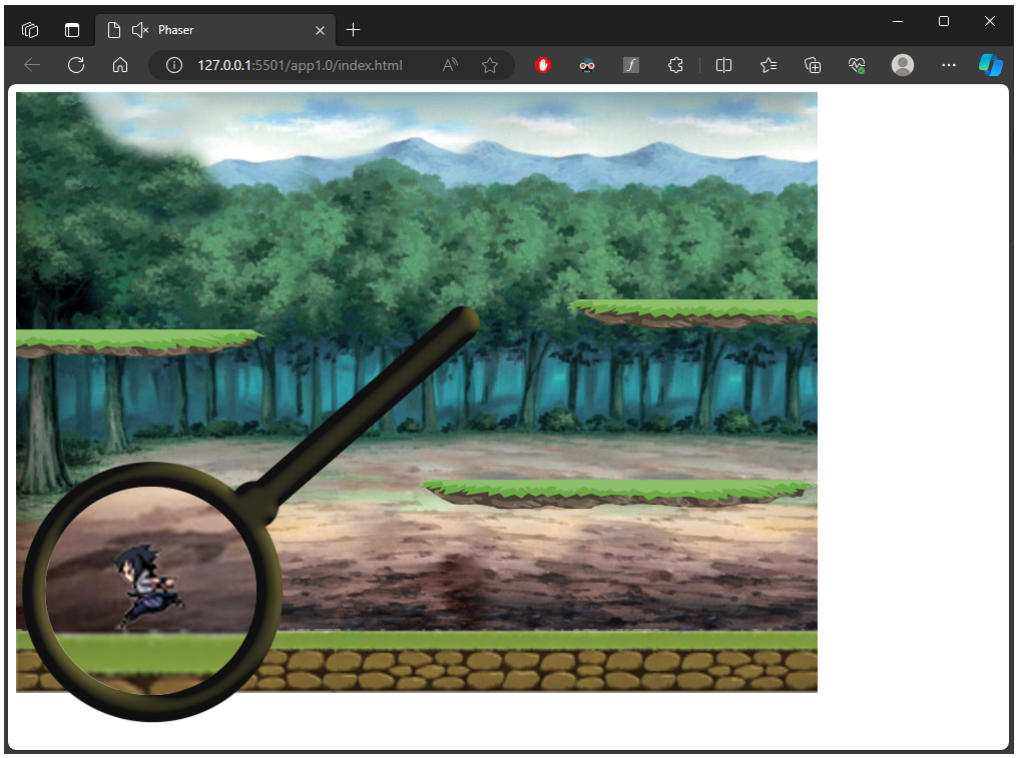
```
1  this.physics.add.collider(player, platforms);
```

Programkód 4.13. Kolízió érzékelés az egyedek között

A karkater mozgásának az animációját fogom magyarázni. Amit az alábbi (4.14) programkódon láthatunk. A `this.anims.create()` metódus létrehoz egy új animációt. A `left` kulcs, ami azonosítja az animációt. Ezt a kulcsot később használjuk a játékban az animáció lejátszásához. Az ezután kódsor a definiálja a képkockák animáció kereteit, hogy az animáció a 0-tól 3-ig tartó keretek felé tartson. A `frameRate` kóddal az animáció sebességét tudjuk meghatározni, ahol a 10 kékocka per másodperc a frameRate értéke. Ezzel szabályozva tudjuk beállítani, hogy milyen gyorsan játssza le az animációt. Amikor a `repeat` kód az érték -1-re van állítva az azt jelenti, hogy az animáció végtelen alkalommal ismétlődik meg. Ha más értéket adnánk meg, például 0-t, akkor az animáció csak egyszer játszódna le.

A jobbra felé tartó animációt az előbb említett módon tudjuk megcsinálni, csak `right` kulcsot használunk és a képkockát értékét a számunkra megfelelő irányba állítjuk.

A `turn` kulcs elnevezésű animáció egyetlen keretet használ a sprite sheet-ből. Ez az animációt akkor valósul meg amikor a karakterünkkel egy helyben állunk.



4.8. ábra: Spritesheet és platform ütközésének hozzáadása.

```
    this.anims.create({
        key: 'left',
        frames: this.anims.generateFrameNumbers('dude', { start: 0, end: 3 }),
        frameRate: 10,
        repeat: -1
    });

    this.anims.create({
        key: 'turn',
        frames: [ { key: 'dude', frame: 4 } ],
        frameRate: 0
    });

    this.anims.create({
        key: 'right',
        frames: this.anims.generateFrameNumbers('dude', { start: 5, end: 8 }),
        frameRate: 10,
        repeat: -1
    });
    cursors = this.input.keyboard.createCursorKeys();
```

Programkód 4.14. karakter mozgásának animációja

Az animációk után a karakterünk vezérlésére fogok kitérni amit a 4.15 program kódon szemlélhető meg. A `this.input.keyboard` kódrész hozzáférést biztosít a Phaser billentyűzetkezelő funkciójához. A `.createCursorKeys()` metódus létrehoz egy objektumot, amely tartalmazza a billentyűzet kurzor gombokhoz rendelt eseménykezelőket. Ez a cursors változó most olyan objektumot tartalmaz, amely a következő tulajdonságokkal rendelkezik.

