**bevezető**

Témaválasztás indoklása, annak elméleti és gyakorlati jelentősége. Célkitűzések és eredmények megfogalmazása.

**tárgyalási rész**

**irodalomkutatás:**

Irodalomkutatás (kb. 2-2.5 oldal) gyanánt összeszedheti a játékkészítéshez felhasználható keretrendszerek fejlődését a nullától a kényelmes Unity, UE világig valami ehhez illő cím alatt.

**Használt technológiák, szoftverek bemutatása.**

Amikor eljut erre a pontra, akkor leírja, hogy a fejlesztés során az Unity-t használta, ezért a következő (al)fejezetben ezt fogja részletesen bemutatni.

Unity, Unity néhány fontosabb ’fogalom’, Unity Editor

Majd a Unity-nél kitér, arra, hogy C#-ben írhatunk szkripteket, ezért adja majd magát, hogy bemutassa a C#-et és a Visual Studio-t.

Tömören, röviden tudassa, hogy miről van szó, milyért van rá szükség, ha alternatív lehetőségek is léteznek, akkor itt is indokolhatja a választást.

Unity Common Components

**játék bemutatása(milyen részletességgel?)**

Először nagyvonalakban szokás bemutatni a játékot, ezt már meg is tette. Utána már részletesebben kell írni az egyes részek megvalósításáról: ennek során beszúrhat rövidebb kódrészleteket is, képernyőképeket is, de ezek terjedelme (egyenként nézve) ne haladja meg a fél oldalt, mert akkor már a függelékben lenne a helye.

Amikor a programot magyarázza, akkor érdemes a lényeget leírni: mi a célja annak a résznek, mit ér el vele.

Tesztelés.

**összefoglalás**

Megállapítások és következtetések, tapasztalatok.

**Bevezető**

**Szakdolgozat célja és témaválasztás**

A számítógépes videójátékokkal való kapcsolatom valahol általános iskola 1., vagy 2. osztályában kezdődhetett, amikoris édesanyám megvette élete első számítógépét. Ekkor az unokatestvérem 40-50 darab CD-t adott ajándékba, amik akkoriban a PC Guru, és GameStar magazinokhoz jártak. Ezeken voltak játék demók is, ezekkel játszottam. A következő nagy lépést az jelentette, amikor már édesapámnál is volt számítógép. Ekkor már olyan játékokkal játszhattam, mint a Need for Speed Most Wanted (2005), a FIFA 07, vagy a S.W.I.N.E..

Az évek alatt így kialakult szeretetem a játékok iránt a mai napig jelen van, így nem meglepő, hogy elkacérkodtam a gondolattal: Mi lenne, ha saját játékot készítenék? Ezen gondolat megszületését nagyban elősegítették az egyetemi programozás órákon szerzett tapasztalatok, ahol például egy felhasználói felülettel rendelkező sakkjáték készítésében is részt vettem. A lehetőségeimet mérlegelve rátaláltam a Unity-re, és elhatároztam, hogy teszek vele egy próbát.

Az elmúlt néhány évtized technológiai fejlődésének hatására a videójátékok is ugrásszerűen fejlődtek. A legtöbb korosztályban hatalmas népszerűségnek örvendenek, így nem meglepő, hogy mára már egy teljes iparág szerveződött a videójátékok köré. Napjainkban már mindenütt jelen vannak, több platformon is elérhetőek szinte bárki számára.

Szakdolgozatom célja a Unity játékmotorban rejlő lehetőségek bemutatása, egy általam készített kétdimenziós platformerjáték segítségével. Dolgozatomat a játékmotorok születésüktől kezdődően napjainkig történő fejlődésének fontosabb mérföldköveinek bemutatásával kezdem. Majd a Unity és a használt eszközök bemutatása után, végül a játékomról írok részletesebben.

**A játékról pár szó**

A játéknak nincsen mély története. Egy fantasy világban kapunk egy küldetést, melynek teljesítése a játék célját és végét jelenti.

Kezdéskor három kaszt közül választhatunk karaktert: harcos, mágus, vadász. Ezeknek különböző statisztikáik vannak, például a vadász gyorsabban mozog a harcosnál, de kevesebb életereje van.

Az ellenfelek legyőzése után tapasztalati pontokat és játékbeli fizetőeszközt kapunk, előbbit képességpontjaink növelésére, utóbbit egy NPC kereskedőnél lehet elkölteni.

A karaktert a billentyűzettel, az egyes menügombokat és inventoryt pedig az egérrel lehet vezérelni.

**irodalomkutatás:**

**A Játékmotorok születése**

**Mi is az a játékmotor?**

A játékmotorok tulajdonképpen olyan keretrendszerek, melyek segítségével gyorsan és hatékonyan lehet főként játékokat fejleszteni. Olyan felhasználói felületet biztosítanak, amellyel könnyű dolgozni, és hatalmas segítséget jelentenek a fejlesztés során.

Olyan funkcionalitásokkal rendelkezik, melyek képesek a játékok különböző aspektusait kezelni, mint a:

* 2D, és 3D grafikus megjelenítés, ide sorolva az animációkat, animálás lehetőségét.
* Általános, játékbeli fizikai tulajdonságok.
* Hangok kezelése.
* Mesterséges Intelligencia.
* Felhasználó által írt szkriptek kezelése.

**A kezdetek**

A játékmotorok születése előtt a játékokat szinguláris entitásokként írták meg. A rendelkezésre álló hardware optimális kihasználása érdekében nulláról kellett létrehoznia mindent a fejlesztőknek.

Az első árkád játék a Computer Space volt, melyet az Atari dobott piacra 1971-ben. Ez volt az első kereskedelmi forgalomba kerülő játék. Alapját a Spacewar! adta, melyet Steve Russel készített 1962-ben, és amit sokan az első videójátékként tartanak számon a Tennis for Two (1958) helyett.

Az árkád gépek hardware-ének gyors fejlődése azt jelentette, hogy az addig írt kódok nagy részét nem lehet majd a későbbiekben használni, mivel a következő generációs játékok teljesen más tervek alapján készültek, melyek kihasználták az extra erőforrásokat. A legtöbb játék ennek megfelelően kemény kódolt szabálykészlettel, néhány pályaszinttel és grafikus adatokkal rendelkezett.[4]

Az árkádgépek aranykora után gyakori lett, hogy a vállalatok házon belüli játékmotorokat készítsenek. Erre példa a Nintendo által a NES konzolra fejlesztett motor, amelyet az Excitebike-hoz, majd később a Super Mario Bros-hoz (1985) is használtak. Ez tette lehetővé, hogy a játékos karaktere szépen, egyenletesen gyorsuljon séta tempóból futásba, ahelyett, hogy egy konstans sebességgel mozogna.[8]

**Az elődök**

Az 1980-as években több kétdimenziós, voltaképpen játékkészítő játékokat készítettek. Ezeket construction kits-nek (építőkészleteknek) nevezték. Ilyen volt például a Pinball Construction Set (1983), melyben a felhasználó saját maga pakolgathatta az egyes pályaelemeket, majd játszhatott az általa összerakott pályán. [11]

Bár a kifejezést először az 1990-es években használták, az 1980-as években is létezett néhány korábbi rendszer, amelyeket szintén játékmotornak tekintettek, mint például a Sierra Adventure Game Interpreter (AGI) és SCI rendszerei, a LucasArts SCUMM rendszere és az Incentive Software Freescape motorja. (1986-ban[14]). A legtöbb modern játékmotortól eltérően ezeket a játékmotorokat soha nem használták harmadik féltől származó termékekben (kivéve a SCUMM rendszert, amelyet a Humongous Entertainment licencelt és használt).[?]

A harmadik féltől származó játékmotorok nem voltak elterjedtek az 1990-es évekig, egészen a háromdimenziós grafika megjelenéséig. Maga a játékmotor, mint fogalom is ekkor, a DOOM játék megjelenése környékén alakulhatott ki.

**DOOM**

A DOOM játék 1993-as megjelenése előtt, az id Software nagy bejelentést tett. Azt ígérték, hogy a DOOM feszegetni fogja az addig, számítógépeken lehetségesnek hitt határokat, egy új típusú, nyílt világú játékot ígértek. Alapvetően nem háromdimenziós játék, viszont a sprite-ok reprezentálásával olyan illúziót kelt, így megkapva a 3D titulust.

