Tartalomjegyzék

1. Bevezető

Első féléves egyetemi hallgatóként már foglalkoztatott, hogy milyen témát válasszak szakdolgozatnak. Többnyire vizuális típusnak tartom önmagamat, ezért valamilyen látványos, dizájnos projekttel szerettem volna előállni egyetemi tanulmányaim befejezéseként.

Szakirányok közül az alkalmazásfejlesztést mobil eszközökre specializációt választottam, ez a szakirány sokkal több lehetőséget tartogatott számomra a kreativitásban. A szakirány lehetővé tette, hogy betekintést nyerjünk a játékfejlesztés alapjaiba is, ezért már a tárgy megkezdése előtt próbálgattam a Unity fejlesztő környezetét, több oktató anyagot is tanulmányoztam, főleg a háromdimenziós projekteket. Nagyon tetszett és motivált, hogy objektumokat lehetett mozgatni, illetve háromdimenziós modellek mozgását meganimálni és irányítani ezeket. Könnyű volt a scriptek megírása is a C# nyelv ismeretében és a Unity-hez kiadott oktató anyag[1] és dokumentáció[2] rengeteget segített. Viszonylag kevés kódolással igen látványos eredményt lehetett elérni, ez volt az, ami igazán megfogott.

A játékfejlesztés kurzuson a kétdimenziós játékokra fektettük a hangsúlyt, ami sokkal egyszerűbb volt számomra a háromdimenziós próbálkozások után. Annak érdekében, hogy az gyakorlaton elhangzottakat gyakorolni tudjam, elkezdtem egy otthoni projektet, ahol több mindennel is kísérleteztem és megpróbáltam egy játszható kétdimenziós játékot összerakni, ami kihívás elé állítja a játékost. A kurzus elvégzésének végére az otthoni projektből egy jó alapokkal rendelkező játékig sikerült eljutnom. Ekkor határoztam el, hogy ebből a projektből egy kész játékot fogok készíteni, ami megállja a helyét a mobiljátékok piacán, illetve ellátni olyan tartalommal, funkciókkal, hogy szakdolgozat kritériumainak is megfeleljen.

A szakdolgozatomban szeretném bemutatni, hogyan is fejlődött egy egyszerű ötletből ez a játék, milyen szakaszai voltak fejlesztésnek, milyen problémákba, illetve zsákutcákba sikerült belefutni a fejlesztés során és ezekre milyen megoldást sikerült találni. Szeretnék kitérni honnan sikerült ötleteket meríteni, mely játékok voltak az iránymutatók a fejlesztés során. Végső soron pedig kitérni a felületre, megjelenésre és milyen fejlesztési potenciál van még a játékban.

2. Alkalmazás bemutatása, feladatok meghatározása

2.1. Az ötlet

Az egyik játékfejlesztés gyakorlaton egy rosszul megírt irányítás script adta az alapötletet, amit az otthoni projektemben valósítottam meg. Az ötlet az volt, hogy egy olyan játékot valósítsak meg, ahol az irányítás okozza a nehézséget és zavarja meg a játékost. Ezt több féle módon is ki lehet vitelezni, például felcserélni az irányokat, tehát ha balra szeretnénk elmozdulni, akkor a jobb irányzékot kell használnunk, ez kicsit olyan mintha a másik kezünkkel próbálnánk írni, nagyfokú koncentrációt igényel. Úgy gondoltam ez önmagában még nem elegendő, úgyhogy kitaláltam azt az irányítást, ami az egész játék alapját jelentette. Egy objektum, ami forog a saját középpontjánál fogva, ezt pedig ellátjuk egy egyszerű vertikális mozgással, ami azt fogja eredményezni, hogy amerre az objektum felső része néz arra lesz mindig a fölfele irányzék és mivel forog az objektumunk, így állandóan változik. Ezzel a megoldással igen nagy koncentrációra van szükségünk, hogy elmozduljunk egy pályán előre. Ez a mozgás a tesztek során nagyon hasonlított a legyek össze-vissza cikázására, ezért az irányítható karakterünk egy légy lett, majd később a játék nevét is ez ihlette meg.

Megfigyelések alapján azok a játékok, amelyek valamilyen kihívást biztosítanak a játékos felé, sokkal nagyobb népszerűségnek örvend és huzamosabb ideig köti le a játékosokat. Elég megfigyelni az elmúlt pár év siker játékait telefonon, a legjobb példa a Flappy Bird[3] nevű játék, amelyben egy végtelenül egyszerű feladata van a játékosnak, azaz két darab cső között átrepülni, a nehézséget az irányítás okozta ebben a játékban is. Erre a tényre alapozva indultam el a fejlesztés során és ez volt a fő szempont, ami alapján a funkciókat is fejlesztettem. A Flappy Bird volt az a játék, ami leginkább inspirált, de például a Color Switch[4] vagy a Piano Tiles[5] nevű játékokból is merítettem ötletet, de egy nagyon régi pc-s játék a South Park Cartman's Authoritah[6] nevű kétdimenziós játék szintén segített a végleges játék elkészítésében.

Ezeket a játékokat alapul véve világos volt, hogy infinite runner-t fogok csinálni, mivel egy épített pálya, szintek sok időt emésztettek volna fel a megtervezésükkel, illetve megépítésükkel és ráadásul monotonitást is eredményezett volna. Így több pályaelem legyártása mellett döntöttem, amelyek majd véletlen sorrendben kerülnek be a pályára, így garantálva a monotonitás elkerülését. Ezelőtt olyan ötletek is terítéken voltak, ami szerint nem pályaelemek generálódnak, hanem egy csővezeték, ami több irányba (föl, le, jobbra,

balra) kanyarodik, illetve kunkorodik, megbontva ezzel az infinit runner játékokban megszokott, egy megadott fix irány felé való menetelt, de végül elvetettem ezt.

Sikerült tehát lefektetni az alapokat, így ezeket követve több feladat is megvalósításra várt. Különös figyelmet tulajdonítottam annak, hogy elkerüljem a monotonitást vagy, hogy unalmassá váljon a játék maga. Szerettem volna, ha sok ideig le tudja kötni a játékosokat, és hogy mindenki megtalálja a kihívást benne, ezért több változó tényezőt is beépítettem a játékban.

2.2. Feladatok meghatározása

Az első feladat a jó irányítás megtalálása, és ahhoz tartozó értékek pontos beállítása volt. Fentebb már szó volt az irányítás működéséről, ez nagyon jól be is vált, az objektumnak van egy sebessége, amit gomblenyomásra fel fog venni, ezt kisebb állítgatás, próbálgatás után be lehetett lőni. A lényeg annyi volt, hogy ne legyen túl gyors, nehogy kontrollálhatatlan legyen a játékos számára, de ne legyen túl lassú sem, mert akkor nem lehet átjutni majd az akadályokon vele. Ezen felül megtalálni az objektum megfelelő forgási sebességét is, itt ugyanúgy azt az egyensúlyt kellett eltalálni, mint a sebességnél.

Infinite runner révén egy irányba kell haladni a karakterünkkel, jellemzően ilyenkor egy állandó sebességgel, megállás nélkül megy előre a karakterünk, sőt egyre gyorsul az előrehaladás mértékétől függően. Ezt az ötletet teljesen elvetettem, a karakterünk mozgását követi a kamera, tehát csakis gomblenyomásra történik mozgás. Ha egy állandó sebességű mozgást alkalmaztam volna túl nehéz lett volna az előrehaladás vagy akár egy új mozgást is ki kellett volna dolgoznom, ami felborította volna az eddigi munkálatokat.

Természetesen ilyenkor fenn áll az a probléma, hogy a játékos direkt nem fog mozogni a karakterrel. Valahogyan rá kell kényszeríteni a mozgásra a játékost, ezt több féle módon is el lehet érni. Ezt az infinite runner játékokban szokták úgy is megoldani, hogy a játékos mögött közeledik valamilyen objektum, ami elpusztítja a karakterünket, ha nem igyekszünk eléggé. Tipikusan jó példa erre a Subway Surfers[7] nevű játék, ahol egy rendőr elkap bennünket, ha túl sokat hibázunk. Ebben az esetben előfordulhat, hogy a mögöttünk érkező objektum túlságosan megnehezíti a játékmenetet. Megeshet, hogy több időre van szükségünk egy-egy akadályon való átjutáshoz, ezért ez a megoldás túlságosan bünteti a játékos, a már amúgy is nehézkes irányítás okozta előrejutásban. Inkább egy időbüntetésen alapul rendszert vezettem be, ami sokkal igazságosabb és jobban is illik bele a játék koncepciójába.

Ezeknél a játékoknál már említettem, hogy egy irány felé mozgás a jellemző, azaz a kamera mozgását csakis egy tengelyen mozgathatjuk. Ez azt jelenti, hogy karakterünket más felé nem fogja követni kamera, csak amit megengedünk neki. Ilyenkor előfordul az, hogy ha karakterünkkel az ellenkező irányba mozdulunk el, akkor ki tudunk menni a kamera látómezőjéből. Meg kell oldani, hogy ez ne fordulhasson elő, több megoldás van itt is. Az egyik, ha a karakterünk mozgásterét csökkentjük, ezáltal bezárjuk vagy a másik megoldás, ha a kamera azon részeit lezárjuk, amerre nem engedélyezünk mozgást.

Ha van karakterünk és el tudunk indulni egy irányba, akkor szükség lesz elemekre, amelyek segítségével a végtelenített pálya hatását lehet kelteni. Az egyik legszembetűnőbb elem az a háttér, ami célja, hogy úgy érezzük, haladunk előre a térben. Ezt legegyszerűbben seamless hátterek használatával tudjuk kelteni, amikből ha egymás mellé rakunk többet, akkor olyan mintha egymást folytatnák. Ezt lehet fokozni akár paralax görgetéssel is. Ami fontos, az a háttér mozgatása, a pontos illesztés, illetve hogy milyen logika alapján görgetjük a hátteret, úgy hogy ne tűnjön fel ez a játékosnak.

Mikor már egy olyan környezetben tudunk mozogni karakterünkkel, ami tényleg azt az illúziót kelti, hogy haladunk előre egy végtelenített rendszerben, akkor fel lehet tölteni ezt a teret pályaelemekkel. Szintén fontos több típusú és nehézségű elemet kitalálni, hogy ne legyen túl monoton a játékélmény. Az egyre nehezedő játékmenet vagy pálya pedig, majdnem minden játék egyik alappillére. A változatosságnál fontos, hogy minden játékelemmel egy teljesen más gondolkodásra kényszerítsük a játékot, hogy ne mindig ugyanazt a stratégiát használja ezeknek az akadályoknak a kikerülése során. Fontos az is, hogy ne csak statikus akadályok legyenek, hanem dinamikus, mozgó elemekkel is találkozzon a játékos, mivel ezzel nem csak nagyobb kihívás elé tudjuk állítani a játékost, de látványosabbá és élvezetesebbé tehetjük a játékélményt. Ezen szempontok alapján kell megtervezni és legyártani ezeket az elemeket, majd valamilyen súlyozás alapján sorrendet kialakítani közöttük, úgy hogy a nehezebb akadályokkal csak később jelenjenek meg a játéktéren. Hogy még inkább elkerüljük a monotonitást, ezért véletlen sorrendben kerüljenek az elemek a pályára.

Nagyon fontos, hogy valamilyen formában jutalmazza a játék a játékosokat, ez az, amiért játszanak az emberek, hogy minél több pontot gyűjtsenek, vagy minél messzebbre jussanak el. Szükség lesz a játékban pontokra, ami alapján be lehet azonosítani, ki milyen messzire jutott el a játék során.

Természetesen fontos, hogy valamikor véget érjen a játék, így egy bizonyos esemény hatására következzen is ez be. Általában ütközés hatására szokott megtörténni, amikor

megsérül a karakterünk, de nem lehetünk túl szigorúak a játékoshoz így több esélyt kell neki adni, azaz több élettel kell ellátni, de itt is meg kell találni a megfelelő egyensúlyt. Ezek mellett a karakter ütközését jól lekezelni más tárgyakkal, ha ezt nem sikerül jól végrehajtanunk és például egy ütközés során egyből meghalunk, akkor a játékos igazságtalannak vélheti a játékot.

Nem szabad megfeledkezni arról sem, hogy a játékot bár gépen fejlesztjük, telefonon vagy táblagépen fog futni. Ez azt jelenti, hogy fel kell készíteni több féle felbontásra is, hogy minden eszközön ugyanúgy nézzen ki. Olyan kezelőfelületet biztosítani, ami kisebb kijelzőn is jól látható a felhasználó számára.

Ezek a fő feladatok, ami alapján elkészült a végleges játék, továbbiakban bemutatom milyen megoldásokkal oldottam meg a feladatokat, illetve milyen plusz funkciókat sikerült beépíteni, hogy még jobban eleget tegyek a kritériumoknak.

2.3. Alkalmazás bemutatása

A kész játékban a fentebb meghatározott feladatokat mind sikerült végrehajtani maradéktalanul, az irányítás során két féle mozgást alkalmaztam. Az egyik mozgás az a karakter saját középpontja körüli forgás, ez egy automatikus mozgás, amit előre beállított sebességgel végez. A másik pedig egy vertikális, tehát fölfelé, illetve lefelé való mozgást jelent. Ennek a két mozgásnak a kombinációja egy cikkcakkos mozgást eredményez a játékban, ami elsőre egy új és furcsa élményt nyújt, de bele lehet tanulni. Az irányítás kiismerése már önmagában kihívás elé állítja a játékost.

A játékosnak előrefelé, esetünkben jobbra kell haladnia, semmilyen más irányba nincs engedélyezve a kamera mozgása. A karakterünk előrehaladását követi a kamera, viszont van egy úgynevezett holttér, ahol a karakterünkkel mozoghatunk, úgy hogy ne mozdulna meg a kamera, ez a hely többnyire a képernyő bal oldalán van. Annak érdekében, hogy karakterünkkel, ne induljunk el más irányban a kamerát egy kerettel körbeépítettem, így mikor megpróbál távozni a játékos a játéktérről, akkor egy falba fog ütközni, ami életlevonással is jár.

Ezen a ponton merül fel a kérdés, ha a karakter mozgásával a játékos diktálja a tempót, mi lesz az, amivel haladásra tudjuk késztetni a játékost. Ezt úgy sikerült megoldani, hogy egy 4 másodperces számlálót helyeztem el a játékban, ami elkezd visszaszámolni, ha a karakter mozdulatlan, viszont egyből visszaugrik a kezdő értékre, ha újra megmozdul. Ez egy igazságos rendszer, mivel ennyi idő bőségesen elég még akkor is, ha nagyon lassan tud a játékos átjutni egy-egy akadályon. Viszont ha letelik az idő, akkor a halál esemény

következik be és elölről indul a játék, ez a halál esemény annyiban különbözik, hogy egy légycsapó jelenik meg hirtelen a képernyőn, érzékeltetve azt, hogy túlságosan lassú volt a játékos. Ez a megoldás tökéletesen beleillik a játék környezetébe.

A játéktéren úgy sikerült megvalósítani a végtelenített pálya érzését, hogy egy képet folyamatosan görgetünk háttérben, ez elé pedig generáljuk a pályaelemeket. A pályaelemek között található teljesen statikus, méretét tekintve hosszú, illetve forgó, csúszkáló és ki-be kapcsolódó is felfedezhető. A képernyő alján egy talaj generálódik, ami megakadályozza, hogy lefelé távozzon a játékos a kamera látószögéből, továbbá egy viszonyítási pontot biztosít, nem a levegőben fog lógni minden. A pályaelemek ehhez a talajhoz tökéletesen illeszkednek, így egy nagyon jó építésű pályát kap a játékos. A pályaelemeken való átjutást pontokkal jutalmazza a játék, egy akadályért egy pont jár, nincsen különbség nehéz vagy könnyű elemek között, mindenért ugyanannyi pontot kap a játékos. A pályaelemek sorrendbe vannak állítva nehézség szerint, és minden harmadik megszerzett pont után nehezebb és nehezebb játékelemek kerülnek a játéktérre. Eleinte főleg statikus elemekkel lehet találkozni, ha pedig elég messzire jutunk, összetettebb mozgó elemekbe is belefuthatunk. A pályaelemek teljesen véletlenül választódnak ki, de két ugyanolyan nem jelenhet meg egymás után.

A monotonitás elkerülése végett bevezettem egy olyan rendszert, ami garantálja, hogy fent kelljen tartani a koncentrációt és egy újabb véletlen faktorral bővíti a játékot. Minden megszerezett ötödik pont után megváltozik a karakterünk mozgása. Az bekövetkezhető irányításmódok a következők:

- A karakter forgás irányának megfordulása
- A karakter forgásának lelassulása
- A karakter sebességének lecsökkenés
- A karakter méretének megnövekedése
- A karakter normál értékekre visszaáll

Ezek teljesen véletlenül választódnak ki és minden ötödik pont után megváltozik, tehát öt akadályt kell teljesíteni az új megváltozott irányítással.

A halál eseménye a fent említett mozdulatlanságon kívül akkor is bekövetkezik, ha túl sokszor ütközünk valaminek. Minden pályaelem, a talaj és a képernyő oldalainál is ütközést érzékel a játék. A maximális ütközés számot három darabban határoztam meg, ezt az élet jelzi nekünk. Minden ütközés után egy darab élet levonódik és vigyázni kell, mert

ha szűk helyre beszorul a karakterünk, akkor gyorsan levonódik az összes élet. Két ütközés között van egy minimális idő arra, hogy kijusson a játékos a szorult helyzetből.

Minden eseményt hangok jeleznek számunkra, amelyek az adott eseményhez illenek, például az ütközéseket éles, szúrós figyelmeztető hang jelzi, míg a pontszerzést egy kellemes, jutalmazó dallam jelzi a játékos felé. Továbbá a játékmód megváltozásához is sikerült találni megfelelő hangjelzést, a mozdulatlanság által bekövetkező halált egy becsapódás hangeffekt jelzi.

A menüben kétállapotú gombokkal tudunk navigálni, amelyek alkalmazkodnak a képernyőméretekhez, viszont a telefonos irányítás érdekében be kellett iktatni egy-egy gombot a játéktéren, amellyel a karaktert lehet mozgatni, ez az érintőfelület miatt volt szükséges.

A játéktéren továbbá megjelenik a halál esetén egy a játék végét jelző képernyő is ahonnan tovább navigálhatunk a menübe, vagy új játékot is kezdhetünk. A képernyőn megjelenik még az elért teljesítményünk is a játékban.

A játék leírását követően szeretném bemutatni azt a környezetet ahol ezt meg is valósítottam, továbbá részletezni azokat az eszközöket, amik a segítségemre voltak a fejlesztés során.

3. Unity bemutatása

3.1. Unity, mint videojáték-motor és fejlesztő környezet

A Unity[8] egy videojáték-motor, amely segítségével háromdimenziós és kettődimenziós videojátékokat, illetve egyéb interaktív jellegű tartalmakat, mint például jeleneteket, látványterveket és háromdimenziós animációkat is lehet készíteni. A szoftver képes nagyméretű adatbázisokat kezelni, ami például egy nagyobb méretű játék esetén nem hátrány, ki tudja használni az animációk és kölcsönhatások képességeit, biztosítja a fények legenerálását valós időben vagy előre kiszámított fények megjelenítését. Nagy előnye még, hogy olyan játékmeghajtó motor, amivel lehetséges több platformra való fejlesztés, a következőket támogatja:

- Microsoft Windows (32bit,64bit)
- Mac OSX
- Linux
- Android
- OS
- Konzolok
- WebGL

Többek között ezért is döntöttem a Unity mellett, mivel lehetővé teszi ezt a fajta támogatást platformok között és ehhez nincs szükség csak a Unity fejlesztő környezetére[9] és C# nyelv ismereteire. Fontos megemlíteni, hogy játékmotor lévén nem képes modellek létrehozására, így azok létrehozásához valamilyen háromdimenziós modellező programra van szüksége a felhasználónak. Unity-n belül csak átméretezésre van lehetőség, esetleg modellek összeillesztésére textúrázására. Az viszont nagy segítség, hogy rengeteg beépített eszköz és modell áll rendelkezésre és van egy saját boltja[10], ahonnan ingyenesen vagy pénzért kész csomagokat lehet letölteni megspórolva ezzel időt és energiát. Amiket a Unity mint játékmotor elvégez a felhasználó számára azok a következők:

- Erőforrások kezelés
- Videokártya vezérlés
- Grafikai algoritmusok
- Fizikai szimulációk

- Karakter animációk
- Részecskerendszerek
- Kameramozgatás
- Multimédiás szolgáltatások, mint hangok és videók kezelés
- Beépített grafikus kezelő felületet biztosít
- Mesterséges intelligencia, állapotkezelés, útvonaltervezés
- Scriptelés
- Többszálúság, párhuzamosság támogatása

Rengeteg feladatot lát el helyettünk, megkönnyítve ezzel a munkát a fejlesztők számára. A Unity mindenki számára biztosít egy ingyenes verziót, ahol egy kezdő vagy középhaladó maximálisan kiélheti a kreativitását, fejlesztők számára pedig elérhető egy fizetős verzió, ez több eszközt tartalmaz a készítők számára, viszont azok inkább magasabb szintű projekteknél lehetnek hasznosak.

Rengeteg eszközt és lehetőséget biztosít és tartalmaz a Unity fejlesztő környezete, én a kétdimenziós eszközöket szeretném részletezni, illetve közülük is azokat, amelyeket használtam a projekt elkészítése során, így a továbbiakban erről lesz szó.

3.2. Unity kétdimenziós környezet

A legnagyobb különbség a kettő és a háromdimenziós fejlesztés között, hogy míg a háromdimenziós környezetben x,y és z tengelyen tudunk mozogni addig a kétdimenziósnál csak x és y tengelyen, bár a z tengely is jelen van, nincs különösebb funkciója. Az egyetlen dolog, amire a z tengely használható az nem más, mint hogy mélységet tudjunk adni a játékban, ez azt jelenti, hogy amikor a kamerán keresztül nézünk egy jelenetet a z tengelyen elfoglalt sorrendben fogjuk látni a dolgokat.