1. up: Felnyíl gomb.
2. down: Lefelé nyíl gomb.
3. left: Balra nyíl gomb.
4. right: Jobbra nyíl gomb.

Ezen értékeket fogjuk használni a játékban a karakter mozgatásához.

```
        cursors = this.input.keyboard.createCursorKeys();
```

Programkód 4.15. karakter mozgatása

Továbbá még szükségünk lesz az `update` függvényre, amiben olyan eseményeket valósítotunk meg mint az irányítás és a Game Over felírate felbukkanása.

Ebben a függvényben valósúl meg a mozgás tényleges része, hogy a sprite-sheetünk képkockájának és a karakterünk helyének frissülését. Az 4.16 program kódón látható.

```
if (cursors.left.isDown)
{
    player.setVelocityX(-player_config.player_speed);

    player.anims.play('left', true);
}
else if (cursors.right.isDown)
{
    player.setVelocityX(player_config.player_speed);

    player.anims.play('right', true);
}
else
{
    player.setVelocityX(0);

    player.anims.play('turn');
}

if (cursors.up.isDown && player.body.touching.down)
{
    player.setVelocityY(player_config.player_jumpspeed);
}
```

Programkód 4.16. karakter mozgása az update függvényben

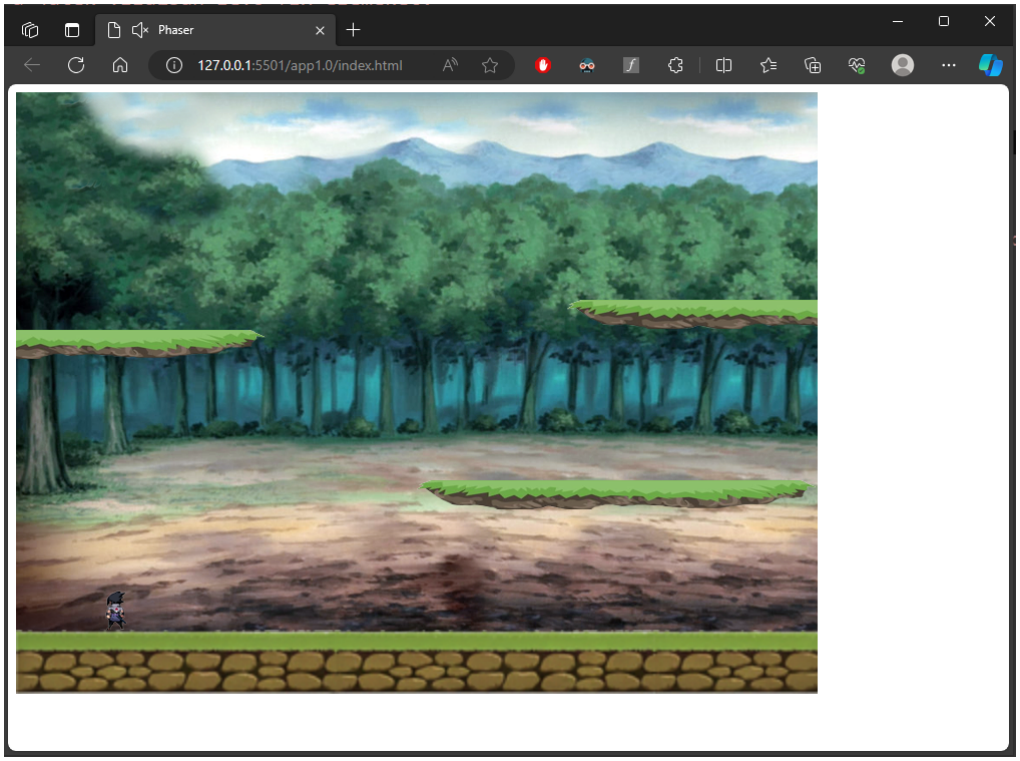
A programkódot fentről lefelé fogom részletesen magyarázni. Ha a balra nyílgomb lenyomva tartását a `(cursors.left.isDown)` kóddal lehet megvalósítani, ekkor a karakter vízszintesen `-X` tengelyen mozog a beállított sebességgel, amit `player_config` függvény és a `.player_speed` metódus együtessel tudunk beállítani. A `player.anims.play('left', true);` sor lejátsza a hozzá rendelt `left` animációt.

A jobbra tartó mozgás, az előzőhöz hasonlóan lett elkészítve, csak a `left` helyet `right` szót használtuk, és a karakter a `+X` tengelyen mozog.

Amikor álló helyzetben van a karakterünk,tehát egyik iránygomb sincs lenyomva, akkor a `(player.setVelocityX(0);)`, és a `turn` animáció játszódik le. Amikor ugrani szeretnénk akkor ha a felfelé nyílgombot kell nyomva tartanunk amiért a `(cursors.up.isDown)` kódsor felel, és ez a mozgás akkor teljesül be amikor a játékos karakter a talajon van ami a

(`player.body.touching.down`) kód visz véghet, ekkor a karakter felfelé mozog a beállított ugrási sebességgel. Ha nem használnánk a `.touching.down` metódust, akkor bárhonnan tudnánk ugrani, akár a levegőben és akkor egy flappy bird-höz hasonló játék mozgást kapnánk.