Ez a sajtóközlemény egy figyelemre méltó dokumentum. Összefoglalta a technológia, a játékmenet, a terjesztés és a tartalomkészítés lenyűgöző újításait. Bevezetett egy kifejezést is, a „DOOM motort”. Ez a kifejezés az id legújabb játékszoftverének motorháztetője alatti technológiát írja le.

Az id azonban nem csak egy új típusú szoftvert készített, hanem új módon rendszerezték a játék komponenseit, az alapvető funkcionalitások játékmotor általi végrehajtásának elkülönítése azoktól a kreatív eszközöktől, amelyek egy adott játékcím játékterét és tartalmát töltötték be. [[2]](https://www.kinephanos.ca/Revue_files/2014-Lowood.pdf)

Ennek az elkülönítésnek az értéke nyilvánvalóvá vált, amikor a fejlesztők elkezdték licencelni a játékokat, és új termékekké alakították át őket új grafikai elemek, világelrendezések, fegyverek, karakterek, járművek és játékszabályok létrehozásával, a „motor” szoftverének minimális változtatásával. Ezzel megszületett a „mod közösség” – egyéni játékosokból és kis független stúdiókból álló csoport, amely új játékokat épített a meglévő játékok módosításával, az eredeti fejlesztők által biztosított ingyenes eszközkészletek segítségével. [1]

Az ilyen technológiák licencelésének gyakorlata hasznos kiegészítő bevételi forrásnak bizonyult egyes játékfejlesztők számára, mivel egy csúcskategóriás kereskedelmi játékmotor licence 10 000 USD-től több millió dollárig terjedhet, a licenctulajdonosok száma pedig több tucat is lehet. az Unreal Engine-nél látható módon. Az újrafelhasználható motorok gyorsabbá és egyszerűbbé teszik egy játék folytatásainak fejlesztését, ami értékes előnyt jelent a versenyképes videojáték-iparban. [12?]

Az újrafelhasználhatóság, a fejlesztési idő csökkenése és a lincencekből származó bevételek, mind arra buzdították a fejlesztőket, hogy egyre jobb és jobb játékmotorokat fejlesszenek. Ezen az úton jutottunk el a legmodernebb játékmotorokig, mint például a CryEngine, a Unity és az Unreal Engine 4.

**Játékmotorok napjainkban**

A játékmotorok további evolúciója már erősen szétválasztotta a renderelést, szkript írást, grafikai alkotásokat és a pályatervezést. A modern játékmotorok a legkomplexebb megírt alkalmazások közé sorolhatóak. Gyakran tucatnyi rendszert foglalnak magukba, hogy jól irányított felhasználói élményt biztosíthassanak.[13]

Ma a játékfejlesztők licencelhetnek egy játékmotort, és újra felhasználhatják a kulcsfontosságú szoftverösszetevők jelentős részét játékok készítéséhez. Bár ez a gyakorlat továbbra is jelentős befektetést igényel az egyedi szoftverfejlesztésbe, sokkal gazdaságosabb lehet, mint az összes alapvető motorkomponens házon belüli fejlesztése.[1]

Ahogy a játékmotor-technológia fejlődik és egyre felhasználóbarátabbá válik, a játékmotorok alkalmazási köre kibővült. Manapság már komolyabb célokra is használják őket: vizualizációs, kiképzési/edzési, orvosi és katonai szimulációs alkalmazások.

A ma már mindenütt jelenlévő háromdimenziós világokat mozgató szoftverek – az olyan játékmotorok, mint az id Software Quake és Doom motorja, az Epic Games Unreal Engine 4, a Valve Source motorja és a Unity játékmotor – teljes értékű újrafelhasználható szoftverfejlesztő készletekké váltak, és szinte bármilyen elképzelhető játék elkészítéséhez felhasználhatóak.

Az elérhetőség megkönnyítése érdekében a játékmotorok új hardverplatformokat céloznak meg, beleértve a mobiltelefonokat (pl. Android telefonok, iPhone) és a webböngészőket (pl. WebGL, Shockwave, Flash, Unity Web Player, O3D).

Ezenkívül több játékmotor épül magasabb szintű nyelvekre, mint például a Java és a C#/.NET (például TorqueX és Visual3D.NET), a Python (Panda3D) vagy a Lua Script (Leadwerks). Mivel a legtöbb 3D-s gazdag játék ma már többnyire GPU-korlátozott (azaz korlátozza a grafikus kártya teljesítménye), a magasabb szintű nyelvek fordításának számítási költségei miatti esetleges lassulás elhanyagolhatóvá válik, miközben az e nyelvek által kínált termelékenységnövekedés a fejlesztők előnyére válik.[17]

Az általam készített játékhoz a Unity játékmotort használtam, melyet a következő fejezetben mutatok be.

[11] Dezaemon, RPG Maker, NScripter: Exploring and classifying game ‘produsage’ in 1990s Japan doi: 10.1386/jgvw.11.3.215\_1

[8] Williams, Andrew (16 March 2017). *History of Digital Games: Developments in Art, Design and Interaction*. [CRC Press](https://en.wikipedia.org/wiki/CRC_Press). pp. 152–4. [ISBN](https://en.wikipedia.org/wiki/ISBN_(identifier)) [978-1-317-50381-1](https://en.wikipedia.org/wiki/Special:BookSources/978-1-317-50381-1).

[13] http://web.cs.wpi.edu/~id111x/c05/slides/intro.ppt

[17] [3D Game Engine Programming - Stefan Zerbst, Oliver Düvel - Google Könyvek](https://books.google.hu/books?id=-vifhqAi0SEC&q=gpu-limited+games&pg=PA338&redir_esc=y)

[12] [id Tech 5 • Eurogamer.net](https://www.eurogamer.net/articles/id-tech-5-interview)

[1] [1511334027376.pdf (usc.ac.ir)](http://ce.eng.usc.ac.ir/files/1511334027376.pdf)

[2] [2014-Lowood.pdf (kinephanos.ca)](https://www.kinephanos.ca/Revue_files/2014-Lowood.pdf)

[3] [(PDF) History and comparative study of modern game engines (researchgate.net)](https://www.researchgate.net/publication/259496289_History_and_comparative_study_of_modern_game_engines)

[4] [The ultimate history of video games: from Pong to... [PDF] (pdfroom.com)](https://pdfroom.com/books/the-ultimate-history-of-video-games-from-pong-to-pokemon-and-beyond-the-story-behind-the-craze-that-touched-our-lives-and-changed-the-world-1st-edition/3kZdoERW2M8)

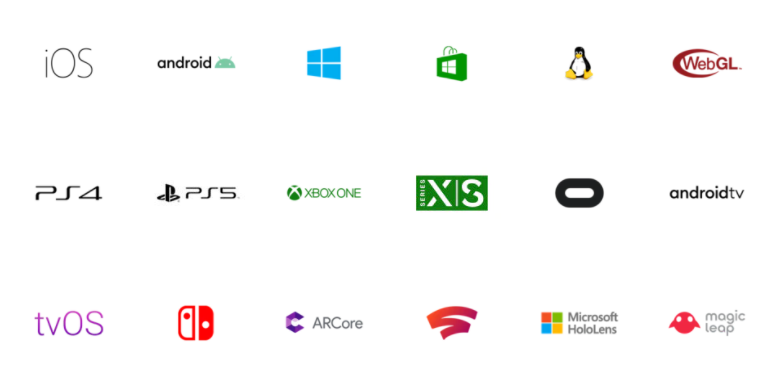
[14] https://www.uvlist.net/groups/info/freescapeengine

**Unity**

A Unity Technologies által fejlesztett játékmotor segítségével kétdimenziós, illetve háromdimenziós videójátékokat, építészeti és mérnöki látványterveket, animációkat, MI megoldásokat, ezeken kívűl pedig egyéb interaktív tartalmakat lehet létrehozni VR szimulációk által.

[Solutions | Unity](https://unity.com/solutions)

A Unity segítségével lehetőségünk van fejleszteni több, mint 25 platformra.