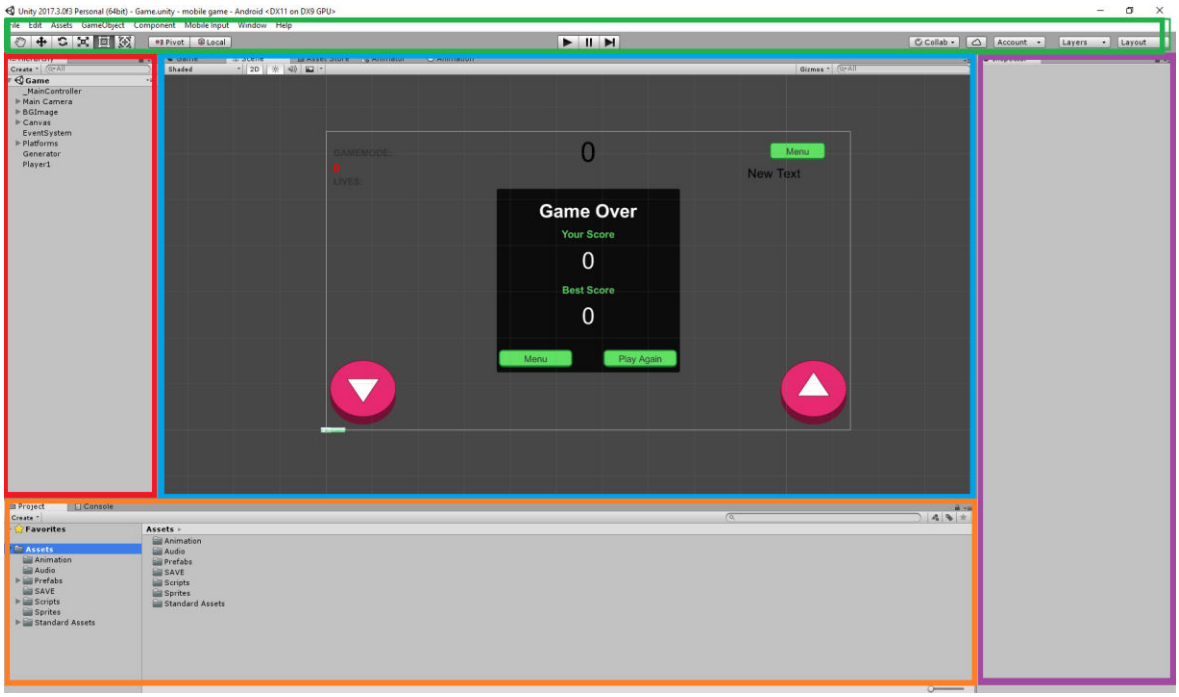
A másik fontos különbség hogy a kétdimenziós fejlesztés során nem modellekkel, hanem úgynevezett sprite-okat használunk az esetek túlnyomó részében. Természetesen vannak itt is kivételek, lehet használni háromdimenziós modelleket például a karakter számára, aminek köszönhetően sokkal részletesebben animált mozgást lehet létrehozni. Ennek köszönhetően sokkal látványosabb élményt nyújt a játékosok számára, mivel ha sprite-okat használunk, tulajdonképpen kétdimenziós bitmapeket látunk, vagyis képeket. Az animáció is ezek a képek alapján készülnek el, tehát sok kép egymás utáni gyors váltakozását látjuk, ami a mozgás illúzióját kelti. Itt rá is térhetünk a következő fontos különbségre.

Az animálásnál is máshogyan kell eljárni a háromdimenziós környezethez képest. Míg a háromdimenziósban egy úgynevezett csontvázra felhúzott modellt tudunk mozgatni görbék segítségével vagy motion szenzorok által rögzített mozgást adunk át a modelleknek, addig a kétdimenziósnál nincsen erre lehetőség. Két dimenzióban sprite-okkal kell megoldanunk ugyanezt. Az eljárás borzasztóan egyszerű, mint a régi vetítőgépeknél, ahol a gép a filmtekercsen lévő film képkockáit vetítette a vászonra egyesével, tette ezt olyan gyorsan, hogy a néző mozgóképet látott. Annyi a dolgunk, hogy a kívánt mozgást lerajzoljuk képkockáról képkockára. Minél több képet használunk egy mozgás leanimálásához, annál simábbnak látjuk a mozdulatokat, ebben az esetben nagyon aprólékosan kell a mozdulatról elkészíteni a képeket, szinte pixelről pixelre. Ha több képkockánk van mozdulatról, akkor sokkal simábbnak látjuk az összhatást, ennyire egyszerű.

Ezek az alapvető különbségek a kettő, illetve háromdimenziós környezet között. A továbbiakban a fejlesztői környezetet fogom bemutatni, ahol a fentebb említett eszközöket és funkciókat mutatom majd be. Szeretném ábrákkal is megmutatni, a környezet főbb részeit, hogy mi hol helyezkedik el.

3.3. A Unity editor bemutatása

A játék elkészítéséhez a Unity Editor ingyenes verzióját használtam, ami bárki számára elérhető a Unity hivatalos weboldalán. Letöltés és telepítés után létre lehet hozni projektet, itt meg kell adni, hogy kettő vagy háromdimenziós projektet szeretnénk csinálni, illetve hogy hová hozza létre a projektet. Miután ez megtörténik maga a szerkesztő program kerül elénk, ahol be is szeretném mutatni a főbb részeket.



1. ábra. Unity editor részei

Az 1. ábrán zöld mezővel jelölt helyen felül található a menüsor, itt alapvető dolgok találhatók, a file-ban például új jelenet létrehozása, mentése vagy új projekt létrehozása lehetséges és itt tudjuk build-elni a projektet valamilyen eszközre. Néhány fontosabb menüpont, mint a GameObject, ahol különféle kettő vagy háromdimenziós objektumokat lehet létrehozni, továbbá hangokat, grafikus felhasználói felületet, fényeket és effekteket is akár. A window menüpont alatt meg tudunk nyitni olyan ablakokat vagy füleket, mint például az animátor vagy az Asset Store. Fontos itt megemlíteni, hogy ezek az ablakok a kék kereten belül fognak majd megjelenni, de át lehet rendezni tetszés szerint, ez az elrendezés az alapértelmezett. A menüsor alatt helyezkednek még el ikonok, amelyekkel mozgatni, forgatni vagy nagyítani tudjuk a jelenetben elhelyezett objektumokat. A középen elhelyezkedő ikonok a futtatásra, illetve szüneteltetés és léptetésre szolgálnak. Jobb oldalon pedig a felhőszolgáltatás, a saját felhasználói fiókunk és az általunk használt réteg és elrendezések tekinthetők meg.

A kék mezőben látható a jelenetünk, ha a window menüpontban kiválasztunk egy új ablakot, az alapértelmezetten itt fog megnyílni, de akár el is tudjuk felezni ezt a területet, teljesen személyre tudjuk szabni mi és hol helyezkedjen el. Kezdésnél a jelenet, illetve a játék fül jelenik meg, a jelenet fül a szerkesztőnk tulajdonképpen, itt tudjuk elrendezni a komponenseket, amit majd a játékban szeretnénk használni. A jelenetben alapértelmezetten szerepel egy kamera is, ez a kamera az, amin keresztül a játékos szemléli a jelenetet, így amikor a játék fülre váltunk, akkor azt a képet kapjuk, amit a kamera lát éppen a

jelenetben. Az Asset Store-ról is szeretnék néhány szót ejteni itt, ez a Unity boltja tulajdonképpen, hasonlít a Google Play-hez, csak itt nem alkalmazások vagy játékok szerepelnek a boltban, hanem a játék elkészítéséhez szükséges modellek, hangok, scriptek, kész mozgások és még rengeteg minden más. Ide mi is tölthetünk fel saját tartalmat és tölthetünk le ingyenesen is, de vannak fizetős tartalmak természetesen.

A piros mezőben a jelenetben használt objektumok hierarchiáját láthatjuk. Minden, amit a jelenetben felhasználunk, az itt megjelenik, és ki tudunk alakítani egy alá-fölérendeltséget. Ha kiválasztjuk ezeket az objektumokat, akkor a lilamezőben meg fog jelenni különféle beállítási lehetőségek ezekhez az objektumokhoz.

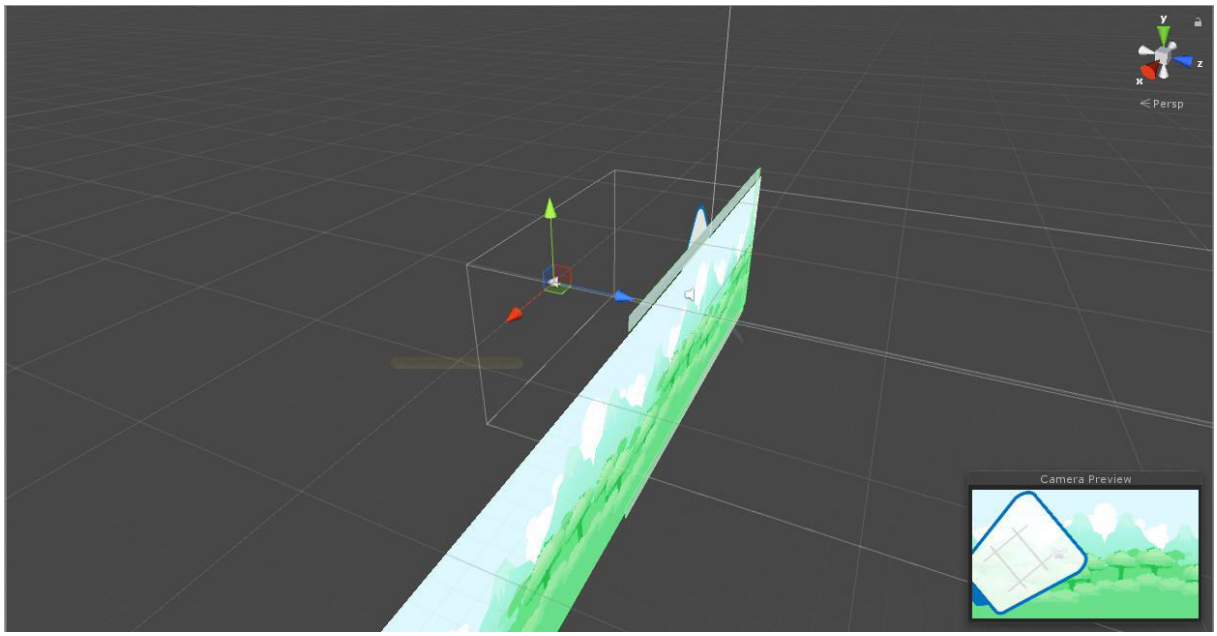
A lila mezőben találhatók a hierarchiából kiválasztott objektumokhoz tartozó komponensek és beállítások. Alapértelmezetten minden objektum rendelkezik egy Transform komponenssel, ez a pozícióját, forgatását és a skálázását mutatja. Általában ezzel a komponenssel minden objektum rendelkezik, de itt van lehetőségünk további komponensekkel felruházni az objektumokat, min például scriptekkel, Collider-ekkel, hangforrásokkal és sok más egyéb szükséges komponenssel.

A narancssárga mezőben található assets-ek hierarchiája, a projekthez felhasznált összes képet, audió fájlt, scriptet vagy bármilyen egyéb a projekthez használni kívánt fájlokat itt tárolja a Unity. Innen tudjuk a jelenetbe behúzni és felhasználni őket. Ezeket a fájlokat a Unity-ben assets-eknek nevezzük, de attól hogy itt vannak nem jelenti azt, hogy fel is kell használni őket, csak itt tároljuk. Ezen felül még itt található a konzol, ahová a hibaüzenetek érkeznek, ha belefutunk valamilyen hibába, de itt ki is tudunk íratni akár értékeket hibakeresés céljából.

3.4. A felhasznált eszközök bemutatása

3.4.1. Kamera

Minden jelenetnek kell rendelkezni és rendelkezik is egy kamerával, mivel a kamera lesz a játékos szeme. Viszont a kamera önmagában csak egy képet közvetít, meg kell adni neki, hogy milyen célpontot kövessen, ezt scripttel lehet megoldani a legjobban. Ha scriptteléssel valósítjuk ezt meg, akkor van lehetőségünk további finombeállításokra, mint például mennyire szorosan kövesse a célpontot, vagy esetleg finomabban mozduljon be a kamera, ezekre mind-mind találunk megoldást.

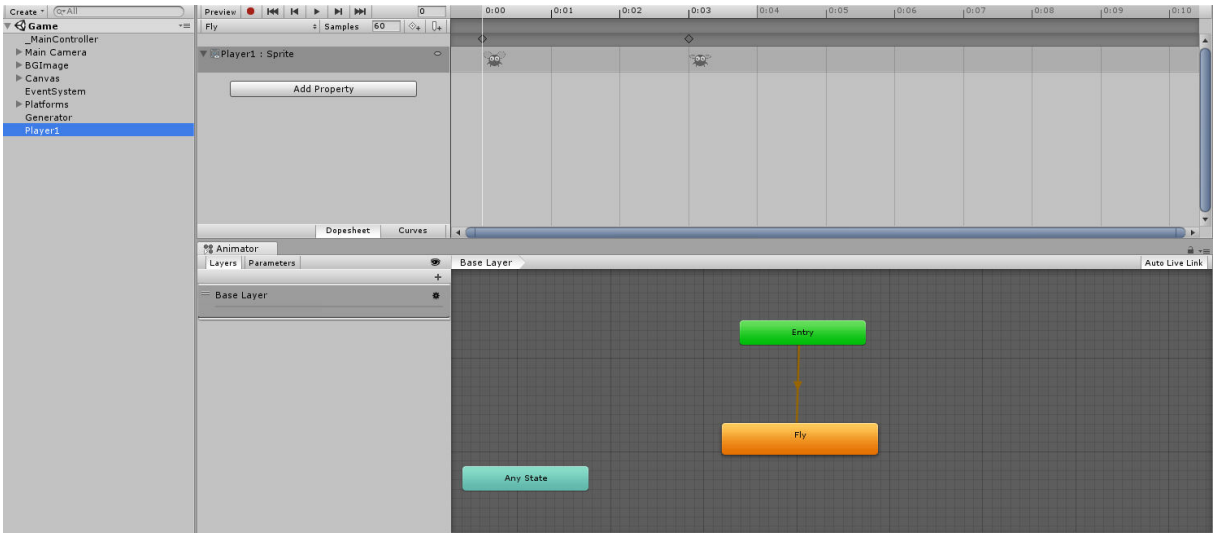


2. ábra. Kamera háromdimenziós nézetben

A 2. ábrán egy kétdimenziós jelenetet látunk háromdimenziós nézetből, itt jól megfigyelhető a kamerának a látószöge, amit fehér négyzet jelöl számunka. A jobb alsó sarokban pedig a kamera aktuális képét. Továbbá itt az is látszódik, hogy a kétdimenziós térben is fontos szerepe van a z tengelynek, ugyanis minél nagyobb az értéke egy objektumnak a z tengelyen, annál közelebb van a kamerához. Ezzel a módszerrel tudunk objektumokat egymás elé és mögé rendezni.

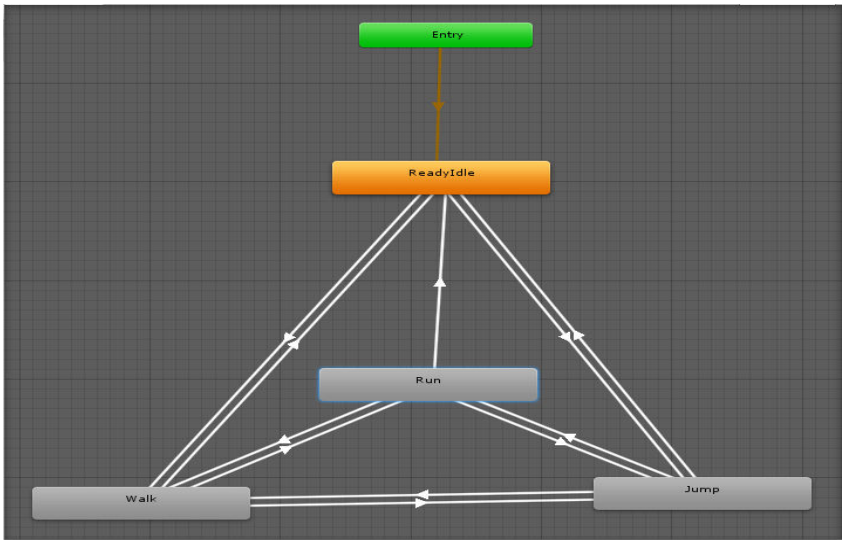
3.4.2. Animátor és animáció

Fentebb már beszéltem, hogyan is kell elképzelni a kétdimenziós animálást, most részletesebben is szeretném bemutatni ezt. Amire szükségünk van egy kétdimenziós animáció elkészítése, az nem más, mint a kívánt mozgásról elkészített sprite-ok, amelyek a mozgás különböző szakaszait mutatják be. Ha ezekkel rendelkezünk, akkor be is húzhatjuk az assets-ek közé egy külön mappába, fontos hogy mindent külön tároljunk, így később sokkal átláthatóbb lesz, mikor már rengeteg elemmel dolgozunk. Ha behúztuk őket, akkor létrehozunk egy játékobjektumot, amikor kijelöljük ezt a hierarchiában és megnyitjuk az animáció ablakot, akkor lehetőségünk adódik létrehozni egy animációt. Amikor ezt megtesszük két fájl fog létrejönni az assets könyvtárban, az egyik az animációhoz tartozó kontroller, amit az animátor ablakban tudunk megnézni, a másik pedig maga az animáció lesz.



3. ábra. Animátor és animációs ablak

A 3. ábrán a kész játékban található egyetlen egyszerű animációval szeretném szemléltetni ennek a menetét. Először az animációt készítsük el, ehhez nincs más dolgunk, mint hozzáadni az animáció ablakban a mozgáshoz szükséges sprite-okat, ezek után pedig meg tudjuk adni, hogy milyen gyorsan váltakozzanak ezek a képen. Ezt láthatjuk a képen is, hogy 3 tized másodpercenként váltakozik a két kép. A létrehozott animációhoz tartozik egy kontroller, is ezt az animátor ablakban láthatjuk, itt minden egyes animáció egy állapotot jelez számunkra, amikor elindul a játék, tehát belépünk (entry) akkor az egyetlen állapot, amibe át fogunk menni az a repülés animáció. Több animációt, azaz több állapotot tudunk létrehozni és az állapotok között átmenetet tudunk készíteni. Itt a képen egyetlen állapotba lépünk át és abból nem is fogunk kilépni többet.



4. ábra. Animáció kontroller

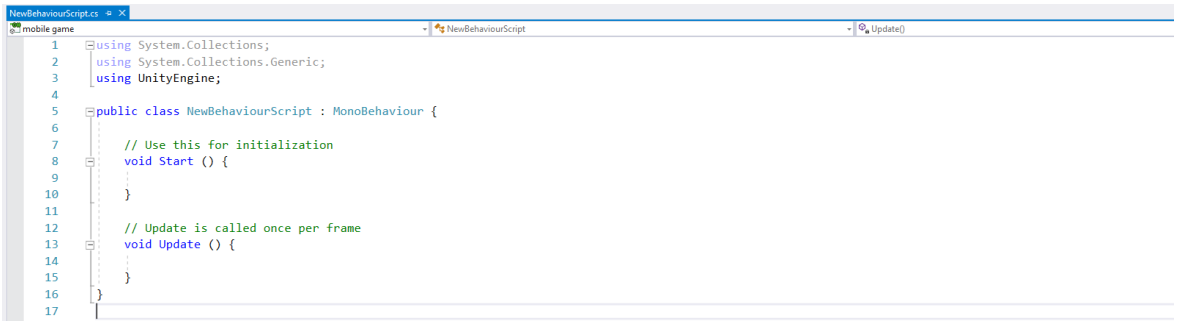
A 4. ábrán szeretném bemutatni, hogy mi a helyzet, amikor több állapotunk van. Ezen a képen egy háromdimenziós karakterhez tartozó teljes mozgást leíró kontroller látható,

amelyben több mozgás animáció található. Mint látható, amikor belépünk a játékba, akkor az első állapot egy idle, azaz várakozó állapot lesz, ez az alapmozgás, ebből ágazik tovább az összes többi mozgás. Az oda-vissza nyilak azt jelzik, hogy kölcsönösen tudunk ugrálni két állapot között. Az ábrán ezt nagyon jól is lehet látni, hogy ha el akarunk kezdeni futni, akkor előbb sétálunk, majd utána tudunk futni, viszont álló helyzetből nem tudunk elkezdeni rohanni. Az is látható még, hogy az ugrást minden mozgás közben tudjuk végezni, ami reális is, mivel futás közben is lehetséges ugrani egyet. Ezek között az állapotok között úgy tudunk ugrálni, hogy paramétereket adunk meg, ebben az esetben a sebesség paraméterére tudunk hivatkozni, ami gombnyomásra növekszik, így megadhatjuk, hogy mely sebességtartományokban, mely animáció érvényesüljön, de ugrásnál szimplán egy gomb lenyomása is megadható paraméternek, amely elő fogja idézni az ugrás animációt. Lehet több rétegű animációkat is létrehozni, például rendelkezünk egy karakterrel, és az ahhoz tartozó sprite-ot szétbontom, majd külön meganimálom, például a kézmozgást vagy a fejmozgást vagy bármi mást is lehet külön kezelni, ha szükséges, így több animáció fog egy időben lefutni. Lehetőség van animációk letükrözésére, ez akkor is hasznos, ha például egy kétdimenziós játékban a karakter mozgása csak a jobb irányba van megcsinálva, ebben az esetben csak le kell tükrözni a jobbra mozgást és máris kész a balra mozgás is. Így rengeteg időt lehet megtakarítani, mert nem kel külön elkészíteni a másik irányú mozgásra is a sprite-okat.