Jelenleg a játékunk kinézete a 4.9 ábrán szemlélhető meg.



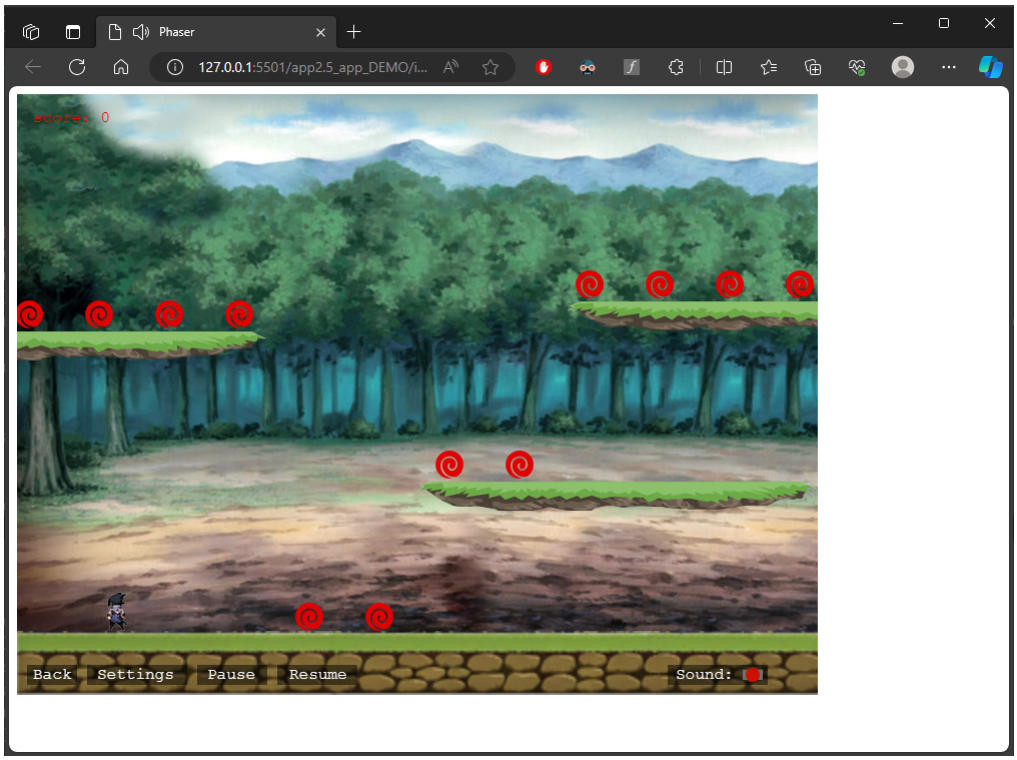
4.9. ábra: Spritesheet hozzáadása a játék világához.

A `gameOver` egyed a `variables.js` fájlban lett deklarálva. A `gameOver` egyed akkor jelenik meg, amikor hozzá ér egy kés a karakterünkhöz, ami a következő képpen lett megvalósítva. Ez az egyed végrehajtja a benne lévő metódusokat, például leállítja a háttérzenét (`bgMusic.stop()`), megjeleníti a játék vége feliratot és, restart gombot. Ezeket a müveleteket a `.visible = true;` metódussal lehet megjeleníteni, ha a metódus `true`-ra az-az igazra állítjuk. Ha `false` értéket adunk neki, akkor eltünteti az elemet. Ezt a 4.17 programkódon tekinthetjük meg.

```
1  update ()
2  {
3        if (gameOver)
4        {
5      bgMusic.stop();
6      gameOverTxt.visible = true;
7      restTxt.visible = true;
8      restButton.visible = true;
9      return;
10   }
11   }
```

Programkód 4.17. Game Over felirat megjelenítése

A játékban szereplő érméket az alábbi módón oldottam meg. A 4.18 programkódban szereplő a `coin` egyed hasonlóképpen hozzá adtuka játékunk fizikájához, mint a `platforms` és



4.10. ábra: Érmék a platformon.

dude egyednél.

Továbbá ennél a kódnál, nem egy érme egyed fog létre jönni, hanem több, amit a `repeat` kóddal meg tudjuk adni, hogy hány példányt hozzon létre. Mi most 11 érmét hoztunk létre. Emaitt az érmék koordinátáit is máshogyan fogjuk megadni, nem pedig úgy mint az előző egyedeknél.

A `setXY: { x: 12, y: 0, stepX: 70 }` kódsorral lehet beállítja az `X` és `Y` kezdőkoordinátákat, valamint az `X` irányú lépésközt a sprite-ok között. Mi a 4.18 programkódnál az érméket 12 pixel szélesen, 0 pixel magasságnál kezdődnek, és 70 pixel távolságra helyezkednek el egymástól.

A következő sor kód `coins.children.iterate(function (child) {});` végigiterálja a `coins` csoport gyerekein, és minden egyes érme esetén be fogja állítani a `setBounceY` metódust, hogy véletlenszerű és különböző úgy mond ugráló értékeket rendeljen az érmékhez. Ezáltal mindegyik érme egy kicsit másképp fog pattogni a platformon, amikor a játék elején lepottyannak az égből.