[Unity Real-Time Development Platform | 3D, 2D VR & AR Engine](https://unity.com/)

Ingyenesen letölthető a Unity Personal kiadása, azonban, ha igazoljuk diák státuszunk, akkor feliratkozhatunk az úgynevezett Student plan csomagra, amely magában foglalja a Unity Pro kiadását, hozzáférést oktató videókhoz és más hasznos szolgáltatást.

A Unity nagy versenytársa az Unreal Engine, melyről általánosságban jobb vélemények vannak. Választásom azért esett mégis a Unity-re, mert számomra átláthatóbb felhasználói felülettel rendelkezik, rengeteg oktatóanyag elérhető hozzá és a benne használható programozási nyelv szimpatikusabb.

A Unity-ben lehetőségünk van szkripteket írni a támogatott nyelveken, melyek a C#, UnityScript (úgy is ismert, mint JavaScript Unity-hez) és a Boo.

**C# programozási nyelv**

…

[1. Introducing C# - Programming C# 8.0 [Book] (oreilly.com)](https://www.oreilly.com/library/view/programming-c-80/9781492056805/ch01.html)

**A nyelv története**

A C# programozási nyelv 2002-ben a Microsoft új fejlesztési környezete, a Visual Studio.NET programcsomag részeként jelent meg.

[C# könyv (elte.hu)](https://people.inf.elte.hu/szlavi/Magamnak/Csharp/Programozas_Csahrp_nyelven_IZ.pdf)

A Microsoft azért kényszerült a kifejlesztésére, mert a 90-es években beperelte a Java nyelv licenceit birtokló Sun Microsystems, és a Java nyelv eltávolítását kényszerítették ki a Windows rendszerekből. A per vádja az volt, hogy a Microsoft saját Java keretkörnyezetét a saját operációsrendszer-specifikus függvényeivel és szolgáltatásaival bővítette ki, így az abban fejlesztett alkalmazások nem lettek volna futtathatóak más rendszereken. Tehát sértették a Java platform-függetlenségre vonatkozó alapelvét.

[SUN MICROSYSTEMS VS. MICROSOFT - Chicago Tribune](https://www.chicagotribune.com/news/ct-xpm-1997-10-08-9710080149-story.html)

**A nyelv jellemzői**

A C# egy modern objektumorientált, komponens orientált és típusbiztonságos programozási nyelv. Kényelmes és gyors lehetőséget biztosítva ahhoz, hogy .NET keretrendszer alá alkalmazásokat készítsünk.

A Microsoft a C++ nyelvet vette alapul a C# kifejlesztése során. Olyan nyelvet igyekeztek létrehozni, mely megtartja a C, C++ nyelvek hatékonyságát, ugyanakkor kiküszöböli a komplexitását, hosszú fejlesztési idejét.

[C# könyv (elte.hu)](https://people.inf.elte.hu/szlavi/Magamnak/Csharp/Programozas_Csahrp_nyelven_IZ.pdf)

**.NET keretrendszer (átírás alatt)**

**…**

[1. Introducing C# - Programming C# 8.0 [Book] (oreilly.com)](https://www.oreilly.com/library/view/programming-c-80/9781492056805/ch01.html)

[Mi az a .NET? - Learn | Microsoft Docs](https://docs.microsoft.com/hu-hu/learn/modules/dotnet-introduction/2-what-is-dotnet) / [What is .NET? - Learn | Microsoft Docs](https://docs.microsoft.com/en-us/learn/modules/dotnet-introduction/2-what-is-dotnet)

A .NET egy ingyenes, nyílt forráskódú fejlesztő platform, különböző típusú alkalmazások fejlesztésére.

A .NET az alábbi operációs rendszerekre történő alkalmazásfejlesztést támogatja:

* Windows
* macOS
* Linux
* Android
* iOS
* tvOS
* watchOS

Kezdetben három programozási nyelvet támogatott, integrált fejlesztői környezeteket (IDEs), és egyéb eszközöket biztosít. Ezek a programnyelvek a C#, F# és a Visual Basic.

### **Visual Studio**

Az általam is használt integrált fejlesztői környezet a .NET keretrendszerhez.

Csak a Windows operációs rendszereken elérhető. Széleskörű beépített funkcionalitásokkal bír, melyek a .NET keretrendszerhez lettek tervezve. Az ún. Community Edition verziója ingyenesen letölthető a Microsoft weboldaláról.

[A Tour of C# - C# Guide | Microsoft Docs](https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/)

[.NET introduction and overview | Microsoft Docs](https://docs.microsoft.com/en-us/dotnet/core/introduction)

**Néhány fogalom tisztázása**

Mielőtt bemutatnám a Unity Editor felületét, nem árt néhány fogalmat tisztázni, melyekkel később még sokszor találkozunk.

**Asset**

Asset-nek nevezünk minden, a projektben használt elemet???. Vizuális és audio elemeket képes reprezentálni, mint például háromdimenziós modelleket, textúrákat, sprite-okat, hangeffekteket, vagy zenét. Egy asset érkezhet külső forrásból, de az Editoron belül is van lehetőségünk létrehozásukra.

**GameObject**

A GameObject (a továbbiakban objektum) egy olyan, az Editoron belüli elem, amely komponenseket tartalmaz. Ezek a komponensek határozzák meg az objektum viselkedését és kinézetét. A Hierarchy ablakban tudunk létrehozni objektumokat.

**Components**

A komponensek a játékon belüli események mozgatórugói. Minden objektum funkcionális részei. Egy objektum kijelölése után megtekinthetjük annak komponenseit az Inspector ablakban. Itt és szkripten keresztül is lehetőségünk van ezeket módosítani, törölni, vagy újat létrehozni.

**Scene**

A Scene-ek azok, ahol a tartalommal dolgozunk a Unityben. Olyan eszközök, amelyek egy játékot vagy alkalmazást vagy azok egy részét tartalmazzák. Például létre lehet hozni egy egyszerű játékot egyetlen jelenetben, míg egy összetettebb játékhoz használhat egy jelenetet szintenként, mindegyik saját környezettel, karakterekkel, akadályokkal, dekorációkkal és felhasználói felülettel. Egy projektben tetszőleges számú Scene hozható létre.

**Unity Editor**

A Unity Editor a játékmotor grafikus felhasználói felülete, melyben a tényleges fejlesztést végezzük.

\*\*kép editorrol

(A1)Az Editoron belül talákható egy általános menüszalag. Itt tudjuk például menteni, buildelni projektünket, illetve haladó beállításokat végezni rajta.

(A)A Toolbar a menüszalag alatt helyezkedik el. Bal oldalán olyan alapvető eszközöket találunk, amelyekkel a játéktérbeli elemek pozícióját és méretét tudjuk szabályozni. Középen a Play, Pause és Step gombok találhatóak, melyek a szimuláció futását kontrollálják. Jobb oldalon pedig a Layout lenyitható listából módosíthatjuk az Editoron belüli ablakok elhelyezkedését, de azokat manuálisan is a kívánt helyre tudjuk igazítani.

(B)A Hierarchy ablakban láthatjuk az aktív Scene-eket és a hozzájuk tartozó GameObjecteket. Itt tudunk új komponenseket létrehozni és kezelni őket. Ahogyan az ablak neve is sugallja, ez egy hierarchikus, szöveges reprezentáció a játékbeli komponensekről.

(C)A Game nézet szimulálja le, hogyan is fog kinézni, illetve futni az éppen aktív Scene, az elsődleges kamerán keresztül. A szimuláció a Play gomb megnyomásakor kezdődik.

(D)A Scene nézet lehetőséget biztosít a játéktérben történő vizuális navigálásra, és annak elemeinek módosítására. Két-, illetve háromdimenziós megjelenítésre is képes, attól függően milyen projekten dolgozunk.

(E)A fejlesztés során talán az Inspector ablakot használjuk a leggyakrabban. Itt lehet az éppen kiválasztott játékelemhez olyan komponenseket, illetve tulajdonságokat kapcsolni melyek meghatározzák annak viselkedését, kinézetét.