3.4.3. Scriptek

A script rövid programot vagy utasítássort jelent, amely valamilyen részfeladat automatizálására szolgál. Unity-ben szintén lehetőségünk van ilyen scriptek írására, hogy azzal valamilyen funkciót hozzunk létre a játék részére. Ezt két féle nyelvvel tudjuk kivitelezni, ezek pedig a C# és a JavaScript. A Unity rendelkezésünkre bocsájt egy részletes dokumentációt, amit scripting API-nak[11] neveznek, ez mindenki számára elérhető és tartalmaz minden tudnivalót arról, hogy hogyan is kell használni a beépített függvényeket és változókat. Mivel a kész játékhoz tartozó összes scriptet C# nyelvben írtam meg, ezért ezt a részét szeretném jobban kiemelni és részletezni. Az viszont nincsen megkötve, hogyha már írtunk egy scriptet C# nyelven, akkor más scripteket ne lehetne megírni akár JavaScript-ben. Természetesen fontos megállapodni önmagunkkal, hogy milyen nyelven is szeretnénk fejleszteni, mivel ez később problémákat okozhat, de a Unity nem fog ezért szólni vagy figyelmeztetni. A szerkesztőn belül tudunk új scripteket létrehozni, ezt érdemes az assets mappában belül egy új mappában külön kezelni, ha

létrehozunk scriptet, akkor természetesen megmondjuk neki, hogy C# vagy JavaScript scirptet hozzon létre. Miután létrehoztuk, meg is tudjuk nyitni, azt hogy milyen szerkesztővel szeretnénk ezt megtenni, szintén ki tudjuk választani a beállításokban. Én a Microsoft Visual Studio 2017-ben[12] dolgoztam, de bármilyen más szerkesztőt hozzá tudunk adni. Amikor létrehozunk egy új C# scriptet, akkor nem egy üres script fájlt kapunk, hanem pár alapvető függvény és névtér automatikusan bele van írva.

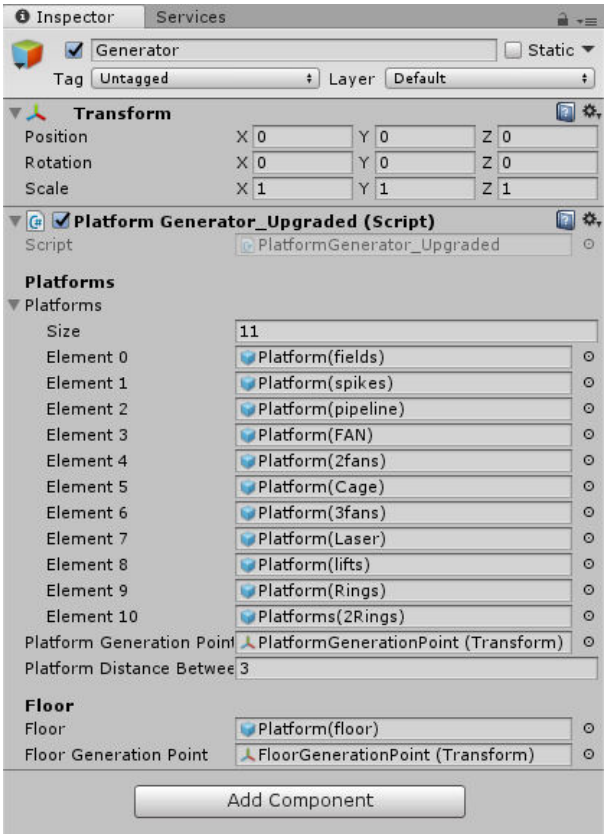


5. ábra. C# script Microsoft Visual Studio 2017-ben

Az 5. ábrán látható, mit is kapunk egy teljesen új C# script fájlban. Az első három sorban az alapvető névtereket láthatjuk, természetesen ezeket tovább bővíthetjük. Utána található a főosztályunk, amely a script nevét visel, majd hogy ezt honnan származtatjuk. Minden osztálynak a *MonoBehaviour* a szülőosztálya, mindent ebből származtatunk Unity-ben, de ez csak a C# scriptek esetén igaz, mert JavaScript esetében nem kell ebből származtatni az osztályokat. A főosztályon belül láthatunk még két darab függvényt, amelyekből a *Start* az inicializálásért felelős, ez csak egyszer fog meghívódni, illetve az *Update* függvény, amely folyamatosan meghívásra kerül, minden egyes képfrissülésnél. Létezik több függvény is inicializálásra és frissítésre egyaránt, inicializálásra ott van még az *Awake* is. Közte és a *Start* között annyi a különbség, hogy az *Awake* még a *Start* előtt le fog futni, tehát ha a script példánya már betöltődött, míg a *Start* lesz az első függvény, ami lefut az *Update* metódusok előtt. A frissítő függvények között is van *LateUpadate* és *FixedUpdate*, az előbbi az *Update* után fog meghívódni, ezt parancsvégrehajtó utasítások során érdemes használni, például a kamera mozgását, ami követ egy objektumot, mivel az objektum mozgása úgyis az *Update*-ben fog megtörténni. A *FixedUpdate* minden stabil képkocka frissítésnél fog meghívódni, ez ütközéseknél fontos, vagy ha valamilyen erőhatás történik két objektum között.

Ha változókat szeretnék létrehozni, azt közvetlen a főosztály alatt tudjuk megtenni. A változók terén nincs különbség, ugyanúgy van egész, tört, logikai, konstans és karakter típusú változó, annyi különbséggel, ha *float* típust használunk, akkor egy F betűt kell a tört szám mögé írnunk. Ha nem csak definiáljuk, hanem deklaráljuk is a változókat, akkor

lehetőségünk adódik szerkesztőn belül ezeket az értékeket megváltoztatni, így nem kell a programkódban átírnunk az értéket, hanem kényelmesebben a szerkesztőből megtehetjük. Ha csak definiáljuk a változókat, akkor nem tudjuk a szerkesztőben megtenni ezt, ha deklarálni szeretnénk változókat, akkor azt az *Awake* függvényen belül érdemes megtenni. Létrehozhatunk *GameObject*, *AudioSource*, *Collider* és szinte a játéktéren használt objektum minden komponensére tudunk típussal hivatkozni. Ezek a Unity saját típusai, ha ilyen típusokat használunk, általában definiálni kell őket, majd a szerkesztőn belül megjelenő helyes mezőbe behúzni a kívánt objektumot vagy annak komponensét.



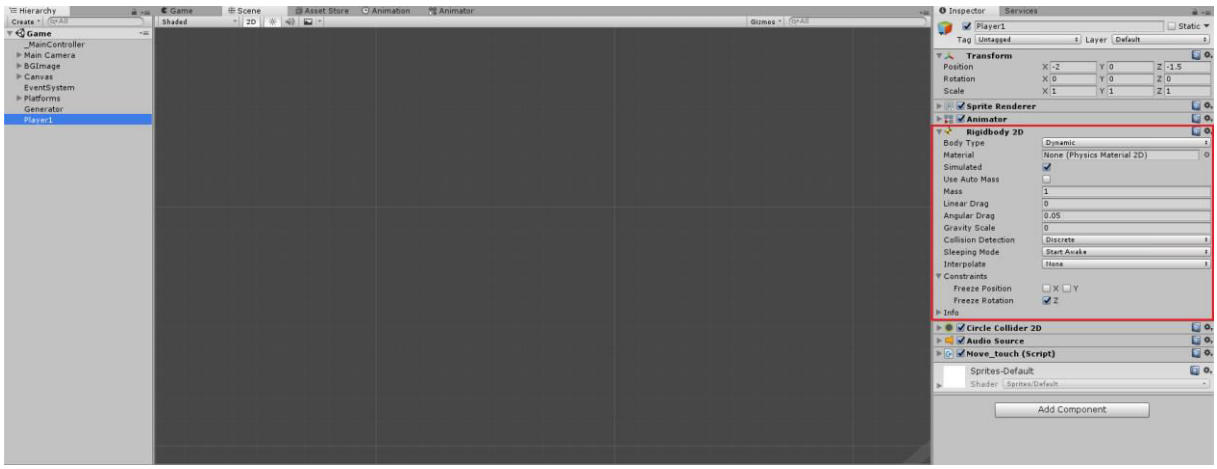
6. ábra. Gamobject-hez hozzáadott C# script Unity Editor-ban

A 6. ábrán megfigyelhető, hogy vannak olyan egész típusú változók, amelyeket itt lehet változtatgatni, illetve vannak *GameObject* és *Transform* típusú változóim is, amelyeket nem deklaráltam csak definiáltam, így lehetőség van hierarchiából behúzni ezeket, úgynevezett drag and drop módszerrel. Ez is a Unity sajátosságai közé tartozik, nagyon egyszerűen lehet hivatkozni így különböző elemekre, objektumokra. Ha véletlenül elrontunk valamit egy scriptben, vagy valamit nem deklarálunk, akkor nem tudjuk lefuttatni a programot, de ilyenkor figyelmeztetést is kapunk, amit a konzolon látunk. Legtöbbször konkrétan meg tudja mondani melyik sorban, milyen probléma adódott, így a hibakeresést is megkönnyíti számunka a Unity.

A kódolást tekintve azt is lehet mondani, hogy C# vagy JavaScript ismeretekkel rendelkezve nagyon jól el lehet boldogulni és csak a különböző típusokat kell megszokni, ami az újdonságot nyújtja. Ugyanúgy a főosztály alatt hozunk létre változókat, osztályokat, a függvényeket is itt írjuk meg vagy akár az inicializálás és a frissítés alatt is megírhatjuk őket, majd a megfelelő függvényen belül meghívjuk őket. A továbbiakban a típusok bemutatásával szeretnék foglalkozni és kitérni a legfontosabb dolgokra, illetve amelyeket én is felhasználtam a játék készítése során.

3.4.4. Rigidbody, Colliderek

A Rigidbody-t nem tudom hogyan is lehetne lefordítani, de ez egy olyan komponenst takar, amelyet ha hozzáadjuk egy objektumhoz, akkor arra az objektumra érvényesülni fognak a fizika törvényei. Természetesen olyan mértékben, amennyire a játékmotor engedi, illetve a beállításaink függvényében. Amikor alkalmazzuk ezt a komponenst, akkor az adott objektumra hatni fog a gravitáció, illetve, ütközés vagy valamilyen egyéb érintkezés hatására elmozdul vagy kitér az eredeti pozíciójából. Ezt nem nekünk kell scriptben megírnunk, hanem ezt biztosítja számunka játékmotor, nekünk csak annyi dolgunk van, hogy a rendelkezésre álló beállításokkal finomítjuk azt.

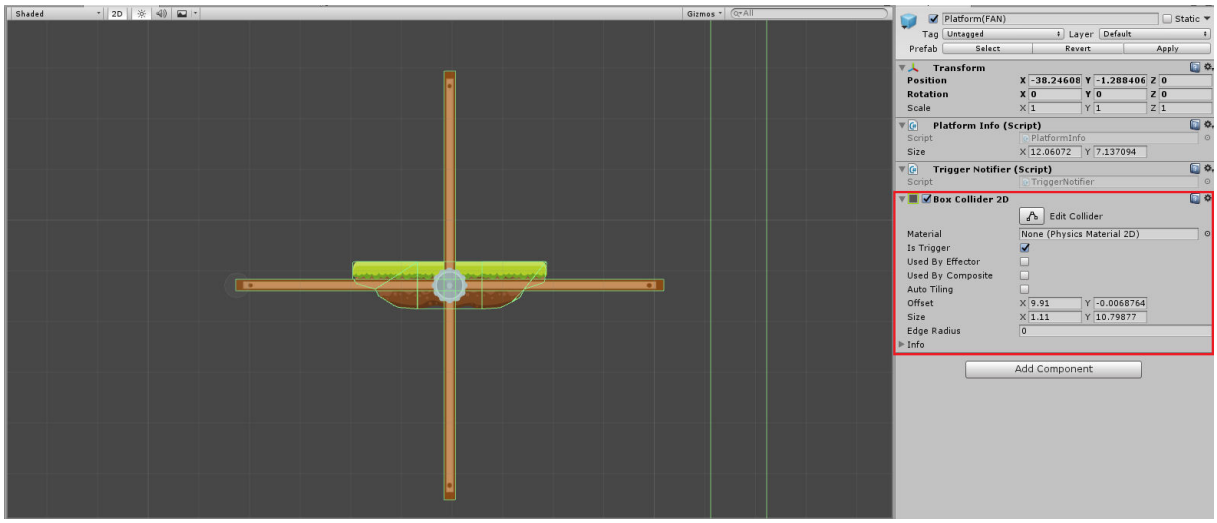


7. ábra. Rigidbody komponens Unity Editor-ban

A 7. ábrán látható hogy a Player1 nevű komponenshez hozzá van rendelve egy Rigidbody, ezt a piros keretben láthatjuk. Itt tudunk a rendelkezésre álló beállításokban megadni olyan értékeket, amelyek megfelelőek számunka a játékban. Be tudjuk állítani, hogy milyen erővel húzza az objektumot a gravitáció vagy az adott objektum tömegét, például ha ütközés esetén egy kisebb tömegű objektum ütközik egy nagyobb tömegűvel, akkor nehezen vagy el se tudja tolni. Ha ez fordítva történik meg, akkor a nagyobb tömegű fogja eltolni a könnyebb objektumokat. Hozzá tudunk adni úgynevezett fizikai anyagot az objektumhoz, ezzel különféle anyagok viselkedését tudjuk szimulálni, ha például valami

rugalmas anyagot hozunk létre, akkor az ütközésnél pattogni fog, de lehet teljesen rugalmatlan anyagot is hozzáadni az objektumokhoz. Továbbá egy fontos beállítás, ennek inkább kétdimenziós környezetben van értelme, ha a spirte-oknak adunk gravitációt és például azok leesnek a magasból, ezt úgy kell elképzelni, mintha egy papírlapot ejtenénk le, nem fog függőlegesen leesni az élére. Ezért meghatározhatjuk, hogy csak x és y tengelyen érvényesüljön a fizika a játékban, ebben az esetben nem fog a kétdimenziós objektum a z tengely felé elfordulni, azaz mindig a kamera felé fog nézni. Persze nincs értelme fizikáról beszélni, ha nincsen semmilyen interakció az objektumok között, és ha szimplán csak Rigidbody-t használnánk, akkor minden objektum csak leesne vagy átesne egymáson, ilyenkor kell használnunk a Collider-eket.

A Collider vagy más néven ütköző, a nevéből eredően az ütközések kezelésére szolgál. Ez egy olyan komponens, amelyet ha hozzáadunk az objektumunkhoz, akkor az körbeveszi az adott testet. Egy objektum több Collider-rel is rendelkezhet, a lényeg hogy minél jobban le tudjuk fedni vele az alakzatot. Több féle formájú is létezik, ezeket felhasználva kell lefednünk az objektumot, persze van olyan is, amely automatikusan felismeri és lefedi az alakzatokat, de ez nem biztos, hogy tökéletes lesz. Megkülönböztetünk kétdimenziós és háromdimenziós Colliderek-et, a kétdimenziós esetén sokkal könnyebb dolgunk van, sokkal egyszerűbb formákat kell lefednünk. Ha alkalmazzuk ezeket az ütközést detektáló komponenseket, akkor az objektumok nem fognak átesni egymáson, tehát a Rigidbody-nak csak Collider-rel együtt van értelme. Ebben az esetben tudunk ütköztetni egymással több objektumot, amely hatására eltolják egymást vagy elpattannak és elgurulnak.



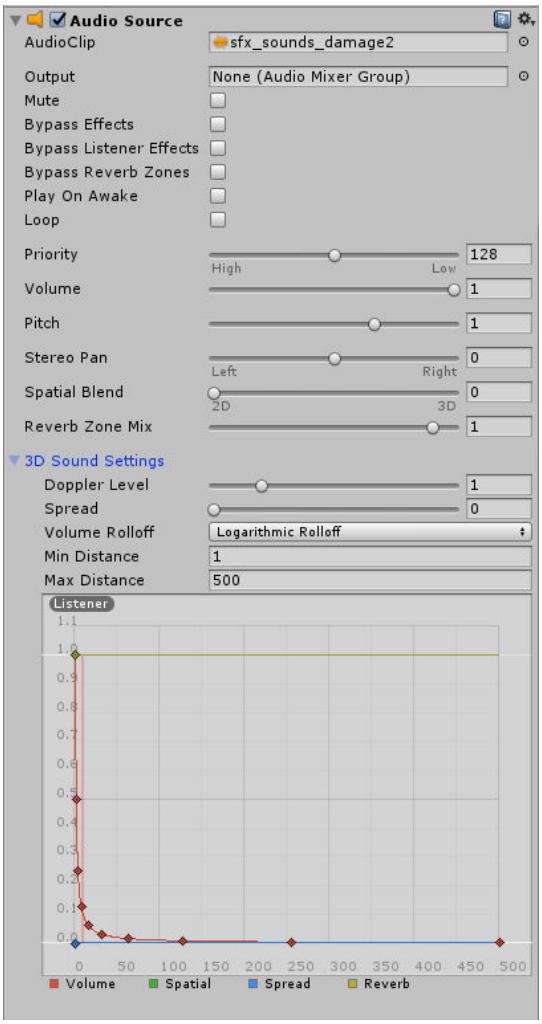
8. ábra. Collider komponens és alkalmazása Unity Editor-ban

A 8. ábrán egy kétdimenziós pályaelem látható, ez több különböző formájú elemből épül fel, amelyeken a zöld él jelzi a Collider-t. A forgó lapátokon egy úgynevezett Box

Collider található, míg a szigeten egy Polygon Collider van. Az utóbbi felveszi a sokszögű test formáját, míg az előbbi esetén egy négyszögletű alakzatot kapunk. Minden Collider-t tudunk szerkeszteni, a pirosan bekeretezett résznél látható egy Edit Collider nevezetű opció, ennek a segítségével tökéletesen méretre lehet igazítani azokat. A képen jól látszik még, hogy az alakzat után is van még egy Collider, amely semmilyen sprite-ot nem határol, ez azért van, mert az egy trigger, azaz valamilyen eseményt indít be, ha beleütközünk. A Collider beállításainál el lehet dönteni, hogy trigger-ként működjön vagy sem, ha előbbit választjuk, akkor nyugodtan átmehetünk rajta, nem ütközés fog történni, hanem egy eseményt indítunk el vele, például egy hangot vagy bármilyen interakciót. Ha rendelkezik az objektumunk Rigidbody-val és Collider-rel akkor felmerül a kérdés, hogyan tudjuk detektálni magát az ütközést, ezt scriptben tudjuk megoldani, létezik egy *OnCollisionEnter* nevű függvény, amely le fog futni, ha két vagy több objektum egymáshoz ér, a függvényen belül természetesen megadhatjuk, mi is történjen ebben az esetben.

3.4.5. Hangok

Többször is szó esett már a hangokról, most szeretném elmondani hogyan is lehet hangokat használni a Unity-ben. Amire szükségünk van az egy hangfájl, amit az assets mappán belül elhelyezünk. Ezután hozzáadunk egy Audio Source nevű komponenst ahhoz az objektumhoz, amelyet meg szeretnénk szólaltatni. Ezek után fogjuk és hozzáadjuk a kívánt hangfájlt, és ha elindítjuk a játékot meg is fog szólalni az objektum. Persze itt megint külön kell választani a háromdimenziós és a kétdimenziós játékteret, mivel háromdimenziós játék esetén lehetőség van arra, hogy térbeli hangzású legyen az adott hang. Ez azt jelenti, hogy hallani milyen irányból szólal meg a hang, vagy ha messzebb van, akkor halkabban szól és más egyéb beállításra is lehetőségünk adódik. Kétdimenziós környezetben ennek nincsen sok értelme, de van pár dolog, amire van lehetőség.



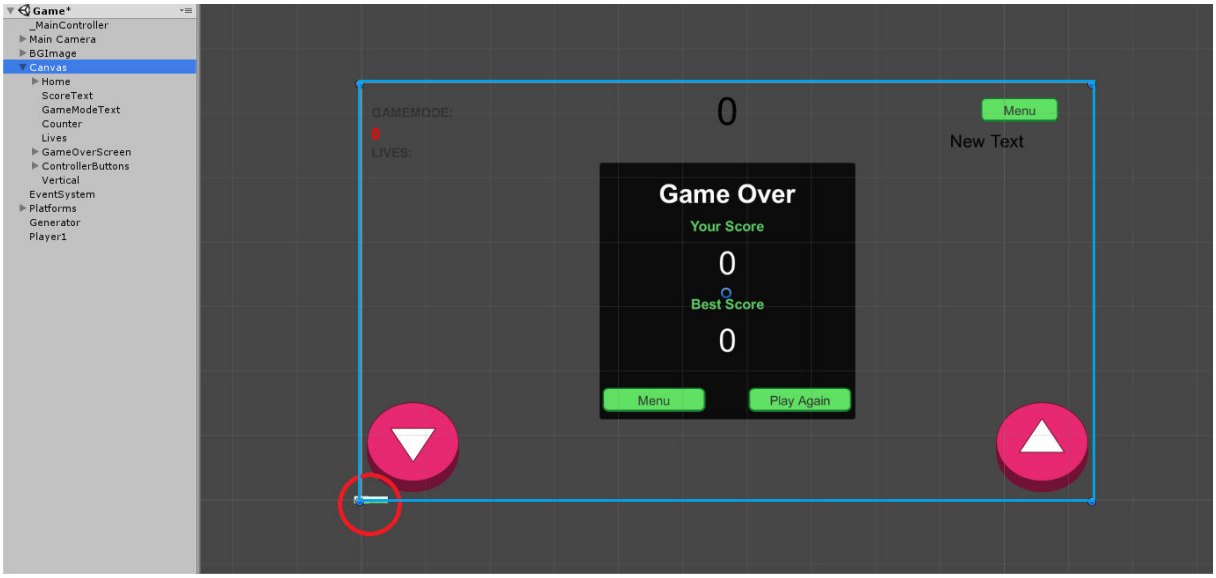
9. ábra. Audio Source komponens Unity Editor-ban

A 9. ábrán egy Audio Source komponens látható, az alsó háromdimenziós hangbeállításoktól most eltekintünk, de ami fölötte van az inkább érdekes a kétdimenziós játéktérben. Lehetőségünk van olyan hangokat hozzáadni, amelyek már az indítástól kezdve lejátszódnak, ilyen például valamilyen aláfestő zene, ezeket folyamatosan ismételtetjük, továbbá beállíthatjuk, milyen hangos legyen vagy milyen hangmagasságú, hol halljuk a hangot és még sok apróbb beállításra van lehetőségünk. Kérdés mi a helyzet, ha nem szeretnénk, hogy folyamatosan szóljon egy zene vagy valamilyen hang. Ilyenkor van értelme a fentebb említett trigger Collider-nek, amely érintésére lejátszhat egy hangfájlt nekünk. Ebben az esetben script-et kell írni, amelynek a lényege, hogy definiálunk egy AudioSource típust, majd ezt az OnCollisionEnter függvényben lejátsszuk, ennyire egyszerű.