A `balls = this.physics.add.group();` rész létrehoz egy üres csoportot a `balls` változón belül, hogy a kések ne ütközzenek az érmékkel. Ehez még szükséges egy kolízió érzékelő metódus is ahol `coins` és a `platforms` egyedek vannak megadva.

```
coins = this.physics.add.group({
  key: 'coin', repeat: 11,
  setXY: { x: 12, y: 0, stepX: 70 }
});
coins.children.iterate(function (child) {
    child.setBounceY(Phaser.Math.FloatBetween(0.4, 0.8));
});
this.physics.add.collider(coins, platforms);
```

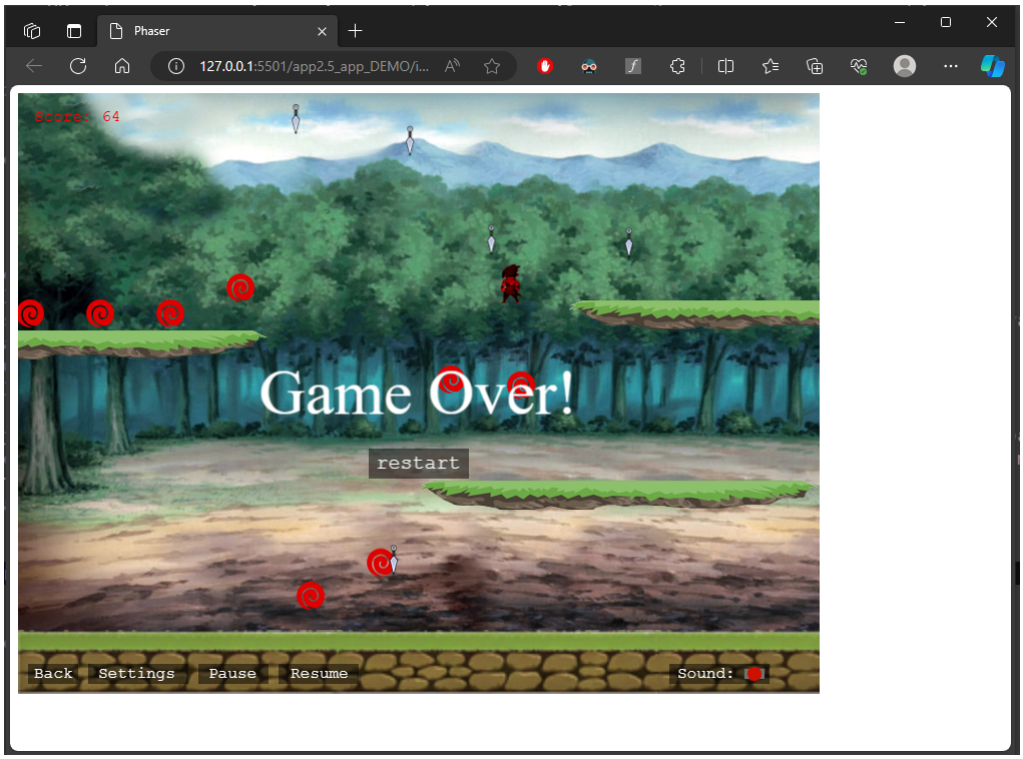
Programkód 4.18. Érmék hozzá adása a játék fizikájáához

Ahoz hogy fel is tudjuk venni az érméket egy `overlap` eseménykezelőt kell még hozzá adnunk a játékmotorhoz. Ez az a metódus azt csinálja, hogy amikor két objektum területei valamilyen mértékben átfednek egymással azt érzékel. Ami a **??**jasi:collectcoinoverleap) programkódon lehet megtekinteni. A `player` egyed és a `coins` csoport közötti átfedése kor, a `collectcoin` függvénnyel hívódik meg. A negyedik paraméter egy opcionális callback függvény, amely meghívódik, ha nincs átfedés. Itt `null` van, tehát nincs ilyen callback függvény. Az ötödik paraméter az a kontextus, amelyen belül a callback függvény meghívódik. Ebben az esetben a `this` a játék objektumot jelenti.

```
this.physics.add.overlap(player, coins, collectcoin, null, this);
```

Programkód 4.19. Érme gyöjtű függvény használata `overleap` metódussal

A 4.10 ábrán az érmékkel teli platform látható.



4.11. ábra: Kés és a karakter kollíziója.

A következő kódal létre hozzuk, és hozzá adjuk a játékunk fizikájához a `balls` egyedet, amit a `variables.js` fájlban deklaráltunk. Ezenkívűl kolízi érzékelő metódust adtunk hozzá, hogy érzékelje a platform ütközését, aminek a működése korábban a magyarázva lett.

```
balls = this.physics.add.group();
this.physics.add.collider(balls, platforms)
```

Programkód 4.20. kések hozzá adása a játék fizikájához

Ahoz hogy a kések ugymond megtudják ölni a karakterünket a `hitball` függvényre lesz szükségünk, amit a 4.4.3 alfejeztnél lett magyarázva és még egy kolízió érintési metódusra ami a 4.21 programkódon szemlélhető meg. A 4.11 ábrán a `balls` elem kolíziója látható.

```
  this.physics.add.collider(player, balls, hitball, null, this);
```