(F)A Project ablak tulajdonképpen egy Editoron belüli fájlkezelő, amely a rendelkezésre álló asseteket jeleníti meg. A projectbe importált asseteket, a létrhozott szkriptjeinket itt találjuk.

(G)A Console ablakban az Editor által, a felhasználó számára küldött üzenetek jelennek meg a szimuláció futása során. Ezek lehetnek hibaüzenetek, figyelmeztetések, vagy a felhasználó által kiíratott üzenetek hibakezelés során.

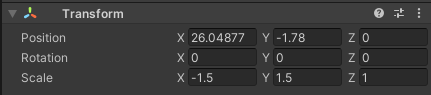
(H)A Status szalag értesítéseket jelenít meg egyes folyamatokról, illetve gyors elérést biztosít a hozzájuk kapcsolódó eszközökhöz és beállításokhoz.

**Leggyakrabban használt komponensek**

Ebben az alfejezetben bemutatom az általam leggyakrabban használt komponenseket, hogy a játék bemutatása gördülékenyebben történjen, ne akkor kelljen a legtöbbet ismertetni.

**Transform**

Alapértelmezetten minden objektum rendelkezik ezzel a komponenssel. Nem lehet, eltávolítani, vagy olyan objektumot létrehozni, ami ne rendelkezne vele, hiszen ez a komponens adja meg, hogy az objektum hol helyezkedik el, hogyan van forgatva, és méretezve.



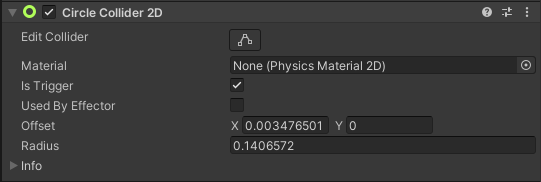
**RectTransform**

A RectTransform a sima Transform kétdimenziós megfelelője. Míg a Transform egy konkrét pontot határoz meg a térben, a RectTransform egy négyzetet egy Canvas komponensen, melybe UI elemeket lehet illeszteni.

**Collider**

A Collider-ek adják meg az egyes objektumok körvonalait, így lehetővé téve a fizikai szimuláció során az ütközéseket. 3D esetén a Collider alakjának meg kell egyeznie a Mesh (3D modell, pl. egy kocka) alakjával. 2D esetén olyat kell választani, amely megfelel a neki szánt célnak. Például egy emberi karakter kaphat Box Collider 2D-t, vagy Capsule Collider 2D-t.

Ez az egyik legszéleskörűbben használható komponens. Beépített függvényekkel lehet meghatározni például, hogy mi történjen, ha két Collider (pl. két karakter a játékban) ütközik egymással.



**Rigidbody**

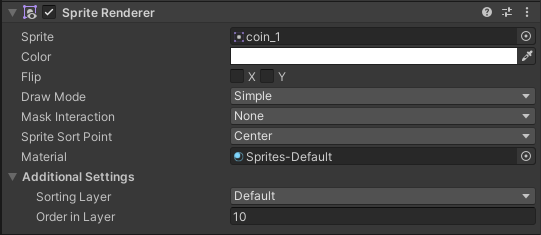
A Rigidbody komponens által vehet részt egy objektum a fizikai szimulációban. Az egyes karaktereket is ezen komponensen keresztül tudja irányítani a játékos szkript segítségével.

Ugyebár a Transform komponens adja meg egy objektum pozícióját. Amikor az objektum helyzete változik, akkor a Transform elküldi az új pozíciót a többi komponensnek, ami így frissít olyan dolgokat, mint például hol legyen megjelenítve az adott objektum, és hogyan vannak a Collider-ek pozícionálva.

A Unity rendelkezik egy a fizikai elemeket szimuláló motorral, amely a Collider-eket mozgatja, és lehetővé teszi, hogy interakcióba lépjenek egymással. Ezt a mozgást viszont továbbitani kell a Transformnak, különben az objektum helyzete nem változna. Így ezért a mozgásért, és kommunikációért, illetve a Collider-ekkel való kapcsolattartásért a Rigidbody felelős.

**Sprite Renderer**

A Sprite-ok kétdimenziós grafikai elemek, tulajdonképpen képek. Ezek a Sprite Renderer komponens által kerülnek megjelenítésre.

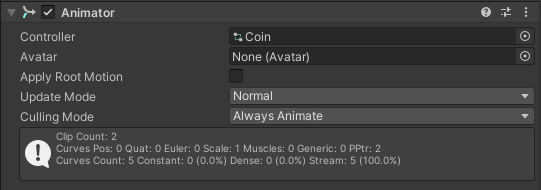


**Camera**

A Camera jeleníti meg a játékteret a játékos számára. Legalább egy mindig van egy Scene-ben.

**Animator**

Az Animator komponens biztosítja, hogy animációkat tudjunk hozzáadni az adott objektumhoz. Paraméterként vár egy Animator Controller-t, amely kezeli, hogy mely animációkat mikor, és hogyan játszhatja le, illetve a köztük lévő átmenetet is.



**Canvas**

A Canvas egy olyan terület, amelyre az összes UI elemet kell helyezni. Minden UI objektumot a Canvas objektum gyermekeként kell beállítani, mely rendelkezik egy Canvas komponenssel.

Új UI objektum, például egy Image (GameObject -> UI -> Image) létrehozásakor, automatikusan létrehozásra kerül egy Canvas objektum, ha még nincsen a Scene-ben. A UI objektum pedig ennek a Canvas-nak a gyermekeként jön létre.

A Canvas területe a Scene nézetben egy téglalapként jelenik meg.

A UI elemek a Canvason olyan sorrendben jelennek meg, ahogyan a Hierarchy ablakban szerepelnek. Az első gyermek lesz először megjelenítve, a második másodjára, és így tovább. Ha két elem fedi egymást akkor az utóbb megjelenített lesz felül.

**Text**

A Text komponens, amelyet Label-nek is neveznek, tartalmaz egy mezőt, melybe a megjelenítendő szöveget írhatjuk. Beállíthatjuk a betűtípust, stílust és méretet.

**Image**

Az Image objektum tartalmaz egy RectTransform és Image komponenst. Az Image komponenshez egy sprite-ot tudunk megadni, amelyet megjelenít.

**Button**

A Button tartalmaz egy OnClick UnityEvent, amellyel beállíthatjuk mi történjen, ha rákattintunk.

**Grid**

A Grid tulajdonképpen egy olyan háló, amely beborítja az egész játékteret, és megkönnyíti az egyes objektumok elhelyezését, például Rectangular Grid esetén a négyzet alakú csempe elemek könnyen elhelyezhetőek.

A komponens az egyes cellák helyzetét felelteti meg az objektum lokális helyzetének (pozíciójuk a komponens középpontjához képest). A Transform pedig ezeket a lokális pozíciókat konvertálja globálissá.

**Tilemap**

A Tilemap komponens tárolja és kezeli a kétdimenziós pályák készítésére szánt csempe elemeket. Továbbadja a rá helyezett csempékről a szükséges információkat, a többi kapcsolódó komponensnek, mint a Tilemap Renderer és a Tilemap Collider 2D.

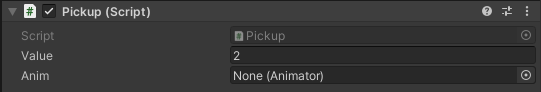
Első alkalommal szükséges letölteni, és importálni a 2D Tilemap Editor csomagot a Package Manager segítségével, mivel a Unity Editor nem tartalmazza alapértelmezetten.(kivéve, ha a unity hubba úgy hozzuk létre a projektet).

Amikor létrehozunk egy Tilemap objektumot, akkor a Grid objektum a Grid komponenssel automatikusan létrehozásra kerül, és a gyermekének állítja be a Tilemap objektumot.

**Script**

A szkriptek, és a kódolás alapvető eleme minden Unity-ben készült alkalmazásnak. Leggyakrabban a játékos által adott input-ot (pl. billentyűlenyomás), és a játékmenetet, eseményeket kezeli.

Például a szkriptekkel irányítjuk a játékost, kezeljük a tárgyait, animációit.