3.4.6. Grafikus felhasználói felület

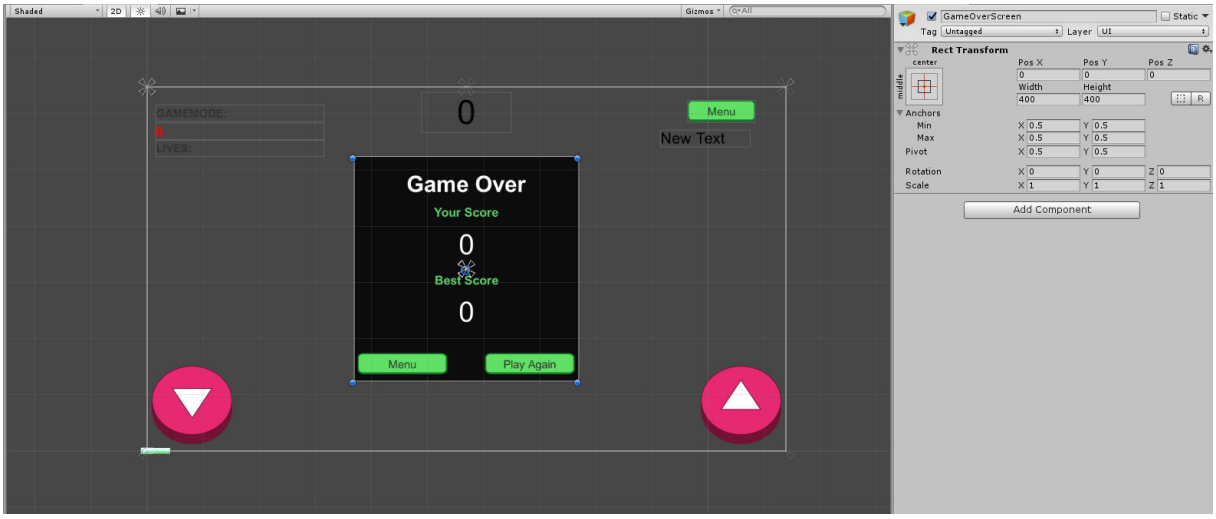
Menük, HUD-ok és egyéb felhasználó felületek elkészítésére a Unity szintén kiszolgálja a fejlesztőket, ugyanis rendelkezik saját beépített GUI-val. A hierarchiában

létre kell hozni egy UI elemet, azon belül is Canvas-t, vagyis egy vásznat, amin majd elhelyezzük az elemeket. Meg kell határoznunk, hogy milyen méretű vásznat szeretnénk használni, ezt a Canvas beállításainál tudjuk elérni. Majd ezután a Canvas-on belül a hierarchiában létrehozhatunk szöveget, gombot, képet, checkboxot és rengeteg más UI elemet. Ezeket el tudjuk helyezni a vásznon a tetszőleges pozícióra, de szeretném ezt bemutatni kicsit pontosabban, mivel egy fokkal nehezebb annál, mint aminek tűnik.



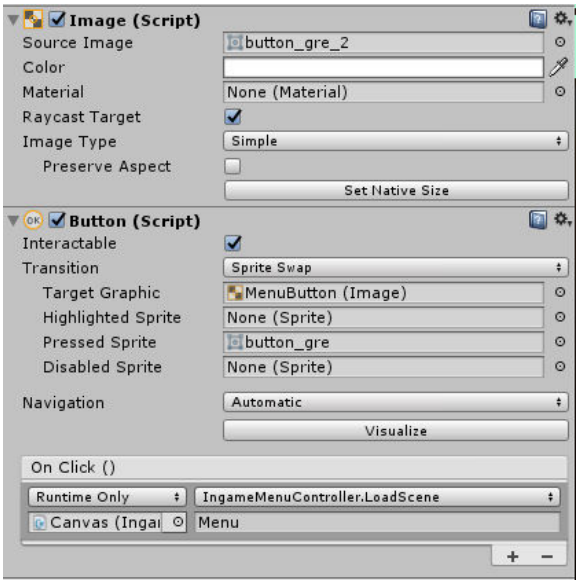
10. ábra. Canvas elhelyezkedése a jelenetben

A 10. ábrán, ha felhasználó felületet hozunk létre, akkor kapunk egy külön felületet, a jelenetben, ahol csak a vásznat látjuk, és a UI elemeket. A pirossal bekarikázott részen található a kamera és az elé elrendezett játékobjektumok, a kékkel bekeretezett rész pedig maga a Canvas. Az ide beszúrt elemeket pedig szépen elrendezhetjük, megadhatjuk, hogy a vászon széleitől milyen messzire legyenek, és ha változik a felbontás, akkor mihez igazodjanak.



11. ábra. UI elemek pozícionálása

11. ábrán láthatjuk, hogy a középső UI elem, a játék végét jelző képernyő az a vászon közepéhez fog igazodni, a 0-s pozícióban, minden tengelyen, továbbá rendelkezik egy fix mérettel. Ezen tudunk változtatni, megadhatjuk, hogy a vászon széleihez vagy a sarkához illeszkedjenek. Ezt az igazítást minden elemnél meg tudjuk tenni. A gomboknál lehetőségünk adódik képeket vagy sprite-okat használni, amikor létrehozunk egy gombot és elhelyeztük, megadhatunk neki egy más stílust.



12. ábra. A gomb UI elem

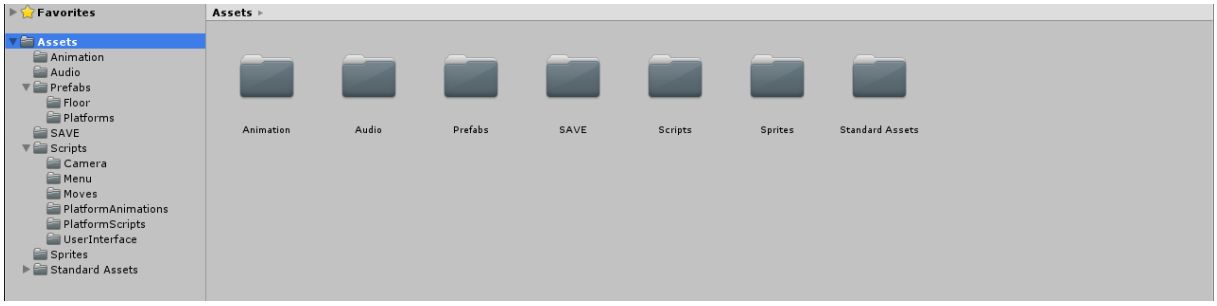
Ennek a menete itt látható, a képforrásnál be van illesztve egy sprite, majd a gomb scriptnél kiválaszthatjuk, milyen módszerrel fogjuk érzékeltetni a lenyomást, itt a spirte cserével van megoldva. Lenyomásra egy másállapotú gomb fog megjelenni a vásznon és a klikk eseményre pedig meg tudjuk mondani, mi történjen. UI elemeknél leginkább a reszponzív dizájn létrehozása okoz nehézséget, illetve az eltérő felbontású képernyőkre való optimalizálás.

A továbbiakban azzal folytatom, hogy pontosan leírom, hogyan használtam fel ezeket az eszközöket, arra, hogy elkészüljenek a kitűzött feladatok.

4. Alkalmazás megtervezése, grafikák bemutatása

4.1. Tervezés

Az alkalmazás megtervezésénél, semmilyen modellt nem követtem, talán az evolúciós modellhez állt a legközelebb a fejlesztés menete. Egy kisebb prototípust hoztam létre, majd egyre több funkcióval láttam el és egyre jobban tökéletesítettem. Természetesen itt is fent állt annak veszélye, hogy ha nagyobb változtatást szeretnék beiktatni, akkor visszamenőleg sok mindenen változtatni kell, de szerencsére ilyen nem fordult elő. Miután elkészültem a prototípussal, ami az alapötletet testesítette meg, onnantól kezdve gyorsan világossá vált, hogy milyen irányba szeretnék elindulni. Ekkor jött az infinite runner ötlet és a műfajhoz tartozó sajátosságok, amelyek gyorsan kitűzték előttem a végső célt. A projekt szépen apránként haladt, mivel rendelkeztem a kitűzött feladatokkal és célokkal, amiket fentebb már ecseteltem. Ezeknek a feladatoknak a megoldására rengeteg segítséget találtam, mivel infinite runner játék már ezelőtt is készült, úgyhogy az alapokat el tudtam lesni. De ez inkább már megvalósításhoz tartozik, főként a scriptíráshoz, az igazi tervezést és fejtörést inkább a látvány és a pálya elkészítése követelte meg. Természetesen át kellett gondolni, hogy a script fájlokból mennyire van szükség, illetve mennyire akarom szétszedni külön script fájlba a funkciókat. Azt az elvet követtem, hogy minden fontosabb funkció kerüljön külön script fájlba, de nem akartam egy-egy függvénynek külön scriptet, ezért csoportosítottam őket. Tehát a mozgáshoz tartozó összes kód egy scripten belül található vagy azok a függvények, amelyek a GUI-t kezeli egy scripten belülre került és így tovább.



13. ábra. Assets hierarchia

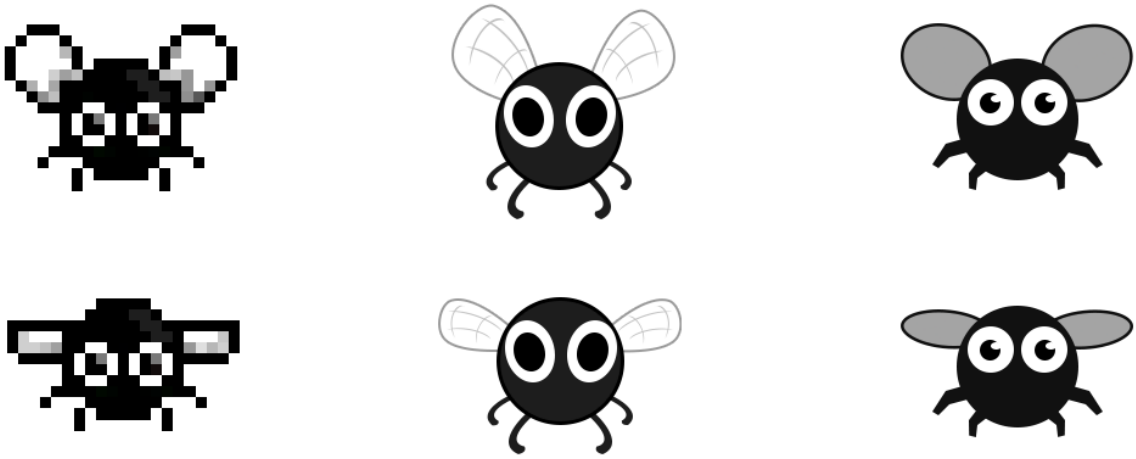
A 13. ábrán látható az assets mappa hierarchiája, itt találunk meg minden olyan fájlt, amelyre szükségünk volt a projekt elkészítése során. A mappák elnevezése egyértelmű, így gyorsan lehet tájékozódni, egyedül a Standard Assets, amit nem én hoztam létre, ez egy beimportált mappa, amelyben különféle eszközök találhatók az érintő felületes kezelésre. A hosszabb scriptekben igyekeztem egyértelmű változókat használni és elkülöníteni a

privát és a publikus változókat. Tovább van lehetőség arra, hogy a definiálásnál címet adjunk, mely változókat mire használunk. Ez a cím csak a szerkesztőn belül fog megjelenni, de kódon belül is igyekeztem a nehezebb funkcióknál kommentezéssel emlékeztetni magamat. De ezek a lépések nem igényeltek túl nagy tervezést, látható, hogy nem egy óriás projektről van szó, a látvány, elrendezés, GUI, pályaelemek és karakter kinézetének eltalálása sokkal időigényesebb folyamat volt.

4.2. Grafikus elemek bemutatása

4.2.1. A karakter

Már említettem, hogy kétdimenziós környezetben sprite-okat szokás használni, itt viszont fontos megemlíteni, hogy rengeteg típus közül lehet válogatni. Ezek fogják a játék hangulatát megadni, illetve ezt fogja a játékos látni, úgyhogy nagyon fontosnak tartottam, hogy egy minőségi látvány fogadja a játékost. Két-féle irányban gondolkodtam az első a pixel art, amely nagyon népszerű és egy retro kinézetet tud kölcsönözni a játéknak, ami kellemes nosztalgia hangulattal szolgálhat egyes felhasználók számára. A második irány természetesen a nagyobb felbontású részletesen kidolgozott sprite-ok voltak, ezt az irányt választottam, mivel sokkal könnyebb volt elkészíteni később saját elemeket. Az első sprite amit használtam, az a karakter sprite-ja volt, a jellegzetes cikázó mozgás után döntöttem a légy mellett. Sikerült találnom egy pixel art-os legyet, amely nagyon jól megfelelt kiindulási pontnak. Mivel általában valamilyen licence tartozik ezekhez a képekhez és sokkal kidolgozottabb karaktert szerettem volna, ezért később lecseréltem ezt.



14. ábra. A karakter fejlődése

A 14. ábrán szeretném ezt megmutatni, két képet látunk mindegyik verzióból, mivel animálásnál a repülést ezzel a két állapottal lehet elérni. Az első pixel art-os[13] verzió

volt a kiinduló pont, majd jött a második verzió, amely már egy saját sprite volt, de túlságosan hosszúkásra sikerült az előző verzióhoz képest, ezért a mozgás is sokkal furcsább volt látványra is és játékélmény szempontjából is. Ezen tapasztalatok után jött a harmadik végleges verzió, ami szintén egy saját készítésű elem és méreteiben sokkal közelebb áll az első verzióhoz. A második verziónál törekedtem arra, hogy minél részletesebb legyen, de a pályaelemek mellett nem úgy festett, mint azt vártam, így inkább az egyszerű és letisztult dizájnnál maradtam, ezt tükrözi is a harmadik végleges kép. A karakter végleges formája bár később került bele a játékba, a pálya és a pályaelemek elkészítésénél is próbáltam a letisztult stílust alkalmazni.

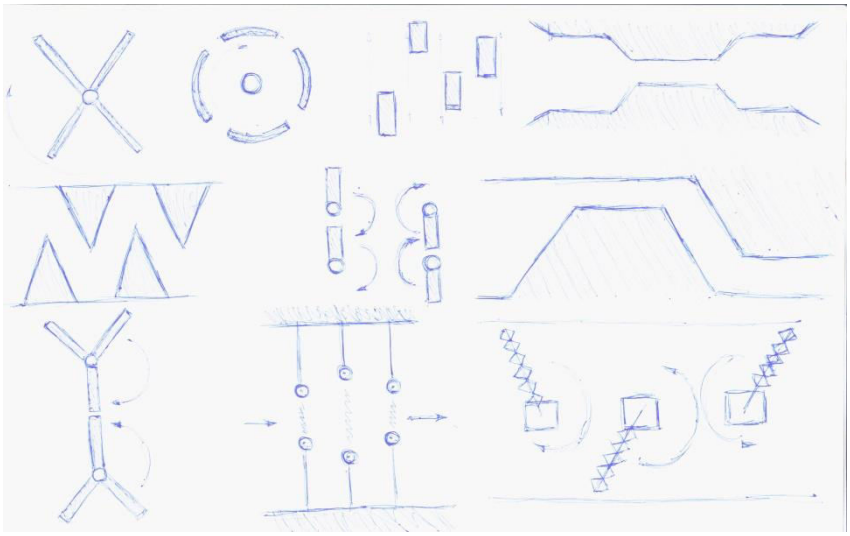
4.2.2. A pálya

A pálya kigondolása és az elemek elkészítése okozta a legnagyobb fejtörést, illetve ennek elkészítése. A tervezés folyamatába szeretnék betekintést adni, hogy mennyire kezdetleges és apró lépésekkel indult az egész és jutott el a fejlesztés a végéhez. Az interneten rengeteg sprite csomag tölthető le ingyenesen, amelyek tartalmaznak minden olyan elemet, amelyek szükségesek lehetnek több féle pálya elkészítésére, két külön[14] csomagot használtam, amelyekből a számomra fontosabb elemeket emeltem ki.

15. ábra. Pályaelem sprite-ok

A 15. ábrán a felhasznált elemeket lehet látni, nincsen túl sok sprite, ezekből épült fel az egész pálya. Kezdetben csak néhány statikus akadályt helyeztem el a pályán, mint négyzeteket, mezőket a levegőn és hasonló kikerülhető akadályokat. A kép bal oldalán két oszlopban láthatóak azok a sprite-ok, amelyeket a csomagból vettem, jobbra pedig a saját készítésű elemek látszódnak. Miután sikerült megvalósítani a végtelenített mozgást, utána fogtam bele a pályaelemek tervezésében és elkészítésébe, mivel előtte nem lett volna

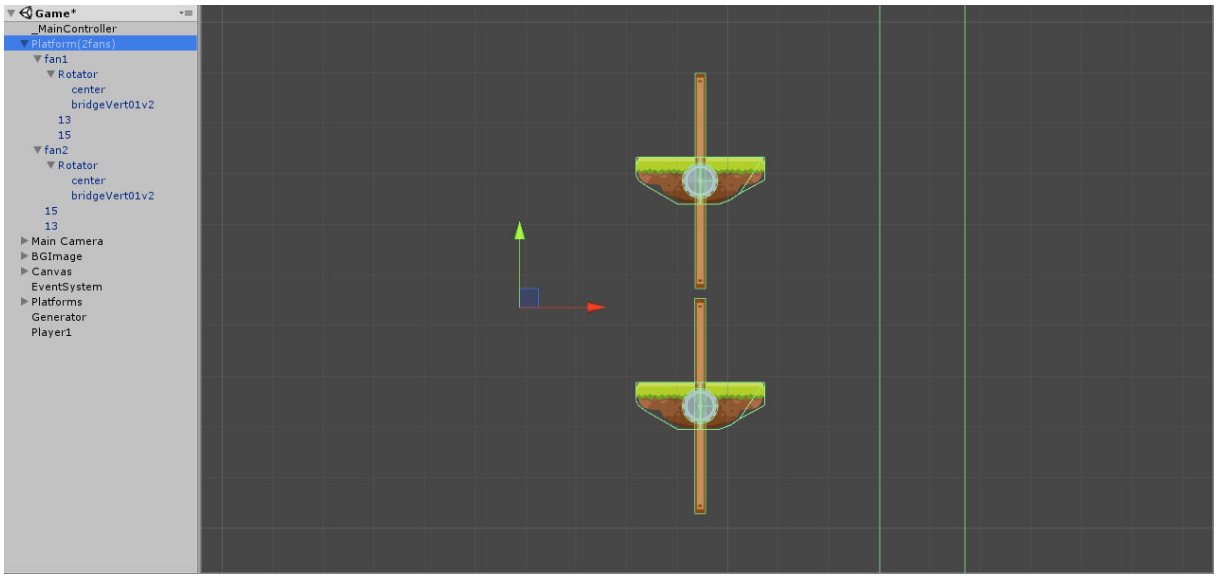
értelme. A terveket főként önállóan találtam ki és papíron rajzoltam meg, hogyan is kellene kinézniük, majd ezután fogtam neki az építkezésnek.



16. ábra. Pályaelemek vázlatai

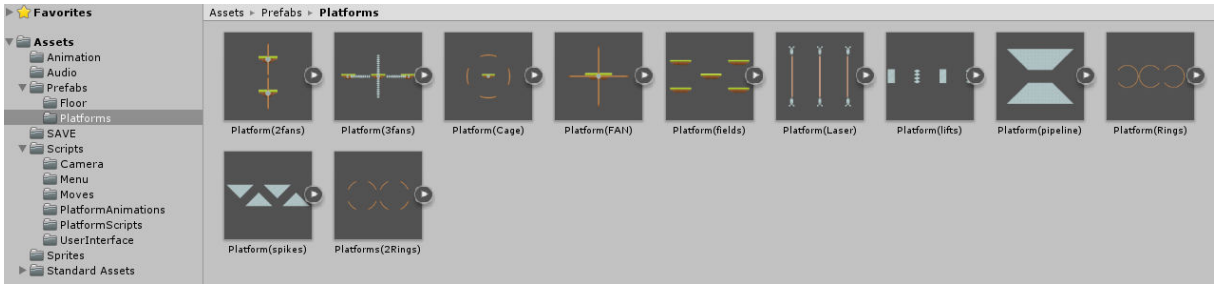
A 16. ábrán néhány vázlat látható arról is, hogy milyen pályaelemekben gondolkodtam először. Ezek közül nem mind kerül be a játékba, ha bekerült, akkor teljesen más formában, mint a vázlatokon. Itt is már az egész könnyű akadályoktól az egészen összetett nehéz akadályokig próbáltam kitalálni a pályaelemeket. Már itt is jelen voltak a forgó, csúszkáló, statikus és ki-be kapcsolódó elemek, a fejlesztés későbbi szakaszában sikerült összetettebb elemeket is hozzáadni a játékhoz. A súlyozás bevezetésével hátrébb sorolódtak, így tényleg csak akkor találkozik vele a játékos, ha igazán messzire jut el a játékban. Ezekhez az összetett elemekhez volt szükség a saját készítésű spriteok-hoz. A mozgó elemekhez kellett külön script, ami mozgatta őket, viszont valamilyen szabvány szerint kellett legyártani az elemeket, ha generálódnak, akkor jó helyre kerüljenek, rendelkezzenek Collider-rel, megfelelő távolság legyen köztük, ne ismétlődjenek és rendelkezzen mindegyik egy trigger-rel. Ezen szempontok alapján lettek összeállítva és lementve, természetesen a generálást script irányította, amelyről későbbiekben részletesebben is írok majd.