Programkód 4.21. kések érintésének hozzá adása a játék fizikájához

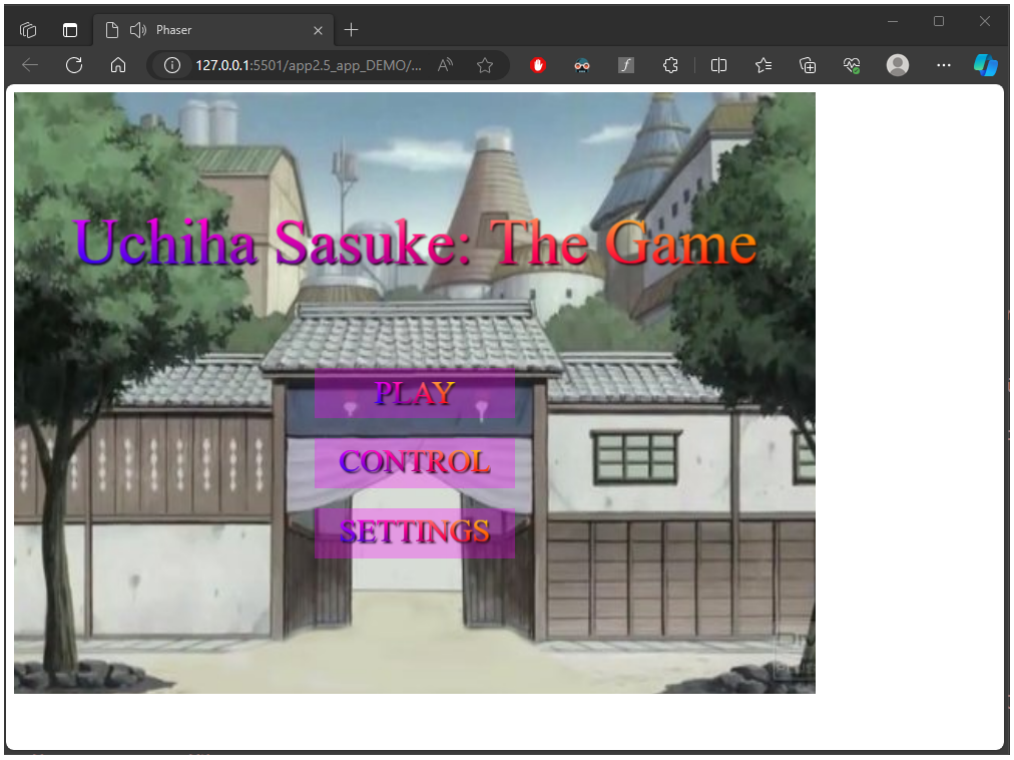
5. fejezet

Tesztelés

Ebben a fejezetben az elkészült játékot fogom bemutatni, hogy mely felületen mit lehet csinálni, és hogyan lehet játékkal játszani.

5.0.1. A játék menüje

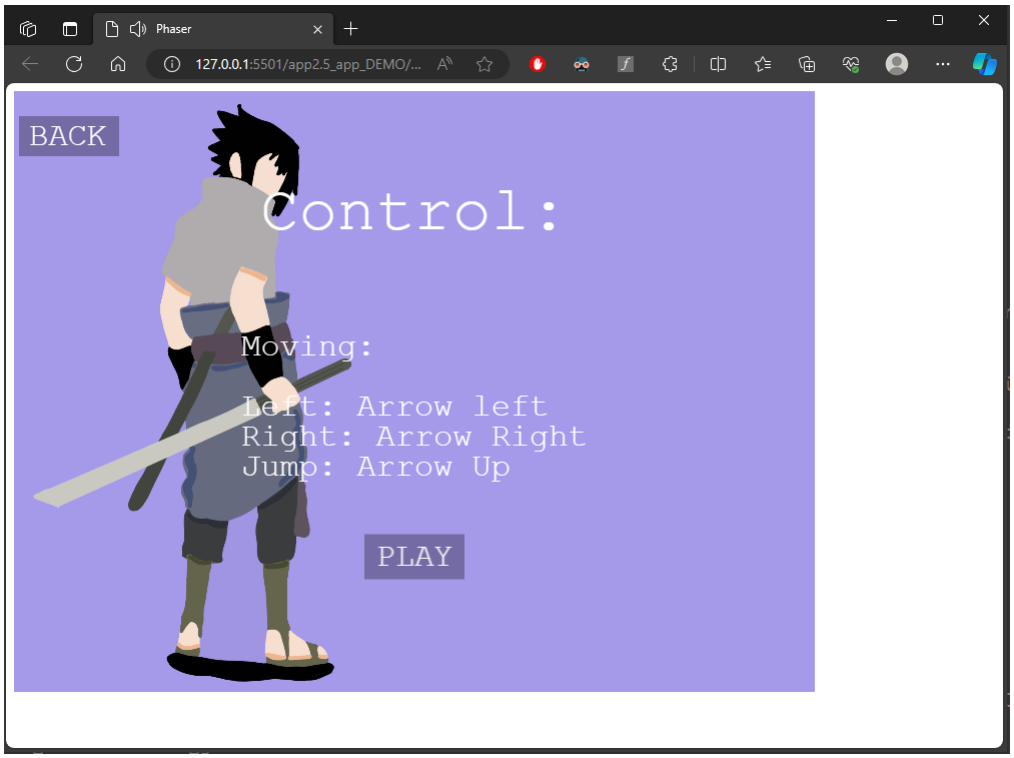
A játék menüjében látható elemeket fentről lefelé fogom ismertetni. ELegelőször a játék címét látja ami ”Uchiha Sasuke: The Game”. Az alatt látható egy Play gomb, amivel a játék szinterére tudunk ugrani. A középső gomb az a Control gomb, ahol megtudjuk nézni, hogy a játékot hogyan kell irányítani. A legalsó gomb az a Settings gomb. Itt játát beállításait tudjuk kezelni. Amit a az 5.1 láthatunk.