[Unity - Manual: Unity User Manual 2020.3 (LTS) (unity3d.com)](https://docs.unity3d.com/Manual/)

**A játék bemutatása**

**Asset store**

A Unity Asset Store egy olyan könyvtár/online áruház, mely ingyenes és fizetős asseteket tartalmaz, melyeket a Unity Technologies, vagy a közösség tagjai készítenek és tesznek közzé. Találunk ott többek közt textúra csomagokat, modelleket, animációkat, egész projekteket.

Miután az Asset Store-ból lementünk egy asset-et, azt az Editoron belül a Package Manager segítségével tudjuk letölteni és importálni.

A játékban szereplő asseteket a Unity Asset store-ból szereztem be, melyeket az alábbi táblázat tartalmaz.

|  |  |  |  |
| --- | --- | --- | --- |
| Sprite-ok és animációk | | Hangok | |
| Danil Chernyaev: 2D Platformer Tileset | [2D Platformer Tileset | 2D Environments | Unity Asset Store](https://assetstore.unity.com/packages/2d/environments/2d-platformer-tileset-173155) | MGWSoundDesign: Footstep(Snow and Grass) | [Footstep(Snow and Grass) | Audio Sound FX | Unity Asset Store](https://assetstore.unity.com/packages/audio/sound-fx/footstep-snow-and-grass-90678) |
| Black Hamme:  Fantasy Wooden GUI : Free | [Fantasy Wooden GUI : Free | 2D GUI | Unity Asset Store](https://assetstore.unity.com/packages/2d/gui/fantasy-wooden-gui-free-103811) | VGcomposer: Action RPG Music Free | [Action RPG Music Free | Audio Music | Unity Asset Store](https://assetstore.unity.com/packages/audio/music/action-rpg-music-free-85434) |
| PONETI: GUI Parts | [GUI Parts | 2D Icons | Unity Asset Store](https://assetstore.unity.com/packages/2d/gui/icons/gui-parts-159068) | Dustyroom: FREE Casual Game SFX Pack | [FREE Casual Game SFX Pack | Audio Sound FX | Unity Asset Store](https://assetstore.unity.com/packages/audio/sound-fx/free-casual-game-sfx-pack-54116) |

**Package manager**

A Package egy olyan konténer, amely különböző funkciókat, vagy asset-eket tartalmaz, például:

* Editor eszközöket és könyvtárakat, például szövegszerkesztőt, animáció megtekintőt.
* Runtime eszközöket és könyvtárakat, mint Physics API vagy Graphics pipeline.
* Asset gyűjteményeket, például textúrák, animációk.
* Projekt sablonokat.

Az Editorban a Package Manager ablak a következő módon érhető el: Window -> Package Manager.

**how to crate objects+add components**

Új objektum létrehozásához jobb egérgombbal kattintsunk a Hierarchy ablakba, és válasszuk ki a kívánt objektumot. ezután tetszőlegesen mozgatható az egérrel, a szülő-gyermek kapcsolatok beállítása érdekében.

Komponens hozzáadása kétféleképpen lehetséges. Először is az objektum kijelölése után (amely történhet a Hierarchy ablakban, vagy a Scene nézetben) az egyik lehetőségünk, hogy az Inspector ablakban a New Component gombra kattintunk. A másik lehetőség, hogy a menüszalag Components fülére kattintunk. Véleményem szerint az első lehetőség jobb, mivel ekkor egy keresőmezőben is lehetőségünk van kikeresni a kívánt komponenst.

A továbbiakban bemutatom az egyes Scene-eket megjelenési sorrend szerint, majd a bennük lévő objektumokat és azok viselkedését részletesebben.

**StartMenu**



\*csere majd?

A játék indításakor ez a Scene fogad minket. Objektumokat tekintve egy Camera, Canvas és egy Event System van benne. A Canvas-hoz hozzáadtam még egy Audio Source komponenst, melynek AudioClip paraméterének megadva a kiválasztott zenét automatikusan lejátssza.

A Canvas objektumnak két gyermeke van a Hierarchy ablakon belül, a Menu, mely tartalmazza azt a három Button objektumot, amely fogad minket, és a Choose, melyben a Play gomb megnyomása után elérhető elemek vannak.

A Menu objektumhoz adtam hozzá egy általam írt szkriptet, mely az egyes gombokra történő kattintásokat kezeli.

A Play gomb megnyomása aktiválja a Choose objektumot, mely azért felel, hogy megjelenítse a választható kasztokat.



Majd a Selectbtn gomb megnyomásakor betöltődik a következő Scene.

**VillageSpawn**

Ebben a Scene-ben a játékost fogadja egy NPC, aki általános információkat közöl.

A Non-Player Character rövidítése NPC.

Ezeknek általában két funkciója van egy játékban:

* + Életet lehel a játékba, a játéktér kitöltésével.
  + Valamiféle interakcióba lehet velük lépni.

Az általam készített játékban 2 NPC található. Az egyik a toronyőr, mely a játékost fogadja a játék kezdetekor, és megadja a kellő iránymutatást. A másik a kereskedő, amelynél vásárolni lehet. Ezek egy alap animációval rendelkeznek, mozogni nem tudnak.

Ez amolyan töltelék Scene, ami átmenetet biztosít a menüből a tényleges játékba. Itt már láthatjuk a választott karakterünket, dekorációs célt szolgáló objektumokat, és egy Grid objektumot. Innen mindig átmozgatásra kerül a többi Scene-be az egyes Camera, InvSys, Background, Canvas, Event System objektumok, és a játékos karaktere.

**Village**

Ez a Scene egy falut tartalmaz, melyben a kereskedő is található.

Ahogyan itt is, úgy a többi Scene-ről is elmondható, hogy a szélükön lévő objektumokkal ha érintkezik a játékos, akkor kerül át egy másik Scene-re. Ennek a mechanizmusnak a bemutatására később kerül sor.

**ForestBridge**

Itt olyan ellenfelekkel találkozhatunk, mint a pók, slime és darázs. Alapvetően nem tartalmaz ezeken kívül egyedi objektumokat, mechanizmusokat.

**Cave**

A barlangban csak pókokkal találkozhatunk. Itt rejtőzik a pókkirálynő, akire úgy is lehet tekinteni, mint egy mini boss. A csempék közt pedig tüskék is vannak. Rájuk van illesztve egy Collider, mellyel ha érintkezik a játékos, akkor sebződik és ugrik egy picit.

**Dungeon**

Itt már erős ellenfelekkel találkozhatunk. Csontvázakkal, melyek közt akad kardos, lándzsás és nyilas. A pálya végén pedig a boss van, melynek legyőzésével kijátszottuk ezt a kis minijátékot.

**UI**

A videojátékokban a HUD (heads-up display) vagy status bar az a módszer, amellyel az információ vizuálisan megjelenítésre kerül a játék felhasználói felületének részeként. Nevét a modern repülőgépekben használt head-up kijelzőkről kapta.

A felhasználói felületet maga a Canvas objektum biztosítja. Ezen találhatóak az egyes UI elemek.

A Játékomban ezeket az elemeket 3 csoportba lehet sorolni:

* Dizájn szerepet betöltők: Tipikusan ilyenek az egyes ablakok hátterét, alapját biztosító Image objektumok.
* Interakcióra képes elemek: Például a gombok.
* Tájékoztató szerepet betöltők: Ide sorolnám például az életerő jelző csíkot, és a Text objektumokat, melyek információt adnak át.



A játékos karakterének életerejét reprezentáló objektum, alapvetően egy slider. Értéke szkripteken keresztül állítódik, a játék kezdetekor, illetve a játékos sebzése vagy gyógyulásakor. A játékbeli fizetőeszközök számát mutatja a Coin\_show , amellyel a játékos rendelkezik.

Az InvBtn megjeleníti a játékos leltárát és statisztikáit. Szintlépéskor itt tudjuk növelni azokat.

A SettingsBtn lenyomása jeleníti meg a beállításokat. Itt tudnánk olyan beállításokat megadni, mint a játék nehézsége, vagy a zene hangereje, de ezek nem kerültek implementálásra.

Az ItemSlots a bal alsó sarokban látható négy rubrika. A játékos leltárában lévő első négy elemet jeleníti meg. A billentyűzeten lévő 1-4 gombokkal tudjuk felhasználni az adott elemeket.