Szeretnék bemutatni egy elemet, hogyan is épült fel, milyen hierarchiával rendelkezik és hogyan sikerült belőni a méretét egy-egy elemnek.



17. ábra. Egy pályaelem felépítése

A 17. ábrán szeretném ezt bemutatni, bal oldalon a kékkel kijelölt elemek építek fel ezt a két darab ventilátor. Megfigyelhető a két ventilátor objektum hierarchiája, hogyan épülnek egymásra az elemek. A *13*-as és *15*-ös elem a lebegő föld, míg a *Rotator* objektumon belül található a ventilátor lapátja, ezen van egy forgásért felelős script komponens, mind két ventilátort külön egymástól függetlenül tudjuk ez által forgatni. Ez az elrendezés a hierarchiában lehetővé teszi, hogy csak a lapátok forogjanak, és ne az egész objektum szigetestül. Megfigyelhető még továbbá, hogy a két ventilátort magába foglaló *Platform(2fans)* objektum pivot pontja az a két ventilátor előtt helyezkedik el, azaz a nullához képest el vannak tolva. Ez a generálás miatt szükséges, mivel ennél a pontnál fogva kerül le a játéktérre majd az elem.



18. ábra. Az összes pályaelem

A 18. ábrán látható az összes pályaelem, amelyeket a Platforms mappában tárolunk, ezek Game Object-ek nem sima sprite-ok. Nincsenek lent a játéktéren, ezt már fentebb említett generáló script végzi majd el és ezek közül fogja kiválogatni véletlenszerűen. Összesen 11 elem van, a nehézségi szintjüket pedig az határozza meg, hogy hányadik helyet foglalják el abban a tömbben, ahonnan a script válogatja őket, de erről részletesebben is írni fogok. A projekten belül nagyobb tervezést tehát ez a rész igényelt,

hogy milyen legyen a megjelenés, illetve milyen pályaelemek legyenek, hogyan súlyozzuk és generáljuk le őket. A fejlesztés ezen szakaszán kezdtem el használni a grafikus felhasználó felületet, gombokat és szövegeket, itt is fő szempont volt, hogy esztétikus legyen, illetve ekkor készítettem el a menüt is. A felhasználó felületnek és a menünek is több verziója volt és hosszú út vezetett el a végleges verzióig.

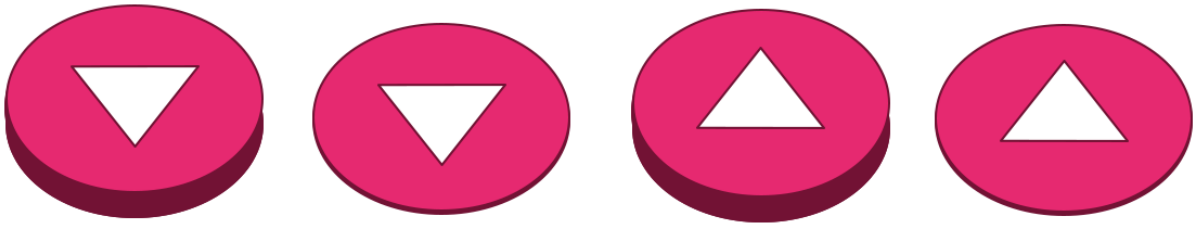
4.2.3. Grafikus felhasználó felület (GUI)

Már beszéltem arról, hogy milyen jó is a Unity-ben a beépített grafikus felhasználó felület, nagyon megkönnyíti a fejlesztést, de ettől függetlenül csak szimpla egyszínű gombokat vagy szövegeket tudunk létrehozni. A fizetős verzióban több szabadságunk és lehetőségünk van szövegek formázására színezésére, de az ingyenes verzióban kicsit több munkával el lehet készíteni személyre szabott gombokat és szövegeket. A feliratok, címek megformázására elég egy betűstílust hozzáadnunk az elemhez és utána színezhetjük szabadon, de sok keresgélés után se találtam olyan betűstílust, amin ne lett volna valamilyen licence, ezért az alapértelmezetten beépített Arial stílus mellett döntöttem. Továbbra is a letisztult formavilágot és az egyszerűséget tartottam szem előtt, ennek tükrében készült el a felhasználó felület. Egyszerű sprite-okat hozzá lehet adni a gombokhoz, amellyel teljesen egyedi kinézetet tudunk adni a gombok számára. Az interneten ahogy sprite csomagokat lehet találni kétdimenziós játékokhoz, úgy különféle gombokhoz és felületekhez is van ingyenesen letölthető csomag. Mivel elég egyszerű képekről volt szó, így csupán mintának használtam fel egy számomra megfelelő sprite-ot, majd az alapján készült a saját verzió.

19. ábra. Kétállapotú gombok spirte-ja

A 19. ábrán helyezkednek el a felhasznált gombok, két állapotot jelölnek, egy lenyomott, illetve felengedett pozíciót. A jobb szélen elkülönítve látható kék színű gomb szolgált mintának, ez alapján készült el a még inkább letisztult végleges verzió. Ezeket a gombokat a menüben is felhasználtam, így teljesen egységesített nézetet adva a játéknak.

Mivel számítógépen fejlesztettem, ezért mindent itt próbáltam ki, viszont számításba kellett venni, hogy telefonon nincsen billentyűzet, ezért valamilyen gombot kell elhelyezni, amely majd az irányításért fog felelni. Szerencsére csak föl, illetve lefelé, tehát a vertikális mozgás van engedélyezve a játékban, ezért két gombnak kell helyet szorítani a képernyőn.

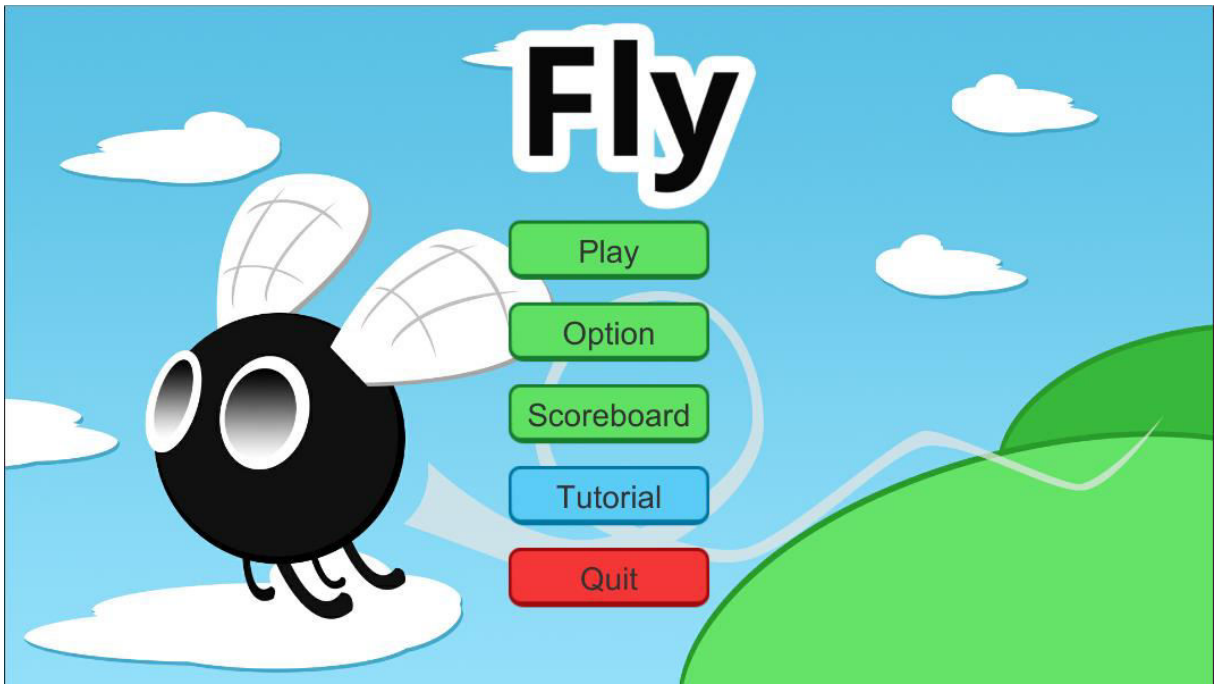


20. ábra. A mozgásért felelős gombok kétállapotú sprite-ja

A 20. ábrán ez az érintő felületre szánt gomb látható, ehhez a régi Game Boy adott ihletet, amelyen található volt kettő színben hasonló gomb. Itt is fontos volt, hogy megmaradjon a letisztult és egyszerű dizájn természetesen. Ahhoz hogy működjenek ezek a gombok importálni kellett a fentebb említett már Standard Assets mappát, amelyben kifejezetten más platformokon való irányítás támogatásához találunk eszközöket.

4.2.4. Menü

A főmenü elkészítésében tulajdonképpen semmilyen nehézség nem volt, ennek ellenére körülbelül négy vagy öt verzió is napvilágot látott, ameddig el nem jutottam a végleges verzióig. Az első alapötlet egy mozgó háttérrel rendelkező menü lett volna, de úgy éreztem felesleges időt és erőforrást pocsékolni, azért hogy a menü jól nézzen ki. Persze nem akartam egy kezdetleges menüt, de ezt túlzásnak éreztem és nem akartam túl sok időt beleölni ebbe. Ezek után különféle HD felbontású egyszerű háttereket próbáltam beszúrni, amelyeken felhők voltak, vagy sima színátmenetes képekkel próbálkoztam. Néhány játék menüjét megvizsgáltam és arra jutottam, hogy érdemesebb lenne egy egyszerű letisztult jó hangulatú képet összeállítani, ami teljesen illik a játékba. Egy vázlat alapján készült el a végleges verzió, amely tökéletesen illet az eddig létrehozott képi világba.



21. ábra. A menü végleges formája

A 21. ábrán látható, hogyan is fest a végeredmény, viszonylag egyszerű formákból épül fel az egész kép. A cím is egy háttér nélküli kép nem pedig szöveges UI elem, erre azért volt szükség, mivel Unity-n belül nem lehet a betűknek szegélyt megadni. A menü elkészítésénél inkább az jelentett fejtörést, hogy több kijelzőn és több felbontásban is ugyanez a látvány fogadja a játékost. Teljesen reszponzív a cím, illetve a gombok kialakítása, így ha csökkentjük a képernyő méretet, úgy minden arányosan csökkenni fog a képen. Továbbá gondolni kellett mobil eszközökre, ahol az ujjunkkal kell rábökni a gombokra, így azokat kellően nagyra kellett méretezni.

A tervezés főként a sprite-ok, képek, elrendezés, látványvilág és a pályaelemek kigondolás jelentette, kódolás szempontjából nem igazán kellett túl sok időt a tervezéssel tölteni. A továbbiakban a fentebb említett feladatok és célok konkrét megvalósítását fogom leírni a már bemutatott eszközökkel és elemekkel.

5. Tervezés megvalósítása

Ebben a fejezetben a scriptekről lesz szó főként, hogy a fő funkciókat hogyan sikerült megvalósítani, illetve a kód részletes bemutatására fog sor kerülni. A scriptek megírásánál az az elv érvényesült, hogy minden fontosabb funkció kerüljön egy külön script állományba, illetve a hasonló célú funkciók ne különüljenek el, így nem került minden egyes függvény külön scriptbe. Összesen tizennégy darab scripttel rendelkezik a játék, a fő script a *MainController.cs*, amelyben több fontosabb függvény van megírva, ezek főként más scriptekben hívódnak meg. Tehát a script állományok között van kommunikáció, de úgy gondolom nem a legoptimálisabb módon hajtottam ezt a részt végre, révén az evolúciós tervezési mintához hasonló fejlesztés miatt. A továbbiakban megpróbálom minél tisztábban magyarázni hogyan is működnek a fő funkciók a játékban.

5.1. Irányítás

Az irányításról többször szót ejtettem már az előzőekben, itt szeretném a teljes működését részletesen kifejteni. Tulajdonképpen egy teljesen alapfüggvényt[15] használtam az elkészítéséhez. A mozgást megvalósító script fájl neve *Move.cs*. Az alapelv, hogy a karakter objektumot forgatjuk a saját középpontja körül valamekkora sebességgel, illetve figyeljük történik-e valamilyen gomblenyomás, ha igen akkor induljunk el a megfelelő irányba. A mozgást megvalósító függvény a következő:

```
public float speed = 10.0F;
public float Rotation = 1.0F;

private void Movement()
{

        float Vertical = Input.GetAxis("Vertical") * speed;
        Vertical *= Time.deltaTime;
        transform.Translate(0, Vertical, 0);

        transform.Rotate(0, 0, Rotation);

}
```

A függvény neve *Movement*, amit az *Update* függvényben fogunk meghívni, a *speed* a mozgás sebességét, míg a *Rotation* a forgás sebességét befolyásolja. Létrehozunk egy lebegőpontos változót, amely elkéri, hogy a billentyűzeten mely gombot nyomtuk le, ezt megszorozzuk a sebességgel és így kapni fogunk egy tört számot. Az így kapott szám csak egy távolságot meghatározó szám, ha megszorozzuk *Time.DeltaTime*-al, akkor igazi sebesség egységet kapunk, mivel ez a változó fogja meghatározni, mekkora távolságot kell

megtenni egy másodperc alatt. A *Vertical* változóban tehát a sebességet tároljuk, amivel mozgatni szeretnénk a karaktert. A *transform.Translate* függvény segítségével tudjuk megmondani, hogy milyen irányba történjen a mozgás, három paraméterrel rendelkezik, x, y és z ebben a sorrendben. A kódban a *Vertical* változó a második paraméterként van átadva, ez azt jelenti, hogy y tengelyen tudjuk mozgatni a karaktert. A *transform.Rotate* függvény szintén három paraméterrel rendelkezik, és azt tudjuk megadni, hogy x,y vagy z tengely körül szeretnénk forgatni az objektumot. A kód szerint a z tengely körül forgatjuk, ezt úgy kell elképzelni mintha a karakter középpontjába hátulról, beleszúrnánk egy tűt és azon forgatnánk valamekkora sebességgel. A furcsaság az, hogy nem az objektum Rigidbody-ját mozgatjuk, ez azért van, mert nem akarjuk, hogy bármilyen fizikai hatás érvényesüljön a karakter esetében csakis kizárólag az ütközés, de ahhoz ott van a Collider. A fenti script viszont érintő képernyő esetén, azaz telefonon nem tudja lekérdezni, melyik gombot nyomtuk le, ezért egy külön scriptet írtam a telefonos irányítás kezelésére, ennek a neve a *Move_touch.cs*. A kód a következő:

Ugyanúgy *Movement* itt is a függvény neve, amit majd az *Update*-ban meghívunk. Az

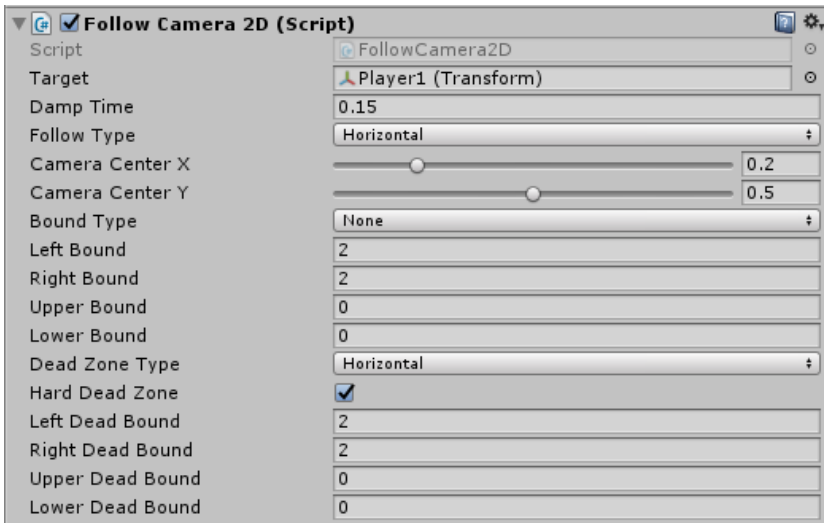
```
private float movValue;
private float velMovValue;

public float smoothTime = 1;

private void Movement()
{

        float Vertical = CrossPlatformInputManager.GetAxis("Vertical") * speed
        *Time.deltaTime;

        movValue = Mathf.SmoothDamp(movValue, Vertical, ref velMovValue, smoothTime);

        transform.Translate(0, movValue, 0);
        transform.Rotate(0, 0, Rotation);

}
```

előzőekben *Input.GetAxis*-sel kérdeztük le a gomblenyomást. Amikor lenyomunk egy gombot, akkor 1 vagy -1 értéket fog fölvenni, ha nincs lenyomva, akkor 0 értéket vesz fel, ez a függvény pedig lehetővé teszi, hogy ne egyből álljon be 1 vagy -1-re. Fokozatosan felgyorsul, ha nem így történne, akkor nagyon darabos lenne a mozgás. A mobilos irányítás esetében *CrossPlatformInputManager.GetAxis*-t használunk, ehhez be kell importálni ezt a csomagot, majd névtérként is hozzá kell adni a scripthez és elérhetővé válik. Ami újdonság az a *Mathf.SmoothDamp* nevű függvény, ezt azért kell itt alkalmazni, mert érintő gombok esetén sajnos nem működik a *GetAxis* sajátossága, hanem egyből felveszi gomblenyomásra a karakter az 1 vagy -1 értéket. Ez nagyon szaggatott mozgást

eredményez sajnos, de ez a függvény megcsinálja a felgyorsítást az adott értékekhez. A működéshez még annyit kell tenni, hogy a gomb UI elemekhez hozzáadunk egy *AxisTouchButton* scriptet, amelyet már tartalmaz a Cross Platform csomag, majd a beállításoknál megadjuk a vertikális irányt és az értéket, amit lenyomásra fel fog venni. A mozgatás ennyiből állna telefonon és számítógépen.

5.2. Kamera

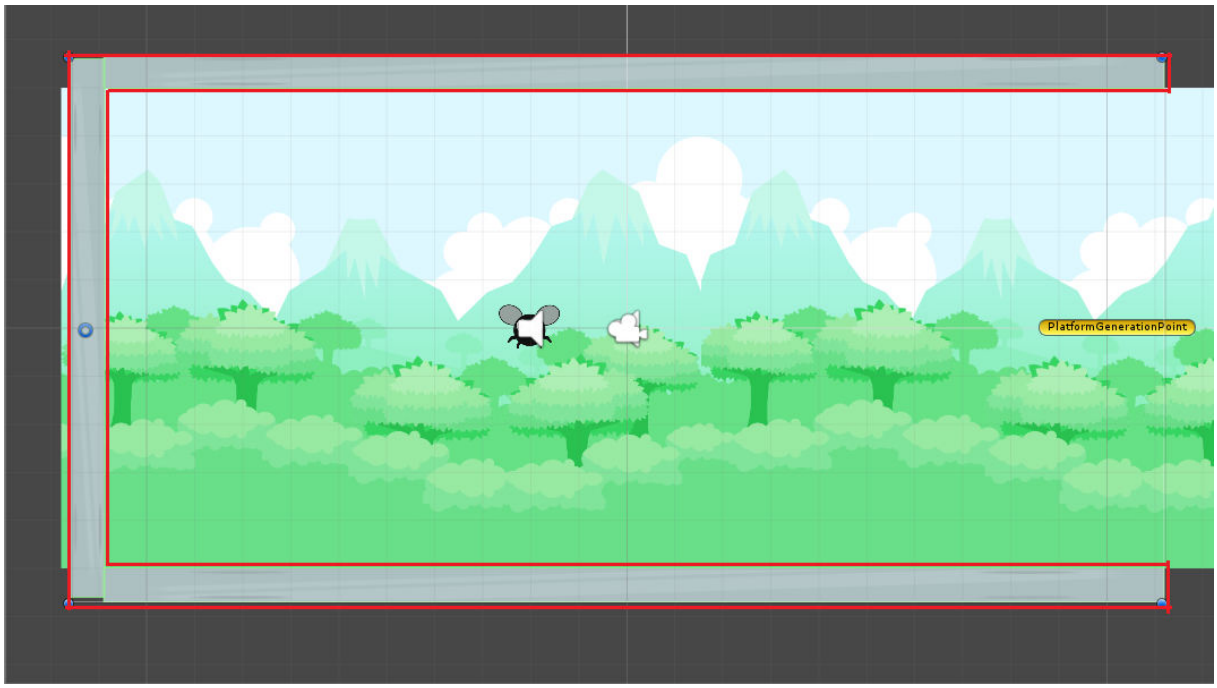
A kamera esetében három dolgot kell megoldani, az első, hogy csak egy irányba mozduljon el a kamera, a második, hogy valamilyen holt teret biztosítsunk a karakternek, ahol szabadon tud mozogni, és a harmadik, hogy ne tudjunk kimenni a kamera látószögéből. Rengeteg keresgélés és próbálkozás után sikerült találni egy kész scriptet[16], ahol a szerkesztőn belüli beállításokkal, három problémából az első kettőt tökéletesen meg lehet oldani. A script neve FollowCamera2D és a kódból nem is, hanem a beállítási lehetőségeket szeretném bemutatni.