5.1. ábra: Menü

5.0.2. Control menüpont

A control menüpontben meg tudjuk nézni mely gombokkal kell irányítani a játékot és a ball felső sarokban található egy vissza menüben gomb, középen alul meg egy play gomb, amivel a játékot indítjuk el. A 5.2 ábrán látható a control menüpont.



5.2. ábra: Control menüpont.

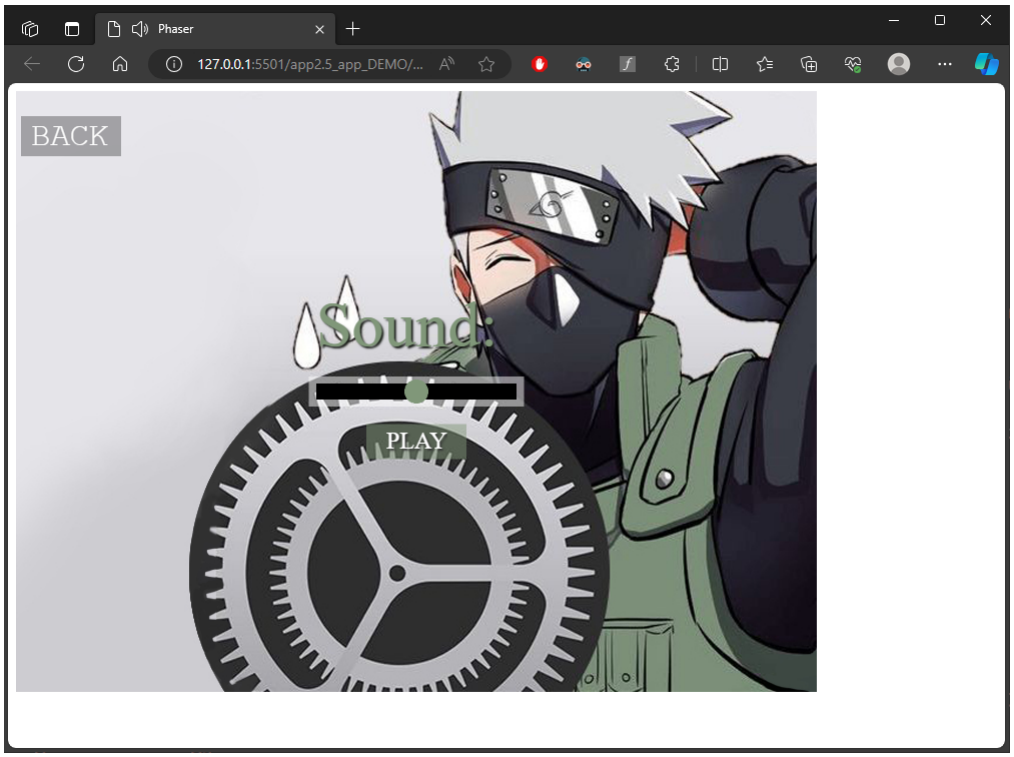
5.0.3. Settings menüpont

A settings menüpontban a játék háttérzenéjének a hangerejét tudjuk beállítani egy csuszka segítségével. Itt is a ball felső sarokban található egy vissza a menübe gomb és középen alul egy play gomb. Az itt beállított hangerő értéket megjegyzi, ha játékszinterére tovább lépünk,hogy, ha menübe megyünk akkor visszaáll az alapértelmezett hangerő értékre. A 5.3 ábrán látható a settings menüpont.

5.0.4. A játékszintér

A játékszinterét a menüből, a setting, és a control felületekről tudjuk elérni gomb segítségével. Ennél a szintérnél alul látható számos gomb. Ballról jobbra tartva fogom ezeket prezentálni. Először is egy vissza gombot látunk amivel a menübe tudnunk vissza menni. Utána látható egy Settings gomb amivel a beállítások menüpontra tudunk ugrani. A harmadik gommb amit látunk egy pause gomb ami megállítja a játékot, ilyenkor nem tudjuk irányítani a karakterünket. A következő gomb egy resume gomb amivel eltudjuk indítani a megállított játékot. A jobb alsó szélen található egy mini csuszka amivel a játék hangerejét tudjuk lejebb vinni. Továbbá ball felül található egy piros szinnel szedett pontszámláló.