A Canvas még további olyan objektumokat tartalmaz, melyek egy bizonyos feltétel teljesülése esetén jelennek meg, például szintlépéskor egy ezt jelző felirat, vagy éppen maga a bolt felülete.

Egy gombnyomásra történő esemény beállítása a következő módon történik:

\*\*képek

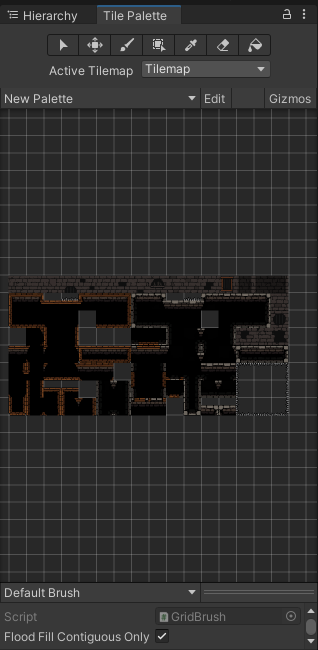
A Button komponenshez hozzáadunk egy OnClick eseményt a ’+’ gomb megnyomásával, majd az object paraméterre egérrel húzzuk a Hierarchy ablakból azt az objektumot, amelyen a szkript található. Ezután a legördülő listából kiválasztjuk a kívánt metódust. Csak akkor tudjuk kiválasztani, ha public.

**grid, tilemap**

A Start Menu-n kívül mindegyik Scene-ben megtalálható. Tulajdonképpen a kétdimenziós játéktér alapja. Létrehozáskor a rectangulart választottam ki ezért a grid tulajdonképpen egy négyzetrácsos háló lesz, melyre pakolhatjuk a csempe elemeket.

Ahhoz, hogy létrehozzunk, szerkesszünk, és kiválasszuk a festeni kívánt csempe elemet, a Tile Palette ablakot kell megnyitnunk. (menu: Window > 2D > Tile Palette)

[Unity - Manual: Tilemap (unity3d.com)](https://docs.unity3d.com/Manual/class-Tilemap.html)



Az ablak felső részén találhatóak olyan eszközök, mint például a kiválasztás, radírozás, festés.

Miután pedig a Creat New Palette gombra kattintás után kiválasztuk a beimportált TileSet assetünket, egy szintén négyzetrácsos hálón látjuk a rendelkezésünkre álló csempekészletet. Amíg ezzel dolgozunk érdemes az ablakot rögzíteni a Hierarchy ablak mellé, nehogy bezáródjon.

Hogy a Scene nézetben található négyzetrácsok mérete megegyezzen a csempék méretével, érdemes ennek megfelelően állítani a méretet. Én fordítva dolgoztam, ami azt jelenti, hogy minden egyes beimportált sprite asset pixels per unit tulajdonságát 256-ra állítottam be.

Ezután a pálya készítés igazán egyszerű. A Tile Palette-en kiválasztjuk a csempét egy kattintással, majd a Scene nézetbe katintással helyezzük le.

Miután elkészítettük a pályát. A Hierarchy ablakban kiválasztjuk a Tilemap objektumot és az Inspector ablakban hozzá kell adnunk az alábbi komponenseket:

* Tilemap Collider 2D:

Biztosítja, hogy a fizikai szimulációban résztvevő objektumok, ne zuhanjanak le, tartsa meg őket. Ennek megfelelően Az Inspector ablak tetején adtam hozzá egy új Layert, amit groundnak neveztem el.

* Composite Collider:

A Tilemap Collider 2D used by composite paraméterét jelöljük be. A Tilemap Collider 2D minden egyes csempéhez külön collidert adott. A Composite Collider ezen collidereket egyesíti.

* Rigidbody 2D:

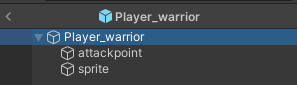
Composite Collider-el együtt automatikusan hozzáadásra kerül. Mivel nem szeretnénk, hogy a játék futása során a pálya is lezuhanjon a gravitáció hatására, ezért ennek a komponensnek a Body Type paraméterét Static-ra állítottam.

Érdemes megemlíteni, hogy van egy ún. Simulated bejelölhető tulajdonsága, amely ha nincs bepipálva, nem vesz részt a komponenshez tartozó objektum a szimulációban. Ekkor nem lenne aktív a Collider sem, így átzuhannának rajta az objektumok.

**Karakter objektumok**

**Játékos karakterek**

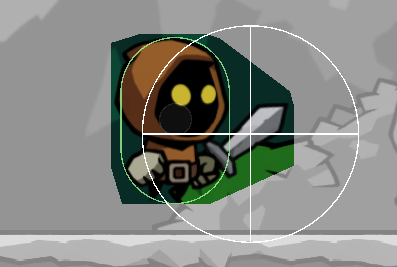
A játékos karakter objektumok felépítése a következő:



Van egy szülő objektum, mely az adott karakter nevét hordozza. Ezen objektumra vannak beállítva a fizikai szimulációhoz szükséges komponensek, mint a RigidBody 2D, CapsuleCollider 2D és az adott karakter főleg irányítására használatos szkript.

Ezen objektum gyermekei a következőek:

Attackpoint: Csak egy Transform-ot tartalmaz. Warrior esetén ez lesz annak a területnek a középpontja, amelybe ha támadáskor ellenfél található, akkor megsebzi azt.



Hunter és Mage esetén ez az objektum adja meg azt a pontot, ahol támadáskor a nyílvessző, vagy a tűzgolyó inicializálásra kerül.

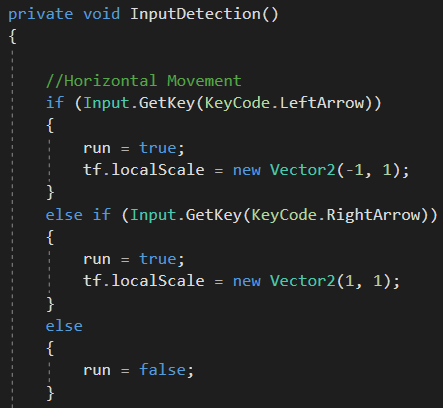
Sprite: Ezen objektum felelős a karakter vizuális megjelenítéséért és animációjáért. Ennek megfelelően rendelkezik a Sprite Renderer és Animator komponenssel, illetve az animációkat kezelő szkripttel. Mage esetén nem létezik ez az objektum, ezek a komponensek a szülő objektumban találhatóak. Ennek okát később fedem majd fel, amikor az animálásról fogok írni.

**Player\_Controller szkript**

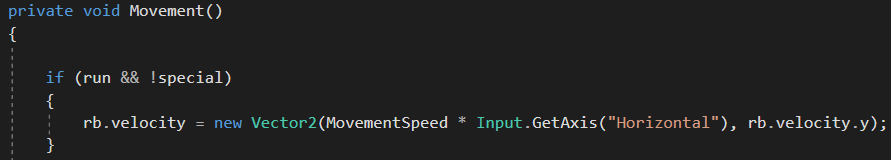
A legtöbb osztály, amelyet létrehoztam a Mono\_Behaviour osztály gyermeke. A szkript tartalmazza a karakter tulajdonságpontjainak értékét és olyan metódusokat, melyek például a mozgásért felelnek.

A játékos karakterét többféleképpen is mozgásra lehet bírni. Én az objektumhoz tartozó Rigidbody komponens velocity értékét módosítottam, a kapott bemeneti értékeknek megfelelően.