22. ábra. A kamera beállítása

A scriptet a Main Camera objektumhoz kell komponensként hozzáadni, utána megjelenik a következő lenyíló ablak számunkra. Nagyon jól skálázható ezzel a scripttel a kamera, ki kell választani, mit kövessen a kamera, esetünkben a karaktert. A Damp Time növelésével lehet késleltetni azt a bemozdulást, ami a karakter és a kamera megmozdulása között eltelik. Beállíthatjuk, hogy a kamera milyen irányba kövesse a célpontot, a horizontális azt jelenti, hogy csak az x tengelyen fog mozogni a kamera. Alapértelmezetten a célpont a kamera közepén fog elhelyezkedni, ezt tudjuk eltolni a csúszkák segítségével, így a karakterünk egy kicsit a képernyő bal oldalán fog lehelyezkedni. Ezek után hozzáadhatunk egy dead zone-t, azaz holt teret, így létrejön egy olyan terület a kamera

látószögén belül, ahol mozoghat a célpont anélkül, hogy bemozdulna a kamera. Ezekkel a beállításokkal tökéletesen megvalósítottuk a kitűzött célokat. Egyedül azt kell orvosolni, hogy ne tudjunk kimenni a kamera látószögéből, erre is rengeteg féle megoldással próbálkoztam, de a legegyszerűbb és fapadosabb megoldás mellett döntöttem, viszont nagyon jól működik.



23. ábra. A kamera köré húzott keret

A 23. ábrán látható a megoldás, Unity fórumán[17] ajánlották, hogy a kamera köré húzzak egy keretet, úgy sem látszódik a játékos számára és a feladatot maradéktalanul ellátja. A pirossal körbekeretezett sprite-ok alkotják a keretet, amelyeket elláttam Collider-rel, ezek a hierarchiában a Main Camera objektumon belül helyezkednek el, így ha megmozdul a kamera, akkor együtt fognak mozogni vele. Mind három kritériumnak eleget tettünk maradéktalanul ezzel az ügyes trükkel és a talált scripttel.

5.3. Háttérgörgetés

Az egyik legtöbb kísérletezést[18] ez a rész okozta, több módszerrel is próbálkoztam, míg sikerült egy teljesen jól működő scriptet megírni. A feladat, hogy a karakter mozgatása által eredményezett előrehaladást érzékeltessük és végtelenítsük. Ezt úgy tudjuk elérni, ha rendelkezünk egy seamless háttérrel, ami azt jelenti, ha több képet egymás mellé rakunk, akkor folytatják egymást. Ezt a hátteret pedig folyton egymás után kell illeszteni, hogy úgy érezze a játékos halad előre, mivel a háttérben folyton elhaladó fák és bokrok a mozgás illúzióját keltik. Az elv tehát a következő, elhelyezünk néhány hátteret, és amikor a

kamerán kívülre kerül az első kép, akkor berakjuk a sor végére, így erőforrást is takarítunk meg, mivel mindig ugyanazt a pár képet tologatjuk, és nem rakunk le a játéktérre újabb és újabb elemeket. A jelenetben elhelyezünk négy darab hátteret egymás mellé illesztve, ennek a hierarchiában BGImage a neve, ezen belül található a négy kép. Azért volt négy darabra szükség, mivel a kameralátószögét bár két darab képpel le lehet fedni, mozgás közben három képet is láthatunk egy időben, ezért kellett egy negyedik, amit majd mozgathatunk. A görgetést megvalósít ó script fájl neve *BackgroundScroll.cs*, amely tartalmaz egy tömböt, ez tárolja a 4 darab hátteret. A görgetés logikája a kódban a következő:

```
public const float SPRITE_SIZE = 13.26f;

private float distancePassed = SPRITE_SIZE*2;

if(distancePassed < cam.position.x)
{
	int orderIndex = _orders[0];
	_orders.RemoveAt(0);

	backgrounds[orderIndex].localPosition += new Vector3(backgrounds.Length *
	SPRITE_SIZE, 0, 0);

	_orders.Add(orderIndex);

	distancePassed += SPRITE_SIZE;
}
```

A tömb neve *backgrounds*, ebben van a négy darab háttér eltárolva, rendelkezünk továbbá egy *_orders* nevű listával, ezt feltöltjük annyi elemmel, ahány elemünk van a tömbben. Ezt a listát fogjuk használni arra, hogy pakolgassuk a háttereket. A kódban látható egy változó, a *distancePassed*, amely arra szolgál, hogy nyilván tartsuk, mekkora távolságot tettünk már meg. A *SPRITE_SIZE* megadja, hogy két háttér pivot pontja között mekkora a távolság, és megszorozzuk kettővel, mert így biztosan a leghátulsó kép már kiesik a látószögből. Ha a kamera x pozíciója nagyobb ennél a távolságnál, azaz elhagyja a kamera látószöge balról az első hátteret, akkor fogja az *_orders* lista nulladik elemét, majd a *backgrounds orderIndex*-edik elemét, azaz a nulladik pozícióját megnöveli annyival, hogy az jobbról az első helyre kerüljön be. Végezetül pedig növeljük a megtett távolságot egy háttér szélességével. Ez az egyszerű logika végzi a háttér mozgatását.

5.4. Pályaelemek generálása és azok súlyozása

A pályaelemek lerakása a pályára szintén egy nehéz feladat volt, de az előző részben is látható volt, hogy milyen egyszerű logikával lehet végrehajtani bizonyos feladatokat, ráadásul viszonylag kevés kódolással. Két script is tartozik ehhez a feladat megoldásához,

a *PlatformGenerator_Upgraded.cs* tartozik, a nevéből látható, hogy itt is több scripttel próbálkoztam. Az elv, hogy ha elérünk egy bizonyos pontot, akkor bekerül egy új elem, így nem előre generálódik több száz pályaelem, mert az túl nagy erőforrás igényű lenne, hanem mindig csak egy pár darab kerül le a játéktérre, amit elhagytunk az pedig eltűnik. Itt is tömböt használunk a pályaelemek tárolására és egy random függvénnyel válogatjuk ki mindig a következő elemet. Vizsgáljuk meg a függvényt, amely lerakja a pályaelemeket a játéktérre.

```
public GameObject[] platforms;
public Transform platformGenerationPoint;
public float platformDistanceBetween = 3;

private float _plafrmWidth;
private float platformDistancePassed;
private MainController mainController;

platformDistancePassed = Camera.main.transform.position.x;

mainController = FindObjectOfType<MainController>();

void Start () {

	pooledObjectList = new List<GameObject>();

	for (int i = 0; i < platforms.Length; i++)
	{
		GameObject obj = (GameObject)Instantiate(platforms[i]);
		obj.SetActive(false);
		pooledObjectList.Add(obj);
	}

}

private void GeneratePlatforms()
{
	if (platformDistancePassed < platformGenerationPoint.position.x)
	{
		int randomNumber = 0;
		do
	    {
	        randomNumber = Random.Range(0, platforms.Length - _gettingHarder);
	    }
	    while (randomNumber == _previousRandomNumber);

	    _previousRandomNumber = randomNumber;

	    if (mainController.score > 0 && mainController.score % 3 == 0)
	    {
	        Debug.Log("nehezseg:" + _gettingHarder);
	        _gettingHarder = Mathf.Max(_gettingHarder - 1, 0);
	    }

	    PlatformInfo info =
	    platforms[randomNumber].GetComponent<PlatformInfo>();

	    _plafrmWidth = info.size.x;

	    Vector3 spawnPos = new Vector3(platformDistancePassed,
	    transform.position.y);

	    GameObject go = pooledObjectList[randomNumber];
	    go.transform.position = spawnPos;
	    go.transform.rotation = transform.rotation;
	    go.SetActive(true);

	    platformDistancePassed += _plafrmWidth + platformDistanceBetween;
	      }
}
```

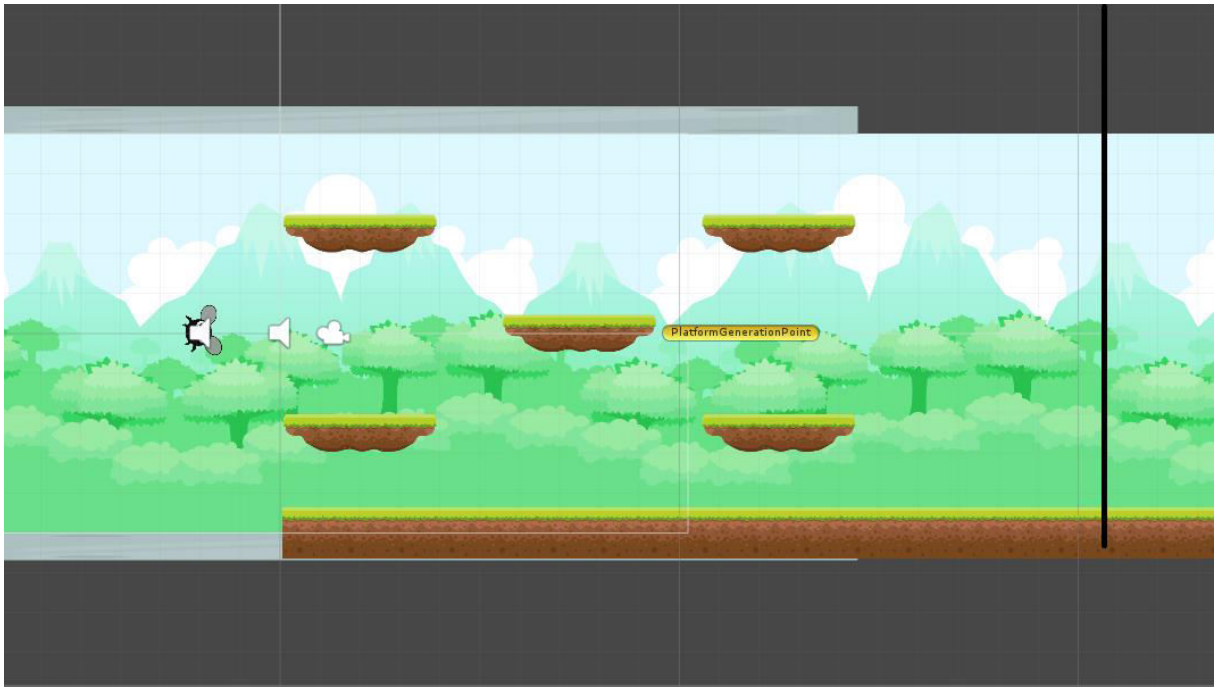
A kód első soraiban természetesen definiálások és deklaráció található meg, itt

láthatjuk a *platforms* nevű tömböt is, amit feltöltünk a kész pályaelemekkel. Azt hogy egy-egy pályaelem milyen széles, azt a *PlatformInfo.cs* nevű script segítségével kapjuk meg, egyetlen sorból áll az egész script és minden elemhez hozzá van rendelve, ebből tudjuk meg, hogy x tengelyen mekkora helyet foglalnak el ezek az akadályok. A *platformDistancePassed* nevű változóba lekérjük a kamera x tengelyének pozícióját, ami induláskor nulla lesz, így le fog futni egyből a generáló függvény, mivel kisebb számot kapunk, mint a generáló pont elhelyezkedése. A *platformGenerationPoint* a kamera látószögén kívül jobb oldalt helyezkedik el, körülbelül 10,3-as eltolással induláskor, ekkora függvény le fog futni, mivel teljesül a feltétel, azaz 0 kisebb, mint 10,3. Ekkor generálunk egy véletlen számot, amit a *randomNumber*-ben tárolunk el, utána elkérem a tömb *randomNumber* sorszámú elemének a szélességét, amit a *PlatformInfo* nevű scriptből kimentek a *_plafrmwidth* nevű változóba. Most már tudjuk, hogy melyik elem fog lekerülni a pályára és annak a szélességét, ezt pedig a *platformDistancePassed* értéke által meghatározott pozícióban fog klónozással lekerülni a pályára, ami az első elem esetében a nulla pozíciót jelenti x tengelyen. Ezek után a *platformDistancePassed* értékéhez hozzáadjuk a pályaelem szélességét és a még a *platfromDistanceBetween* értékét, hogy szellősebben kerüljenek le az elemek. Ha egy nagyobb elemet generálódott a pályára, akkor a generáló pont most kisebb lesz, mint a pályaelem vége, így a függvény nem fog lefutni, ahogy haladunk előre a karakterrel és tolja maga előtt a generáló pontot, amely ha áthalad a pályaelem szélén, akkor újra le fog futni a procedúra. Az elhelyezés, mint említettem klónozással történik, tehát az eredeti példányból készít egy másolatot, és meg kell adni, hogy hová rakja le, ezt a *spawnPos* nevű változó tárolja.

De ezzel a módszerrel, a futás közben végrehajtott klónozással túlságosan nagy erőforrást használunk és természetesen sok memóriaszemetet is generálunk, ezt különösen fontos a mobil eszközök miatt optimalizálni, ahol kisebb erőforrással rendelkeznek a felhasználók. A megoldás az úgynevezett pooling[19] módszer lehet, amelynek lényege, hogy előre legeneráljuk az összes elemet, amit használni szeretnénk, majd csak azokat jelenítjük, meg amelyek éppen soron következnek, illetve eltűntetjük, ha kamerán kívülre kerül egy adott elem. Ennél a módszernél az inicializálásnál megtörténik a klónozás, azaz az *Instantiate* függvény, amely nagyon erőforrás igényes, illetve a *Destroy* nevű függvényt nem is használjuk, ami szintén számításigényes művelet. A kód elején a *Start* függvényben, tehát inicializálásnál a pooling első lépcsőfoka végrehajtódik, feltöltünk egy listát a *platforms* tömb elemeivel majd minden egyes elemet leklónozunk az *Instantiate* függvénnyel és a láthatóságát eltűntetjük, ezt a *SetActive* függvénnyel érjük el. Ezen a

ponton az összes pályaelemből elkészül egy példány, amelyek fel is tűnnek a hierarchiában, viszont a generálásra vonatkozó szabályok nem változnak, egyedül a megjelenítés, ami most már a *SetActive* függvénnyel fog történni. Az egyedüli buktatója ennek a módszernek az, ha újra fel szeretnénk használni egy elemet, ahhoz előtte el kell tűntetni, ezt könnyen végre tudjuk hajtani, mivel a Unity rendelkezik egy beépített *OnBecameVisible* nevezetű függvénnyel, amely akkor fog végrehajtódni, amikor a kamerán kívülre kerül az adott pont. Ezt a *BecameInvisible* script hajtja végre, ez a script minden pályaelemhez hozzá van rendelve, méghozzá az elem széléhez, így amikor a kamera látószöge eléri ezt a pontot, akkor már nem fog látszódni a pályaelem többi része.

A talaj generálása is ugyanezzel a módszerrel történik, annyi különbséggel, hogy ott egy darab pályaelem van a listában és a *for* ciklusban megmondhatjuk, hány darabot klónozzon le előre. Ebben az esetben három darabot használtam, mivel ez volt a legtakarékosabb úgy, hogy a kamera látószögében mindig csak két elem tud egyszerre megjelenni, így mindig lesz egy, ami nem látható és tud mihez nyúlni a függvény.



24. ábra. A generátor pont elhelyezkedése, illetve a határ, ahol új elem kerül a játéktérre

A 24. ábrán látható, hogy hol helyezkedik el a generáló pont (sárga szövegbuborék), illetve hogy, hova mutat a *platformDistancePassed* változó értéke, ezt a fekete vonal jelzi, amint a generáló pont eléri ezt a vonalat, a függvény lefut és ezen a vonalon y tengelyen nullához igazítja, azaz középre fogja elhelyezni az elemet.

A fenti kód még tartalmazza a súlyozás logikáját is, amelyet a random szám generálásánál láthatunk. A *platforms* nevű tömbben nehézség szerint vannak elrendezve,

az elemek, minél nehezebb, annál hátrébb található a tömbben. A *gettingHarder* értéke alapértelmezetten hat, amelyet kivonunk a tömbelemek számából, ez azt fogja eredményezni, hogy a *Random.Range* nevű függvény a tömb első feléből fog válogatni, tehát a könnyebb elemek közül. Megfigyelhető, hogy növeljük ennek az értékét, a pontszámtól függően, ha nem nulla és hárommal osztható a pontszám, ezt pedig a *MainController.cs* scriptből tudjuk elkérni. A *MainController*-t a *FindObjectOfType* segítségével találjuk meg és így lehetséges lekérni a jelenlegi pontszámot, minden harmadik pont után egyel nő a *gettingHarder* változó értéke, egyre több elem közül válogathat a random függvény. Látható benne egy sor, amely arra szolgál, hogy ne kerülhessen egymás mögé két ugyanolyan elem, ezt a *_previousRandomNumber* változóval ellenőrizzük. Megadunk neki egy értéket, amit biztos nem vehet fel, majd indulásnál beletöltődik a generált random szám. Így mindig nyilván van tartva, hogy előzőleg milyen sorszámú elem került le a játéktérre, ha a következő is ugyanolyan lenne, mint az előző, akkor új számot kérünk.

5.5. Pontszerzés

A pontokat akkor kap a játékos, ha áthalad egy-egy pályaelemen, nincs különbség nehezebb vagy könnyebb elem között. Minden akadály rendelkezik egy trigger Collider-rel, ami azt jelenti, hogy nem ütközünk bele, viszont érzékeli, ha áthalad rajta valami és ennek hatására egy esemény fog történni. Fentebb már leírtam mi a teendő, ha ilyen Collider-t szeretnénk létrehozni, itt az akadályok végén található ez a trigger, ahol már biztonságban túljutottunk a pályaelemen. A pontok megszerzése a következőképpen történik:

```
public AudioSource coinSound;

public void OnScoreEarned()
{

        if (!coinSound.isPlaying) coinSound.Play();

        score = score+1;

        UI.Instance.UpdateScoreText(score);

}
```

A *MainController.cs*-ben található ez a függvény, amely eggyel növeli a *score* változót, ami viszont érdekesebb, hogy rendelkezik egy audió fájllal és egy UI elemmel. Az *AudioSource* típusú *coinSound* változóhoz drag and drop módszerrel a szerkesztőben hozzá tudjuk adni a kiválasztott audió fájlt, ezt a Play függvénnyel tudjuk lejátszani.

Fontos hogy ne akadjanak össze a hangok, kell elé egy feltétel, hogy csak akkor játszódjon le, ha éppen nincsen lejátszás alatt. A UI elemek frissítésével, illetve különféle adatok kiíratásával egy külön fejezetben fogok foglalkozni.

5.6. Játékmódok

Összesen öt féle játékmód van a játékban, ezekről már korábban is volt szó, fontos elem, mivel garantálja, hogy ne váljon monotonná a játék. Ezek szintén a szerzett pontok után váltakoznak, pontosabban minden ötödik megszerzett pont után megváltozik, ez pedig főként az irányításra fog kihatni. A játékmódok nevét egy enumerációs (*enum*) típusú változóban tároljuk, aminek a neve *GamePlayMechanic*, ez azért jó, mert szövegesen tudunk hivatkozni az egyes játékmódokra. Magát a generálást egy *GameModeSwitch* nevű függvény végzi, amely szintén a *MainController.cs* scriptben található. Ez a függvény a *TriggerNotifier* scriptben fog meghívódni, mindig, amikor a trigger-en áthaladunk a karakterrel, azaz amikor pontszerzés történik. Ilyenkor a következőt vizsgálja:

```
public GamePlayMechanic currentMod = GamePlayMechanic.Normal;

if (score % 5 == 0)
{
	if (!gameModeSwitch.isPlaying)   gameModeSwitch.Play();

	int nextGameMode = Random.Range(0, (int)GamePlayMechanic.COUNT);
	if (GamePlayMechanic.COUNT == currentMod)
	{
		nextGameMode = Random.Range(0, (int)GamePlayMechanic.COUNT);
}
	else
	{
		currentMod = (GamePlayMechanic)(nextGameMode);
	}
}
```

Megnézi minden pontszerzésnél, hogy osztható-e öttel, ha igen akkor lejátssza a kiválasztott audió fájlt, majd generál egy véletlen számot nulla és az enumerációs típus mérete között, tehát 0 és 5 között. A függvény előtt még deklaráltunk egy *currentMod*-ot, amelyben a kezdő játékmódot tároljuk, miután generáltunk egy véletlen számot, megvizsgáljuk, hogy ez nem-e éppen a használatban lévő játékmód. Ha igen akkor generálunk egy újat, ha nem akkor a *currentMod*-ba belemásoljuk a *nextGameMode*-ban tárolt mód nevét. Ha ez megtörtént, akkor egy *switch case* elágazással meghívjuk a játékmódokhoz tartozó függvényt, minden játékmód rendelkezik egy külön függvénnyel, amelyben le van írva milyen változás fog bekövetkezni. Lássuk először a *switch case* szerkezetet.

```
switch (currentMod)
{
        case GamePlayMechanic.Normal:
        Debug.Log("NORMAL");
        PlayerNormalMove();
        break;
        case GamePlayMechanic.Reverse:
        Debug.Log("REVERSE");
        PlayerReverseRotation();
        break;
        case GamePlayMechanic.Sluggish:
        Debug.Log("SLUGGISH");
        PlayerSluggishSpeed();
        break;
        case GamePlayMechanic.Slow:
        Debug.Log("SLOW");
        PlayerSlowRotation();
        break;
        case GamePlayMechanic.Big:
        Debug.Log("BIG");
        PlayerBiggerScale();
        break;
}
```

A *currentMod* fogja eldönteni melyik eset következik be, illetve a megfelelő eseten belül történik a játékmódot megvalósító függvény meghívása is. A függvény nevéből, illetve a konzolra kiírt szöveg alapján jól kivehető, hogy milyen változások is állhatnak be a karakter irányításában. Az alap játékmód a normál, amely az alapértelmezett értékeket tárolja, ezen keresztül szeretném bemutatni, milyen értékek változhatnak meg egyes játékmódok esetén.

```
public void PlayerNormalMove()
{

        Move movement = player.GetComponent<Move>();
        movement.Rotation = 3;
        movement.speed = 10;

        player.transform.localScale = new Vector3(1, 1, 1);

}
```

Hozzá kell adni a scripthez a karaktert, hogy annak le lehessen kérdezni a sebességét, forgását és a skálázását. Mindhárom érték szerepel az összes többi játékmódban, ez azért van, mert ha módváltás után nem állítjuk vissza alapértelmezettre, akkor egymásra halmozódhatnak a változtatások. Így mindig csak egy-egy értékben térnek el csak egymástól a játékmódok függvényei. A játékmód függvények szintén a *MainController.cs*-ben találhatók meg.