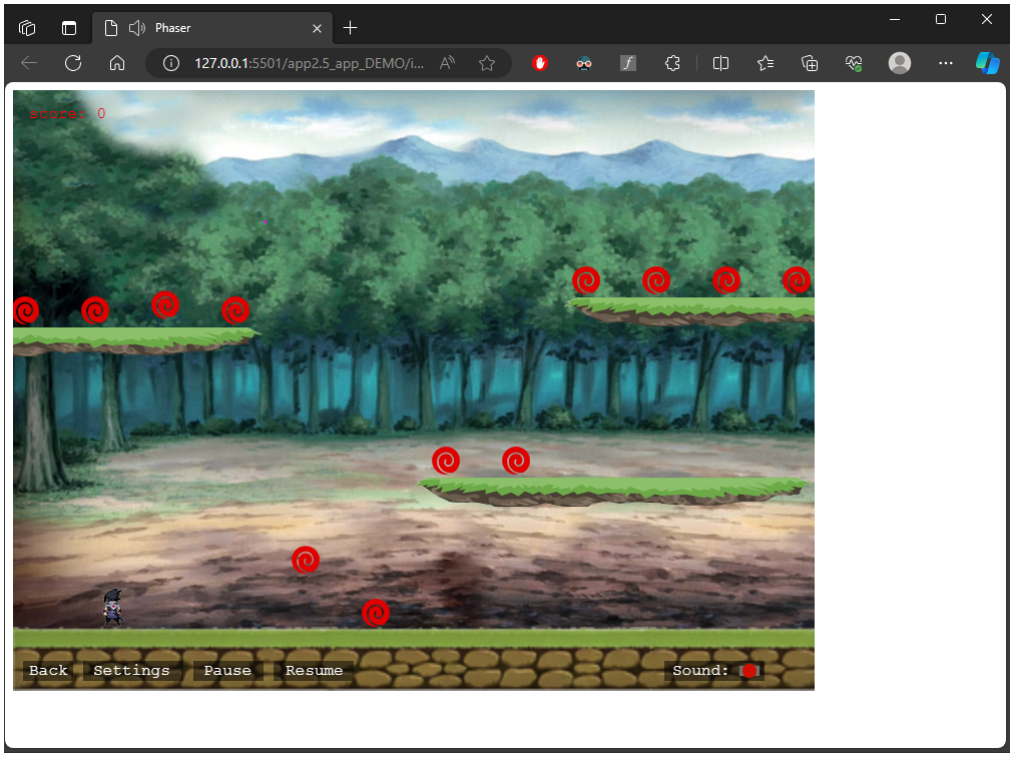
A karakter amit tudunk mozgatni egy nekünk háttal álló kék és szürke ruhát viselő ember. Akit tudunk a kurzor nyilaival mozgatni. A jobb felé mutató nyillal jobbra tudunk menni. A



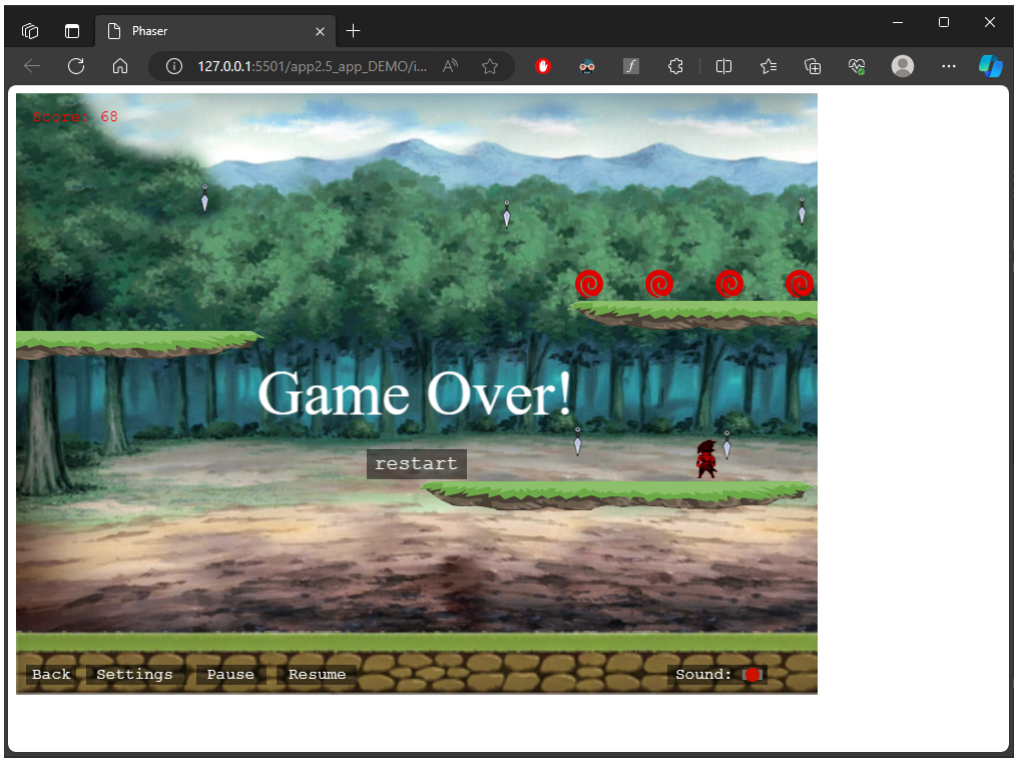
5.3. ábra: Settings menüpont.

ball felé mutató nyillal ballra. Ezekben az irányokban az irányított karakter futó mozgás animáció mutat. Amikor a fel fele mutató nyilat megnyomjuk akkor ugrik egyet a karakterünk, amit a gravitáció húz vissza. A lefelé mutató nyilra nem reagál a karakterünk. A 5.4 ábrán látható a játék szintere látható.