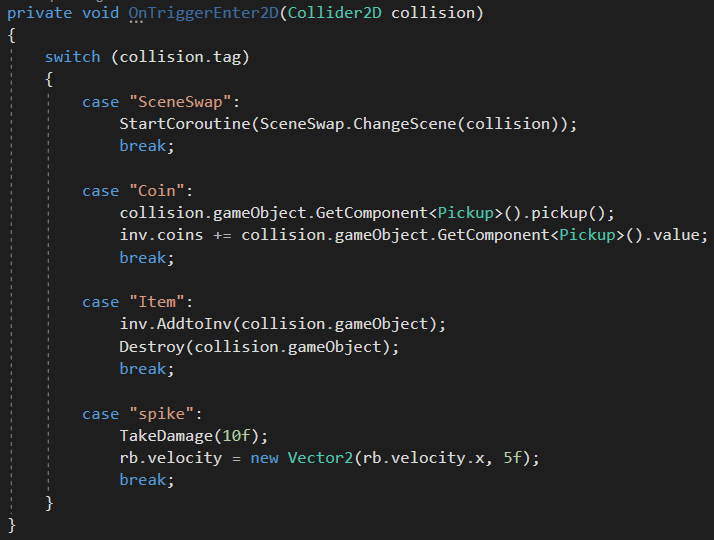
A horizontális irányban történő mozgás például így néz ki:



A bemeneti értékeket az InputDetection metódus fogadja. Itt gondoskodok arról is, hogy a sprite menetirány felé nézzen.



A velocity értékének módosítását pedig a Movement metódus végzi.



Az OnTriggerEnter2D metódus akkor kerül meghívásra, amikor olyan objektummal ütközik a játékos, mely Collider-ének Trigger opciója be van pipálva.

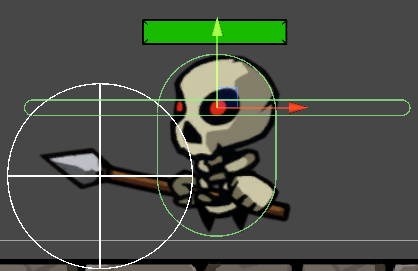
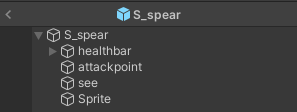
A csak a Warrior kaszt álltal használt metódus az Attack, és az Update metóduson belülil egyik if ág az egyes támadásokat hajtják végre.

**Animator\_Controller szkript**

Az Animator\_Controller az előbb említett InputDetection metódushoz hasonló módon, várja a bemeneti értékeket és azok alapján hívja meg a ChangeAnimationState metódust, amely beállítja a szükséges animációt. Továbbá olyan függvényeket tartalmaz, melyeket az animáció során hívok meg. Ez akkor lényeges, amikor például a nyílvessző inicializálását az animáció adott képkockájánál kell végezni.

**Ellenséges karakterek**

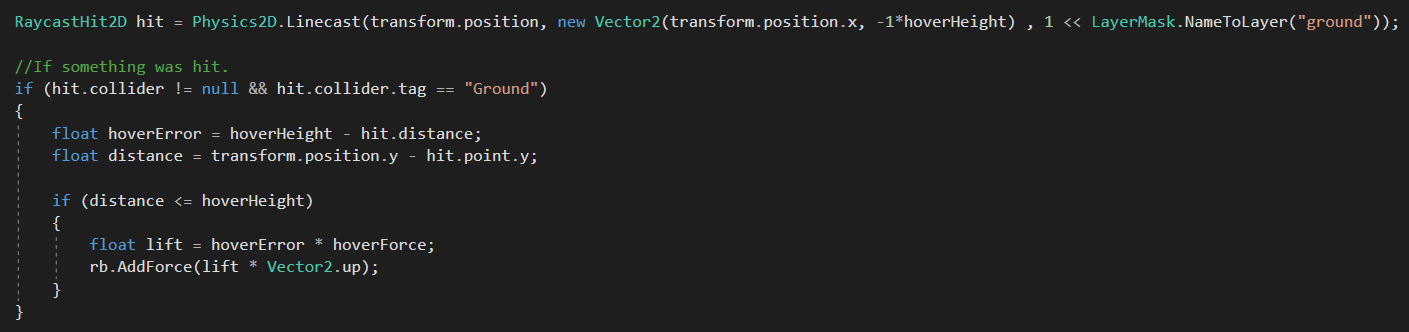
Ezen objektumok felépítése hasonló a játékos karakterekhez, viszont itt a gyermekek között megtaláljuk még a see és a healthbar objektumokat. A see egy Collidert tartalmaz, melynek funkciója, hogy érzékelje az esetleges ütközést a pályaelemekkel (ground), így ugrásra késztetve a karaktert. Az életerőcsíkot jelen esetben sprite-ok alkotják. Sebződéskor a zöld színű sprite szélessége állítódik.



**Enemy szkriptek**

Az Enemy scriptből négy osztály öröklődik.

A két távolsági harcos karakter itt a darázs és a nyilas csontváz. A nyilas csontváz az Enemy\_Ranged osztályt használja. Ez a karaktek nem végez mozgást, így csak az olyan alap metódusok vannak benne, vmint például a TakeDamage. A darázs speciális mozgást végez, hiszen lebeg, amely a repülés látszatát kelti, ezért került új osztályba.



A Physics2D.Linecast egy vonalat húz a föld felé, és ennek megfelelően állítódik a magassága.

Mindkét karakter támadása hasonlóan a játékos karakterekhez lényegében az animációjukat kezelő szkriptben hajtódik végre.

\*darázs támadás kódrész(lehet inkább projectileba)

#A csontváz nyilvesszőjét csak a játékos karakterének Collider-e triggereli, tehát csak őt sebzi.

Az ellenfeleken található szkriptek tartalmazzák az alapvető statisztikai értékeket, mint az életerőt, vagy a mozgási sebességet. Az objektumok köré rajzolódik egy láthatatlan kör, melynek sugara a detection\_range paraméter. Amennyiben a játékos ezen körön belülre kerül, az ellenfél elkezdi követni őt. Egyszerűen megnézi, hogy önmagához képest merre van a játékos, és addig megy, amíg el nem érte a dist (distance) paraméter által megadott, közöttük lévő távolságot. Miután megállt, megkezdődik a támadás az animáció lejátszásával.

**Projectiles**

Az Arrow, Sting, FireBall objektumok. A távolsági harcos karakterek lövedékeként kerülnek inicializálásra támadáskor. A viselkedésüket meghatározó szkriptet és a Rigidbody 2D, Collider 2D és Sprite Renderer komponenseket tartalmazzák.

A speciális támadások hatása is ezen szkripten belül aktiválódik. Megnövelt támadási értéket kapnak, illetve a nyílvessző úgy állítódik be, hogy ne semmisüljön meg az első ütközéskor, minden ellenfelet sebezzen végig amelyen áthalad.

**Items**

**Coin**

Gyűjthető objektum, mely fizetőeszközként szolgál. Egy Collider-rel és Animator Controller-el, illetve az interakciót menedzselő szkripttel rendelkezik.

\*\*esetleg érték felülírás kódrész

**Potions**

Felhasználható objektum, mely a játékos karakterének tulajdonságpontjait növelik.

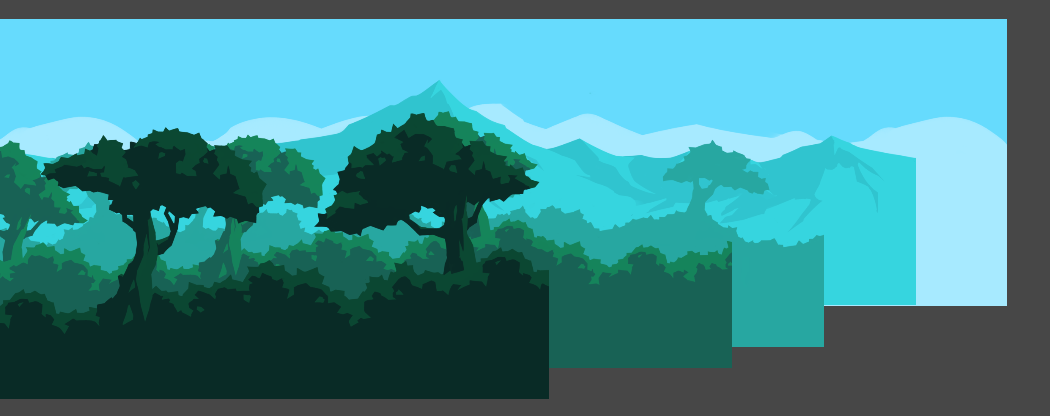
Négy féle van:

* Health potion: 20 életerőpontot tölt vissza.
* Agility potion: Növeli a mozgási sebességet és az ugrás magasságát.
* Strenth potion: Növeli az alap és a speciális támadás sebzését és a támadási sebességet.
* Special potion: Növeli a speciális támadás sebzését és támadási sebességét. Warrior esetén pedig a területét is.