5.7. Grafikus felhasználói felület

Ebben a részben szeretnék részletesebben kitérni arra, hogyan is lehet kiíratni fontosabb dolgokat, mit a pontokat vagy az életet. Ha elhelyeztünk egy szöveges UI elemet, akkor ezt scripttel úgy tudjuk összekötni, hogy definiálunk Text típusú változókat, majd szerkesztőn belül drag and drop módszerrel behúzzuk a UI objektumot a scriptbe. A névtereknél a scriptben engedélyezni kell a UnityEngine.UI névteret, anélkül hibát fog dobni. Ezek után semmi más dolgunk nincs, mint értéket adni a szöveg UI elemnek, ezt úgy tudjuk elérni, hogy minden szöveges UI elem tartalmaz egy Text objektumot is, ami maga a megjelenő szöveg a szövegdobozban. Erre tudunk hivatkozni, de csak úgy tudunk például számot kiirtani, ha előtte szöveggé alakítjuk azt. A játék scriptjeiből szeretnék erre adni példát.

```
public Text scoreText;
public Text gameMode;
public void UpdateScoreText(int score)
{
	scoreText.text = score.ToString();
	gameOverScoreText.text = score.ToString();
}
```

Itt látható a pont és a játékmódot megjelenítő Text típusú változó, amelybe a függvény a paraméterben megkapott pontot *string* típusúvá alakítja majd átadja a UI elemnek. Ez a függvény a *UI.cs* scriptben található, a pontot viszont a *MainController.cs*-ben számoltatjuk, az *UpdateScroeText* függvényt is itt fogjuk meghívni, a pontszerzés menüben látható is az *OnScoreEarned* függvényben hogyan hívjuk meg, illetve adjuk át a pontokat paraméterként. Minden amit kiírunk ilyen formában a képernyőre, azt hasonló eljárással kell végigcsinálni, mint más területen is a Unity leegyszerűsíti a feladatot.

5.8. Halál esemény

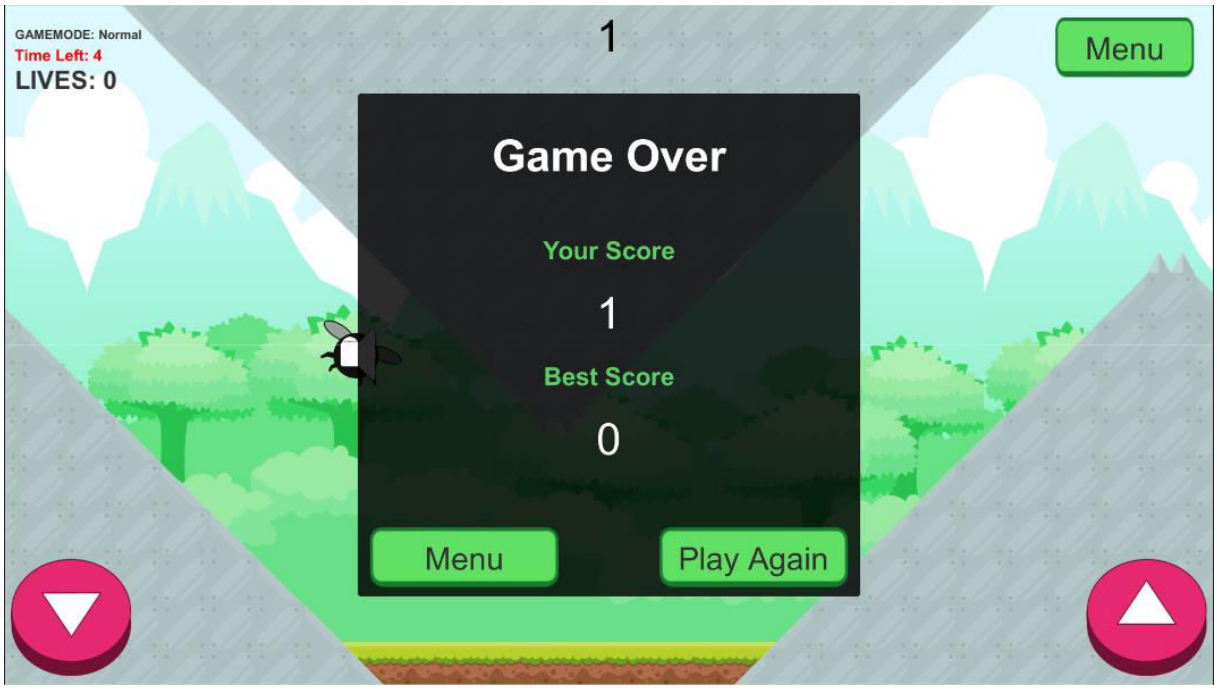
A játék végét jelentő halál két módon is bekövetkezhet, ha túl sokszor ütközik a karakter neki az akadályoknak, illetve ha túl sokáig marad mozdulatlan. Maga a halál egy olyan esemény, amikor megáll a játékmenet és egy új képernyő fogadja a játékost. Ez általánosságban így működik a legtöbb játékban, én is ezt az elvet követtem, ehhez pedig egy kis segítséget is nyújt a Unity. Maga a játék végét kezelő függvény nagyon egyszerű, a *MainController.cs* scriptben található meg.

```
public void PauseScene()
{
	Time.timeScale = 0;
}

public void Death()
{

	// Application.LoadLevel("Game");

	ShowDeathScreen();
	PauseScene();

}
```

A játékmenetet úgy tudjuk megállítani, ha a *Time.timeScale*-t kinullázzuk, ezzel a játékban úgymond megáll az idő, ha ez az érték egy, akkor az idő ugyan olyan gyorsan telik, mint a valóságban, de ha ez a változó kisebb, mint egy, akkor lassítva látható a mozgás játékon belül. Ezt a *PauseScene* függvényt hívjuk meg a *Death* függvényen belül, de van, amit az idő kinullázásával nem lehet megállítani, az pedig a forgás, így a pályaelemek, illetve a karakter tovább fog forogni, de ez nem probléma. Amit szándékosan nem töröltem ki a kódból az a zöld kikommentált rész, mivel a Unity rendelkezik egy *Application.LoadLevel* nevű belső függvénnyel, amely képes más jeleneteket betölteni. Ez azért jó mert, ha bekövetkezik a halál, akár a menübe is dobhatjuk a játékost vagy újraindul a játék. Itt most nem élünk ezzel a lehetőséggel, hanem a játékosra kell bízni a választást, ezért egy úgynevezett gameover screen-t[20] alkalmaztam, ami megjelenik halál esetén, a *ShowDeathScreen* függvénnyel hívódik meg, ez is csak egy pár sor kódolást igényelt.

```
public GameObject deathscreen;

public void ShowDeathScreen()
{

	 deathscreen.SetActive(true);

}
```

Definiálunk egy deathscreen nevű objektumot, amihez drag and drop módszerrel a szerkesztőben hozzáadjuk az általunk elkészített UI elemekből felépített szülő objektumot. A szerkesztőben is lehet engedélyezni, hogy egyes elemek látszódjanak vagy ne, a *ShowDeathScreen* függvényben ezt a *SetActive*-val lehet állítani. Tehát alapértelmezetten ki van kapcsolva a láthatóság, ezt fogja a függvény megmutatni a játékosnak halál esetén.



25. ábra. A gameover screen végleges formája

A 25. ábrán látható ez a gameover screen, ahol a játékosnak lehetősége van új játékot kezdeni vagy a menübe lépni, illetve láthatja a legjobb elért eredményét és az aktuális pontszámát.

Ahhoz, hogy elérje a játékos ezt az állapotot, ütköznie kell a karakterrel szám szerint háromszor, mivel összesen ennyi élet áll a rendelkezésére, már fentebb említettem, hogy a pályaelemek, illetve a kamerát körül ölelő keret rendelkezik Collider-rel. Viszont ütközés esetén nem fog történni semmi, ezeket az ütközéseket kell detektálni, és valamilyen eseményt meghívni, ezt scriptben lehet megvalósítani. Mivel a karakter is rendelkezik Collider-rel, ezért elegendő azt figyelni, hogy ő mikor érintkezik más objektumokkal, ezt a detektálást a *Move.cs* és a *Move_touch.cs* egyaránt tartalmazza.

```
private void OnCollisionEnter2D(Collision2D collision)
{

	if (_delayCollisionTime == 1)
	{
		if (!_audioSource.isPlaying) _audioSource.Play();
		MainController.Instance.LivesCounter();
		DelayBetweenTwoCollision();
	}
	else
	{
		_delayCollisionTime = 1.0F;
	}

}
```

Itt több minden is látható, a Unity saját függvénye az OnCollisionEnter2D, ahol meg kell adni a Collider-t paraméterben, itt is van egy audió fájl az ütközés érzékelésénél, amivel figyelmezteti a játékost, hogy hozzáért valamihez. Meghívja továbbá a *LivesCounter* függvényt, amely levon egyet az életből ütközés hatására, ez a *MainController.cs*-ben van megírva. Itt sincs túl sok kód viszont, ami érdekesebb az a *_delayCollisionTime*, ami a következőt végzi:

```
private float _delayCollisionTime = 1.0F;

public void DelayBetweenTwoCollision()
{
	_delayCollisionTime -= Time.deltaTime;
}
```

A *_delayCollisionTime* értéke egy egész, amelynek az értékét csökkentjük az idő függvényében, ez körülbelül egy másodperc alatt nullára fog csökkenni, ezt az időt késleltetésre fogjuk használni. Ha ezt nem alkalmazzuk az ütközés detektálásánál, akkor túlságosan sok egyszerre, akár több ütközést is érzékelne a játék, ez probléma, mivel az életet akár egy rosszabb ütközés esetén levonhatja az összes életet. Előfordult tesztelés során, hogy a halál bekövetkezésénél negatív számot kaptunk, de ezzel a késleltetéssel körülbelül egy másodpercig nem fog érzékelni újabb ütközést, ez elég idő ahhoz, hogy a játékos arrébb menjen a karakterrel, de elég kevés, hogy csalni tudjon. Amikor ütközés történik, megvizsgálja az *OnCollisinEnter2D* függvény, hogy egyes értéken van-e a *_delayCollisionTime*, ha igen, akkor lejátssza a hangot, illetve levonja az életet, majd nullára csökkenti a *_delayCollisionTime* változót. A következő ütközésnél a függvény látja, hogy nullán van a változó értéke, ezért az *else* ágba fut majd bele és újra egyesre állítja a *_delayCollisonTime*-ot, ezzel előfordulhat, hogy néhány kisebb ütközést nem detektál majd a játék, de ezek fölött szemet hunyunk.

A másik út a halálba, ha a karakter nem mozdul semmilyen irányba négy másodpercig, szükséges volt ezt bevezetni, mert így nem tud hosszú ideig megpihenni a játékos. Nagyon hasonló a visszaszámlálás, mint az ütközés késleltetésénél, csak itt nagyobb számról kell visszaszámolni. Az ezt kezelő függvény a UI.cs-ben található.

```
private const int TIME_TO_DEATH = 4;

float timeLeft = 4;

if (Vector3.Distance(previousPosition, playerGO.transform.position) < 0.1f)
{
	timeLeft -= Time.deltaTime;
	counter.text = "Time Left: " + timeLeft;
	if (timeLeft < 0)
	{
		MainController.Instance.Death();
		MainController.Instance.ShowSwatter();
	}
}
else
{
	timeLeft = TIME_TO_DEATH;
}
```

A *timeLeft* nevű változóba fogjuk azt az időt tárolni, ahonnan visszaszámolunk majd. A feltételben megvizsgáljuk, hogy a karakter pozíciója nulla-e, tehát mozdulatlan, ha igen, akkor elkezd csökkenni a *timeLeft* értéke másodpercenként, ezt egy UI elembe is kiíratjuk folyamatosan. Amikor nulla alá csökken, akkor meghívódik a *Death* függvény és a *ShowSwatter*, ami azt fogja eredményezni, hogy egy légycsapó jelenik meg a képernyőn érzékeltetve, hogy más okból ért véget a játék. Ha mozgásba lendül az idő lejárta előtt a karakter, akkor a *timeLeft* értéke visszaáll az eredeti 4 másodpercre, ezt pedig a *TIME_TO_DEATH* konstans változóban tároljuk. A légycsapó megjelenítéséhez hasonló logikát használunk, mint a gameover screen megjelenítéséhez.

```
public void ShowSwatter()
{

	swatter.SetActive(true);
	if (!swatterSlap.isPlaying && played == 1)
	{
		swatterSlap.Play();
	}
	played = 0;

}
```

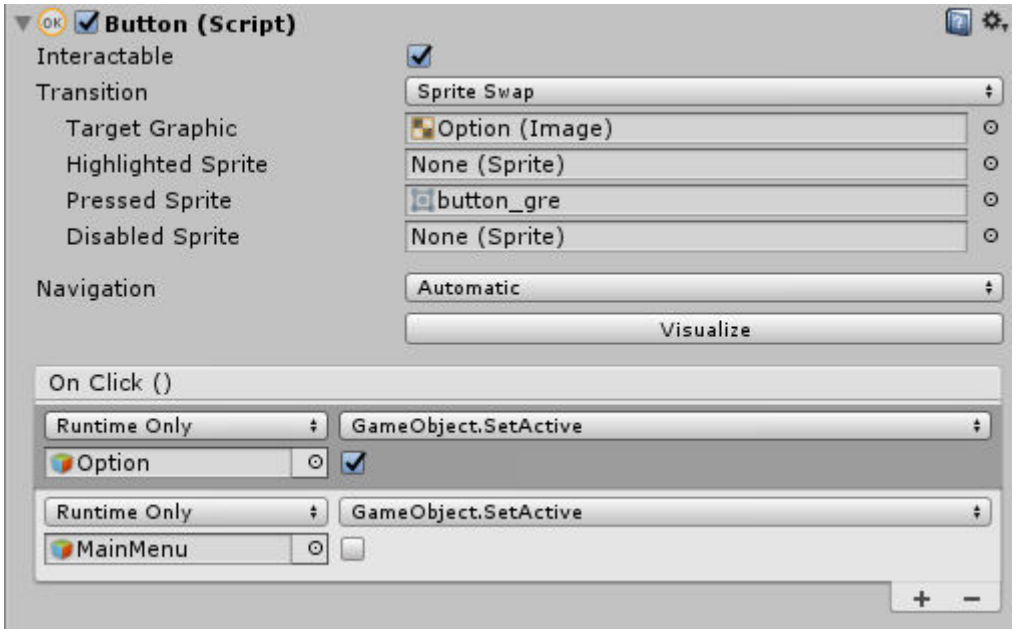
A *SetActive* függvénnyel az alapértelmezetten nem látható spirte-ot aktiválja ilyen módon, és egy audió fájlt játszik le, a megszokott ellenőrzéssel.

5.9. Menü

A menü[21] elkészítésénél minimális kódolásra volt szükség, inkább a reszponzív dizájn elkészítésével kellett időt eltölteni. A már bemutatott háttér, illetve, gombok már megfigyelhetőek voltak, hogyan is helyezkednek el a képernyőn. A menü egy külön jelenet, így az egyik jelenetből a másikba szeretnénk váltani, azt kóddal lehet elérni, a play és a quit gomb lenyomására van két külön függvény.

```
public string startLevel;

public void NewGame()
{
	Application.LoadLevel(startLevel);
}

public void QuitGame()
{
	Application.Quit();
}
```

Korábban érintőlegesen említettem, hogy az *Application.LoadLevel* egy beépített függvény, amellyel jeleneteket lehet betölteni, paraméterben a betöltendő jelenet nevét kell megadni *string* formájában. Az *Application.Quit* szintén egy beépített függvény, ez szolgál az alkalmazás bezárására. Viszont a menünek további menüpontjai vannak, de ezek nem töltenek be újabb és újabb menüpontokat. Egy egyszerű trükkel el lehet tűntetni bizonyos elemeket és közben megjeleníteni új UI elemeket.

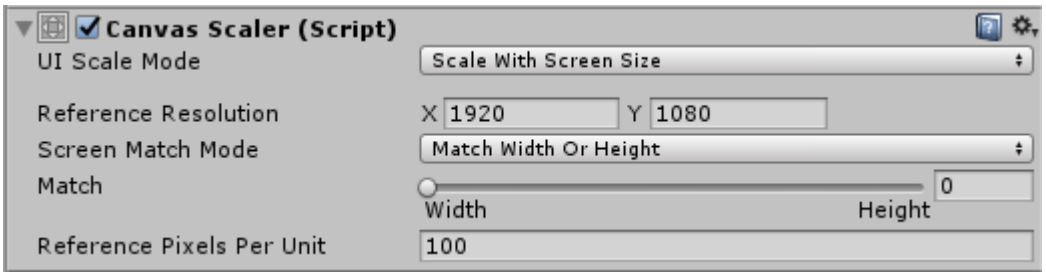


26. ábra. A button script beállításai

A 26. ábrán látható a gombokon aláértelmezetten használat button script, amelyben látható egy On Click esemény. Itt kiválasztható, hogy melyik gomb lenyomására mi

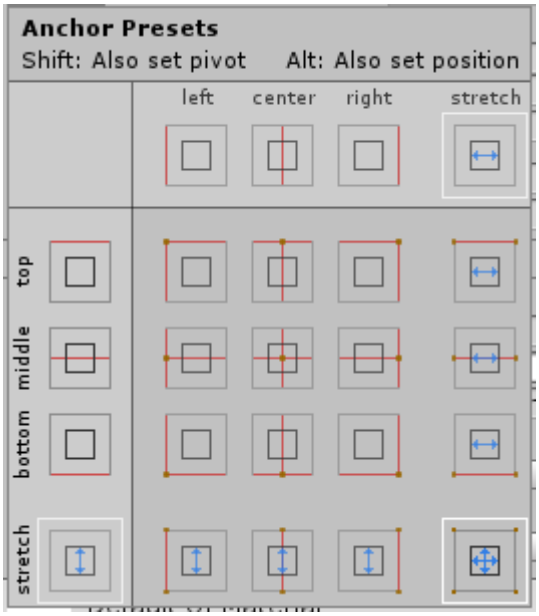
történjen, a képen az option menü gomb button scriptje látható, amely lenyomására megjelenik az Option objektum és eltűnik a MainMenu objektum, mivel a SetActive-nál hamisra állítjuk. Ezzel a módszerrel lehet egy gombnyomásra eltűntetni menüpontokat, így egy jelenetben maradunk, de mégis az az érzése a játékosnak, mintha több ablak között ugrálna.

A menü a HD felbontáshoz van igazítva, ezt úgy sikerült elérni, hogy a Canvas-t, azaz a vászon méretét beállítottuk 1920x1080-as felbontásra, majd erre ráigazítottuk a hátteret.



27. ábra. Canvas méretezése

Ez a 27. ábrán látható, ahogy beállítjuk a Canvas felbontását, ezen a felbontáson fogjuk elrendezni a menüt, mivel a legtöbb telefon is képes erre, illetve innentől kell lefelé skálázni a UI elemeket. Az elrendezés után minden UI elemhez tartozik a következő beállítási lehetőség.



28. ábra. UI elemek igazítása

A 28. ábrán látható hogy a UI elemek hová igazodjanak, itt be lehet állítani, hogy úgymond együtt nyúljanak a felbontás változtatásával és a képernyő melyik részéhez igazodjon, így lehet kicsinyíteni vagy nagyítani a felbontást, a UI elemek ehhez alkalmazkodni fognak.

A menü továbbá lehetőséget nyújt minimális személyre szabásra is, az *option* menüpont alatt ki lehet választani, hogy milyen gombkiosztás érvényesüljön a játékban, továbbá itt lehet lenémítani is a játékot. Itt felmerül a probléma, hogy két jelenet között, hogyan lehet átadni értékeket, ebben is segítséget nyújt a Unity az úgynevezett *PlayerPrefs* függvény révén. A *PlayerPrefs* rendelkezik egy *Set* és egy *Get* ezen belül is *Int*, *Float* és *String* függvényekkel, amelyek segítségével egy memória címre el tudunk menteni bizonyos változókat, két paraméterre van szüksége, az első egy szöveges kulcs, amely alapján be tudja azonosítani az értéket, a második paraméter pedig maga az érték, amit el akarunk menteni. Ennek a segítségével kisebb mennyiségű adatot el tudunk tárolni, úgy hogy adatbázist nem használunk. Ez a gyakorlatban a következőképpen néz ki:

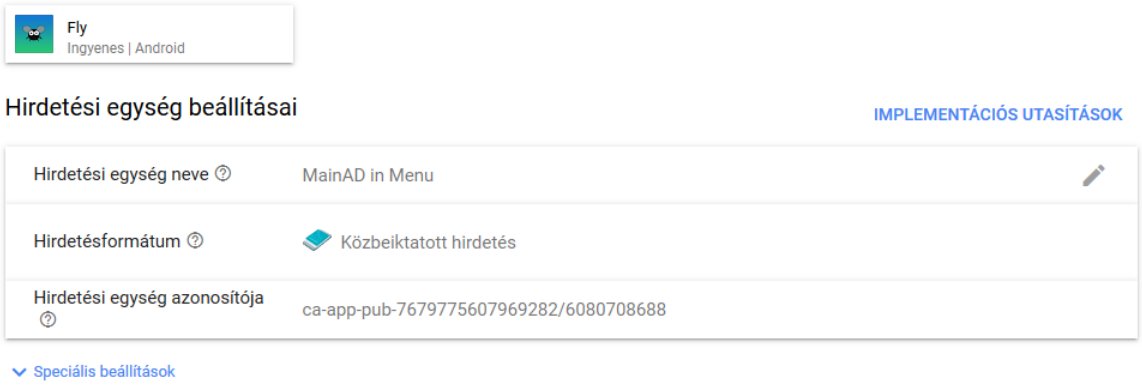
```
public void SetButtonFirstLayout()
{
        layoutNumber = 1;
        PlayerPrefs.SetInt("ButtonLayout", layoutNumber);
        currentLayout.text = "You are using Layout " +
        PlayerPrefs.GetInt("ButtonLayout").ToString();
}

public void SetButtonSecondLayout()
{
        layoutNumber = 2;
        PlayerPrefs.SetInt("ButtonLayout", layoutNumber);
        currentLayout.text = "You are using Layout " +
        PlayerPrefs.GetInt("ButtonLayout").ToString();
}
```

Ez a *MenuController.cs*-ben található, ez a két függvény egy-egy gomblenyomásra fog meghívódni és beállítja a *ButtonLayout* kulcs alatt található értéket, látható hogy a *SetInt*-el beleírunk értéket a *GetInt*-el pedig lekérdezzük az adott kulcsszó alatt található értéket. A játék jelenetben a *MainConroller.cs*-ben pedig inicializálásnál elkérjük a kulcsszó alatt található értéket, amit kimentünk egy változóba, majd annak függvényében *SetActive*-al látható tesszük a felhasználó által választott gombkiosztást. Ugyanígy járunk el a némítás esetében is a menüben gombnyomásra beállítunk a *Mute* kulcsszó alatt egy értéket, majd a játék jelentben elkérjük, majd ennek függvényében aktiváljuk a hangot vagy sem. A hangerőt az *AudiosListener.volume* nevű függvénnyel lehet szabályozni.