Az irányított karakterrel a játék elején leeső érmékhez hozzá érve össze tudjuk gyüjteni őket, ezekből az érmékből 12 darab van, minden 12. felszedett érme után újból esik le még 12 darab érme, és meg egy késnek látszó tárgy is, ami a platformon össze vissza pattog. Minnél több érmét szedünk fel annál több kés jelenik meg. Ha hozzáérünk ezekhez a késekhez a játék képe meg vagy az irányított karakterünk piros színű lesz és még egy hang effektet hallhatunk. Közben megjelenik egy ”Game Over” felírat és egy restart gomb, ami újra tölti a játékot, majd a menüben találjuk magunkat. A 5.5 lábrán látható a control menüpont.



5.4. ábra: Játékszintere.



5.5. ábra: Játék vége.

6. fejezet

Összefoglalás

A Phaser játékmotorral fejlesztés számomra nagyon tetszett és kihívásokkal teli. Mivel számos időtt eltöltöttem a Phaserrel való fejlesztéssel, és több különböző dolgott is kipróbáltam magáról a Phaserről azt jelenthetem ki, hogy tényleg ajánlható kezdőfejlesztőkszámára, hiszen én is tudtam csinálni egy egyszerűbb játékot.

A könnyű része számomra abból fakadt, hogy egyetemi tanulmányaim alatt, tanultunk JavaScriptet, és HTML-is. Továbbá már játékot készítettem ez előtt Unity-ben, ami egy Angry Birds-höz hasonló mechanikájú játék volt. Unitytiben C#-ot használtam, ami hasonló logika szerint működött mint a phaser, ellenben ott a különböző fájlok összekötése és a platform építés sokkal egyszerűbb volt, hiszen megcsinálta helyettem a Unity. Itt a Phaser kapcsán nem lehetett erről beszélni. Itt mindent magamnak kellett lekódolni. Ezeken kívűl a Phaser azért is jó választás mert kezdőprogramozók számára mert tényleg tömérdeknyi weboldal áll rendelkezésre ahonnan meglehet ezt tanulni és nagyon segítőkész közösséggel rendelkez, ezen felűl még a számos közösségi oldalon is fent vannak. Mindemelett amikor írtam a szakdogámat a Phaser hivatalos oldala 1 teljes hónapon át üzemenkívűl, volt és ideiglenes megosztottak számos olyan weboldalt ahonnan a példaprogramjaikat elérhatjük és a már elkészült ”játékokkal” lehet játszani úgy, hogy ezzel párhuzamosan lehetett változtatni a kódot.

A nehezebb része az volt, hogy megtalálni a phaser weboldalán a játék motor globális feltelepítését és a játék motor logikájára rá jönni. Játék motor logikáján a refaktorálás jelentett számomra nehézséget, mivel nehezen találtam rá példát, hogyan is mükődik. Nekem ez egy nem evidens dolog volt, a refaktorálás müködése, azért mert JavaScripttel még ezelőtt nem kellett csinálnom. Ezen kívűl nagyon sok régi phaser kóddal készült fél kész kód található az interneten, ami a fejlesztést többször is zsákutcába vidte. Ráadásul nagyon sok példa programkód TypeScriptben volt írva, nem pedig JavaScriptben. Tapasztalataim alapjána két Skript nyelv szintaktikája nagyon hasonlít, ezért könnyen összekevertem a két példa kódot.

A Phaser játék motorral, számos 2D flash játékot lehet fejleszteni, ahoz hogy kezdő programzókat elindítson ezen pályán arra alkalmas. Ha az ember eléggé kitartó és van egy kisebb ismerete a programozásterén akkor magától is megtudja tanulni ezt játék motort, mert a weboldalán többnyire minden információ elérhető, ahoz hogy egy bármilyen 2D-s játékot tudjon fejleszteni az ember. Mindemelett ez egy online felületre tervezünk játék, ezáltal szinte bármilyen eszközzel tudjuk használni a programunkat.

Számomra nagyon jó tapasztalatt szerzés volt a szakdolgozatt készítés mindez az egyetemi tanáraim, és konzulensem által tanult tudás nélkül nem tudtam volna véghez vinni.

CD Használati útmutató

Ennek a címe lehet például *A mellékelt CD tartalma* vagy *Adathordozó használati útmutató* is.

Ez jellemzően csak egy fél-egy oldalas leírás. Arra szolgál, hogy ha valaki kézhez kapja a szakdolgozathoz tartozó CD-t, akkor tudja, hogy mi hol van rajta. Jellemzően elég csak felsorolni, hogy milyen jegyzékek vannak, és azokban mi található. Az elkészített programok telepítéséhez, futtatásához tartozó instrukciók kerülhetnek ide.

A CD lemezre mindenképpen rá kell tenni

- a dolgozatot egy `dolgozat.pdf` fájl formájában,
- a LaTeX forráskódját a dolgozatnak,
- az elkészített programot, fontosabb futási eredményeket (például ha kép a kimenet),
- egy útmutatót a CD használatához (ami lehet ez a fejezet külön PDF-be vagy MarkDown fájlként kimentve).