Ebben a fejezetben sikerült minden olyan korábban megfogalmazott célt és feladatot végrehajtani, amely iránymutató volt a projekt szempontjából. Rengeteg oktatóanyag írott és videó formában állt a rendelkezésemre, illetve olyan emberek tapasztalata, akik foglalkoztak ezzel és természetesen az iskolai kurzus során is sok ismertre tettem szert és ez a kurzus indította el ennek a projektnek a munkálatait.

6. Google Play és Admob

Átfogó tesztelés után meggyőződtem, hogy semmilyen hiba nincsen a programban, ezért a következő lépcsőfok a játék publikálása volt, ehhez a Google Play biztosít felületet, viszont egy picivel komplikáltabb a játék feltöltése, mivel rengeteg kitételnek kell megfelelnie. Feltöltés előtt több oldalt és videót tanulmányoztam ezzel kapcsolatban, itt szereztem tudomást a Google egy másik szolgáltatásáról az Admob-ról, amely hirdetésekhez nyújt támogatást, így mielőtt feltöltöttem volna a boltba a játékot szerettem volna ezt is beépíteni a játékba, hogy legyen tapasztalatom az éles reklámokkal kapcsolatban is. Az Admob[22] oldalán található egy komplett leírás, ahhoz hogyan kell hozzáadni a projekthez a reklámhoz szükséges felületet. A működése többnyire elég egyszerű és nem is igényel túl sok kódolást, először az Admob rendszerébe kell belépnünk, amit megtehetünk egy meglévő Google fiókkal is, utána ki kell választanunk, hogy milyen reklámot akarunk alkalmazni.



29. ábra. Admob hirdetés egység

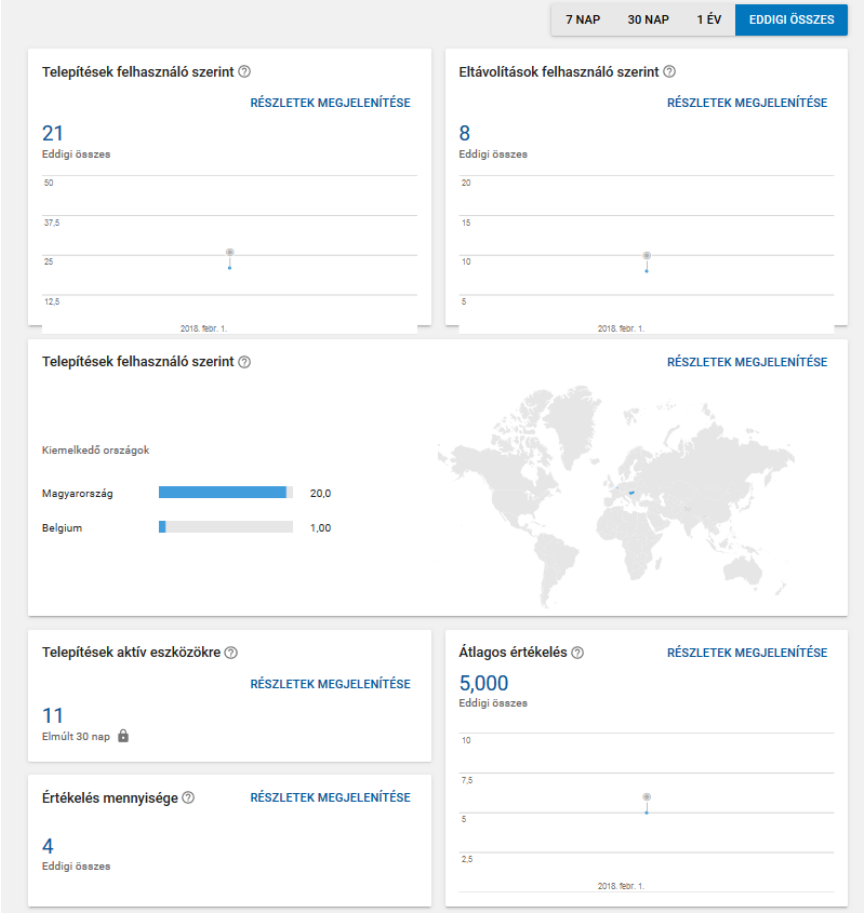
Ha ezt megtettük, akkor kapni fogunk egy azonosítót a hirdetéshez, az Admob felületén innentől kezdve ezen az azonosítón keresztül tudjuk nyomon követni. Ezek után a Unity-be importálnunk kell az Admob plugin-ját, a szerkesztőben a hierarchiában meg fog jelenni három új mappa, ezek fogják tartalmazni a szükséges bővítményeket. Létre kell hozni egy objektumot a jelenetben, ez fog felelni a felület beillesztésért, de nem egy UI elemként kell tekinteni rá, inkább olyan, mint amikor beágyazunk egy weboldalra valamilyen videót.

```
private void AppInitialize()
{

        #if UNITY_ANDROID
            string appId = "ca-app-pub-7679775607969282~5668810994";
        #elif UNITY_IPHONE
            string appId = "ca-app-pub-7679775607969282~5668810994";
        #else
            string appId = "unexpected_platform";
        #endif

        // Initialize the Google Mobile Ads SDK.
        MobileAds.Initialize(appId);

}


private void RequestInterstitial()
{
        #if UNITY_ANDROID
            string adUnitId = "ca-app-pubt-76797728189169282/6080708688";
        #elif UNITY_IPHONE
            string adUnitId = "ca-app-pubt-76791898165969282/6080708688";
        #else
            string adUnitId = "unexpected_platform";
        #endif

        // Initialize an InterstitialAd.
        interstitial = new InterstitialAd(adUnitId);

        // Create an empty ad request.
        AdRequest request = new AdRequest.Builder().Build();
        // Load the interstitial with the request.
        interstitial.LoadAd(request);
}
```

A fenti kód az inicializálást, illetve a betöltést hajtja végre, látható, hogy nagyon egyszerű és rövid kódról van szó, az *AppInitialize* függvényben meg kell adnunk azt az azonosítót, amit az Admob generált számunkra az alkalmazáshoz, ez lesz az *appId*. Itt még csak inicializáljuk az alkalmazásunk és az Admob közötti kapcsolatot. a *RequestInterstitial* függvényben pedig már be is töltjük a korábban már létrehozott hirdetési egységet, amihez szintén kaptunk az Admob-ban egy azonosítót. Ezek a kódrészletek mind megtalálhatók az Admob főoldalán és a leírást[23] követve szimplán el lehet jutni idáig. Ezen a ponton csak annyi dolgunk van, hogy egy megfelelő eseménynél meghívjuk az *interstitial.Show* függvényt, ezt akár gombnyomásra vagy időzítő lejárásánál vagy bármilyen bekövetkező eseménynél a játékban meg lehet hívni.

Miután sikerült összekötni a hirdetést az alkalmazással és beidőzíteni, úgy hogy a lehető legkevésbé legye zavaró a felhasználó számára, le is lehet build-elni Unity szerkesztőben a játékot, a beállításoknál nem sok mindenbe kell belepiszkálni a játék nevét a hozzátartozó ikont és a fejlesztő nevét kell megadni, illetve a verziószámot, ezek után

kapni fogunk egy apk kiterjesztésű fájlt. Ahhoz hogy fel tudjuk tölteni a Google Play-be az alkalmazást szükségünk van egy fejlesztői fiókra[24], szintén a Google fiókunkat tudjuk felhasználni itt is, az android oldalán keresztül lehet elérni ezt az úgynevezett developer console-t, a regisztrációhoz szükséges befizetni 25 dollárt, amely egy egyszeri költség. Ezek után egyből be is tudunk lépni a fejlesztői fiókunkba, ahol végig kell csinálni egy hosszas procedúrát. Könnyen el lehet navigálni a felületen, mivel szinte rávezet bennünket a teendőkre. A feltöltés után várni kell egy pár órát még felkerül a boltba az alkalmazás, és körülbelül egy napnak kell eltelnie, hogy frissüljenek az adatok, mivel a felületen keresztül lehet monitorozni különféle adatokat, illetve össze tudjuk kapcsolni az immár éles projektet az Admob-bal.



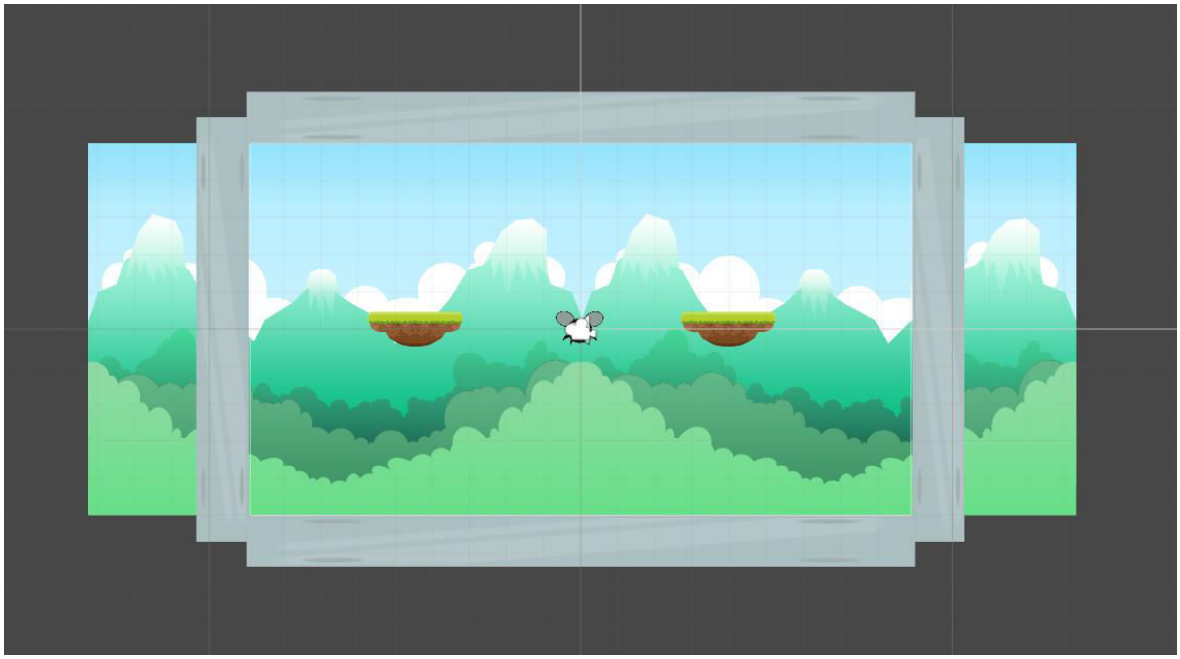
30. ábra. Developer console monitorozás

Ilyen csempéken tudjuk nyomon követni a történéseket, nagyon sok információról tudunk értesülni és miután publikáltuk az alkalmazást van lehetőségünk bővíteni információval a leírást vagy további képeket, esetleg videót hozzáadni a játék névjegyéhez. Lehetőség van arra is, hogy ha további változtatásokat hajtunk végre a játékon, akkor azt be tudjuk frissíteni és egy fejlesztői naplót is kapunk, ahol leírhatjuk milyen új funkciókkal bővült a játék. Ezen a téren nagyon jól kiszolgálja a fejlesztőt a Google, viszont pár nap

alatt több negatívumot is sikerült kiszűrni a rendszerből, például az egyik legfontosabbat, hogy el tudjuk-e juttatni az emberekhez a játékot. Semmilyen megoldást vagy segítséget nem találtam ezzel kapcsolatban és abszolút nem is jelenik meg az új játékok között az alkalmazás, esetleg a lista több századik oldalán. A másik negatívum pedig az Admob rendszerével kapcsolatban ért, mivel azon a felületen is nyomon lehet követni hányszor intéztek kérést az Admob felé, hányszor kattintottak a hirdetésre, hányszor jelent meg a felhasználónak a reklám. Száz fölötti megjelenítés után se termelt semmilyen bevételt a hirdetést, ilyenkor megkérdőjelezi a fejlesztő, hogy szükséges-e a reklám, sajnos a legtöbb játék tele van tömve reklámmal és rengeteg felhasználó panaszkodik is érte. Ezektől a negatívumoktól eltekintve nagyon jó tapasztalatot sikerült szerezni a Google ezen szolgáltatásaival kapcsolatban.

7. Felhasználók visszajelzése

Jó pár felhasználóhoz sikerült eljuttatni a játékot, akik hasznos visszajelzéseket adtak a játékról. Legtöbben a játék nehézségére panaszkodtak, illetve az irányítás nehézkes elsajátítására, ezért úgy döntöttem, hogy létrehozok a menüben egy Tutorial menüpontot, ahol ábrával szemléltetve, illetve szöveggel elmagyarázom, hogyan is működik az irányítás. Továbbá biztosítottam egy gyakorlópályát, ahol szabadon lehet próbálgatni az irányítást, élettől és időtől függetlenül.



31. ábra. A gyakorlópálya

A 31. ábrán látható ez a gyakorlópálya, itt a kamera nem mozdul a helyéről, két darab akadály van, amit gyakorlás céljából lehet kerülgetni, illetve a kamera körbe van határolva, hogy a játékos ne tudjon a kamera látószögén kívülre kerülni. Természetesen egy lassabb forgással és mozgási sebességgel lehet irányítani a karaktert, hogy a játékos ráérezzen a logikájára. Ezt a gyakorlópályát több játékos is örömmel fogadta, viszont élesben a gyorsabb irányítás, illetve az akadályok túl nagy kihívás elé állították a felhasználókat és szinte mindenki megjegyezte, hogy nagyon nehéz. Mivel a fejlesztés során a cél az volt, hogy ne legyen monoton és kihívás elé állítsa a játékost, ezért ezeket a kritikákat valamilyen szinten sikerként könyveltem el, de nem szabad túlságosan ebbe az irányba elmenni, ezért a mozgási sebesség csökkentésével szeretnék könnyíteni az irányításon. A másik visszatérő kritikai az a zenének a hiánya volt játékmenet közben, amiben szintén egyetértek, viszont nehéz találni nem licencelt zenéket, szerencsére ez nem a

játékmechanikát érintő probléma és könnyen orvosolni lehet, ezért a későbbiekben, ha sikerül találni egy megfelelő hangulatú aláfestő zenét, akkor a következő frissítéssel be is fogom élesíteni. Ezen kívül nem érte más negatív kritika a játékot, többnyire mindenkinek tetszett az alapötlet, illetve a látványvilágot is sokan megdicsérték, úgy gondolom, ha a következő verziófrissítésnél a fent említett dolgok bekerülnek a játékba, akkor sokkal sikeresebb is lehet a játék.

8. Összegzés, továbbfejlesztési lehetőségek

Összességében sikerült megvalósítani a célt, amely egy olyan játék, ami megjelenésben és minőségben is fel tudja venni a versenyt néhány játékkal a piacon. Természetesen hordoz magában néhány kisebb hibát, amit meg is említettem, de ezektől eltekintve maradéktalanul sikerült az ötleteimet és az elképzeléseimet megvalósítani erről a projektről. Nagyon jó alaptudást szereztem az elkészítése során, amivel akár újabb és újabb játékok készítésébe is bele lehet fogni. Nagyon motiváló volt számomra, hogy egy olyan projektet készíthettem, amely során látható és látványos dolgok történnek, nem csak egy fekete fehér konzolon látok kiírva adatokat. Továbbá felkeltette a lelkesedésemet a programozás és a fejlesztés iránt is ez a projekt, így amikor akadályokba ütköztem is mindig sikerült fenntartani a motivációt, illetve sikerült ezeken túllépni.

Maga a játék rengeteg fejlesztési potenciált tartalmaz és sok olyan ötlet is maradt még, ami esetleg később bele is kerülhetne a játékba. Egy saját licencementes betűstílussal magán a megjelenésen rengeteget lehetne javítani, az alap betűstílushoz képest, továbbá aláfestő zenével vagy karakter mozgásához illesztett hangokkal tovább lehetne növelni a játékélményt. Tervben volt, hogy ritkán felbukkanna a pályán egy-egy elszórt élet, ha túl nehéznek bizonyul a játék, akár be is lehetne vezetni, továbbá létrehozni egy bolt menüpontot. A boltban lehetne vásárolni a megszerzett pontok után különféle karaktereket, így motiválva a játékost további pontok megszerzésére, esetleg még további pályaelemek tervezése és beépítése, akár még egy olyan adatbázis hozzáadása a játékhoz, ahol minden egyes játékos legjobb pontja szerepelne, így versenghetnének egymással. Természetesen mindig van hová fejlődni, de úgy gondolom maximálisan sikerült a kész játékkal megvalósítani, amit az elején megálmodtam.

Irodalomjegyzék

1. Csatorna neve: Unity3D
   Elektronikus forrás:https://www.youtube.com/user/Unity3D
   Megtekintés dátuma: 2017.06.10
2. Szerző: Unity3D
   Elektronikus forrás:https://docs.unity3d.com/Manual/index.html
   Megtekintés dátuma: 2017.06.10
3. Elektronikus forrás:http://flappybird.io/
   Megtekintés dátuma: 2017.06.18
4. Elektronikus forrás:
   https://play.google.com/store/apps/details?id=com.Beba.JumpColorz&hl=hu
   Megtekintés dátuma: 2017.06.18
5. Elektronikus forrás:
   https://play.google.com/store/apps/details?id=com.piano.tiles.games
   Megtekintés dátuma: 2017.06.18
6. Elektronikus forrás:http://www.megaderek.com/southpark/
   Megtekintés dátuma: 2017.06.20
7. Elektronikus forrás:
   https://play.google.com/store/apps/details?id=com.kiloo.subwaysurf
   Megtekintés dátuma: 2017.06.18
8. Elektronikus forrás:https://en.wikipedia.org/wiki/Unity_(game_engine)
   Megtekintés dátuma: 2018.01.15
9. Elektronikus forrás:https://unity3d.com/
   Megtekintés dátuma: 2017. 05.10
10. Elektronikus forrás:https://assetstore.unity.com/
    Megtekintés dátuma: 2017.05.10
11. Szerző: Unity3D
    Elektronikus forrás:https://docs.unity3d.com/ScriptReference/index.html
    Megtekintés dátuma: 2017.05.17
12. Elektronikus forrás:https://www.visualstudio.com/downloads/
    Megtekintés dátuma:
13. Elektronikus forrás:https://marketplace.yoyogames.com/assets/2136/pixel-fly-sprite
    Megtekintés dátuma: 2017.07.02
14. Elektronikus forrás:https://opengameart.org/content/platformer-art-complete-pack-often-updated
    https://www.gameart2d.com/free-platformer-game-tileset.html
    Megtekintés dátuma: 2017.07.10
15. Szerző: Unity3D
    Elektronikus forrás:https://unity3d.com/earn/tutorials/projects/space-shooter/moving-the-player
    Megtekintés dátuma: 2017.06.15

16. Elektronikus forrás:https://github.com/minhhh/unity-camera-follow-2d/blob/master/Assets/FollowCamera2D.cs
Megtekintés dátuma: 2017.07.23
17. Elektronikus forrás:https://answers.unity.com/index.html
Megtekintés dátuma:
18. Csatorna neve: N3K EN
Elektronikus forrás:https://www.youtube.com/watch?v=QkisHNmcK7Y
Megtekintés dátuma: 2017.07.17
19. Szerző: Unity3D
Elektronikus forrás:https://unity3d.com/learn/tutorials/topics/scripting/object-pooling
Megtekintés dátuma: 2018.02.27
20. Csatorna neve: Brackeys
Elektronikus forrás:https://www.youtube.com/watch?v=bSG5m0Q6W4c
Megtekintés dátuma: 2017.08.06
21. Csatorna neve: Brackeys
Elektronikus forrás:https://www.youtube.com/watch?v=zc8ac_qUXQY
Megtekintés dátuma: 2017.07.19
22. Elektronikus forrás:https://www.google.com/admob/
Megtekintés dátuma: 2018.02.25
23. Elektronikus forrás:https://developers.google.com/admob/unity/start
Megtekintés dátuma: 2018.02.26
24. Elektronikus forrás:https://developer.android.com/distribute/console/index.html
Megtekintés dátuma: 2018.02.26

További források:

1. Szerző: Beck Jonathan
Elektronikus forrás: https://codepen.io/jonnyb1250/
2. Szerző: Polereczki Krisztián
Elektronikus forrás: http://protonox.combord.com/