\*\*felhasználati kódrész pl

**Background**

Háttérként az úgynevezett parralax backgroundot alkalmaztam. Ennek a lényege, hogy több rétegben helyezkednek el egymáson az egyes háttérelemek, és a Camera objektummal együtt mozognak, eltérő sebességgel.



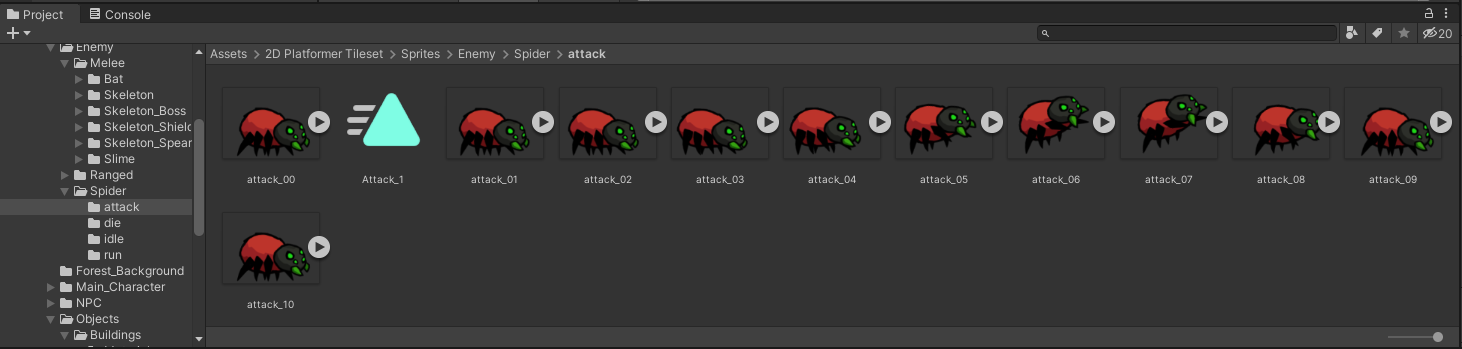
Ettől olyan hatást kelt, mintha tényleg lenne mélysége, mintha valóban távol lennének a távoli, és közel a közeli elemek. Maga az objekum egy szülő, amely azon gyermekeket fogja össze, amelyek az egyes rétegeket és a mozgatásukat végző szkriptet tartalmazzák.

**Animation**

Két karakter animálását mutatnám be. Az egyik a mágus játékos karakter, melyet egyedi módon animáltam, a másik pedig egy ellenséges karakter a pók.

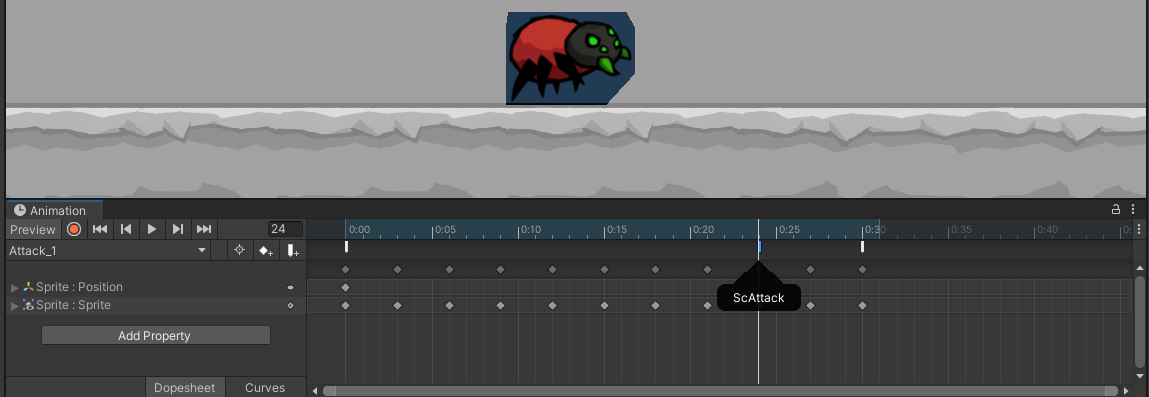
**Sprite animation**

Az olyan asset csomagok, mint amilyet és is használok, általában járnak az animációkhoz használható sprite-ok.



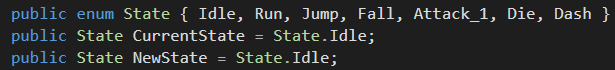
A Project ablakban létre kell hoznunk egy új animációt (jobbklikk -> create -> animation), majd ráhúzni az egérrel az Animator komponenst tartalmazó objektumra. Esetünkben a Sprite objektumra.

Ezután megnyitjuk az Animation ablakot, és az adott sprite-okat egyszerűen ráhúzzuk.



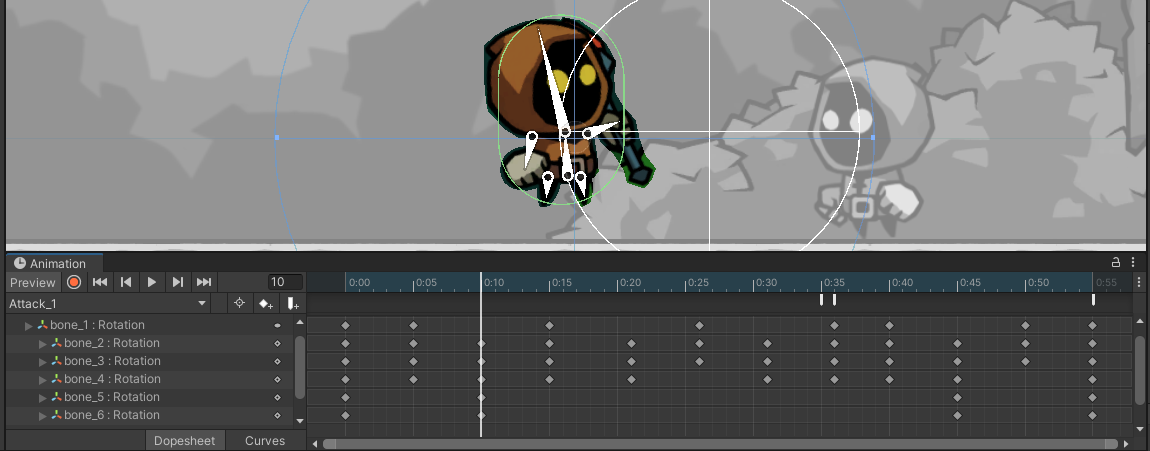
Az Add Event gomb segítségével állíthatunk be olyan eseményeket, mint például a támadást, adott képkockánál.

Esetemben fontos hogy a létrehozott animáció neve megegyezzen valamely State enum értékkel, hiszen úgy adom át a ChangeAnimationState metódusnak, mely az alapján játssza le az animációt.

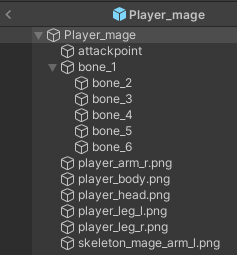


**Skeletal animation**

A mágus karakter animálása a többi karaktertől eltérő módon nem a sprite-ok egymásután állításával történt, hanem úgynevezett skeletal animation-t alkalmaztam. Ennek lényege, hogy csakúgy, mint az emberi test vázát, csontok alkotják, melyekre a karakter ’húsa’ van rögzítve. Ezáltal képkockánként végighaladva, manuálisan mozgatva az egyes csontokat, mozognak a hozzájuk tartozó sprite elemek.



Ehhez egy psb formátumú képet kell beimportálnunk, melyet például photoshopban hozunk létre oly módon, hogy minden egyes sprite elemet (fej, kar, stb.) külön rétegekre illesztünk be. Ezt beimportálva a Unity szét tudja vágni az egyes rétegeket és hozzájuk tudja rendelni a csontokat.



**Leltár**

A leltár működéséhez három szkript ’csapatmunkájára’ van szükség:

**Inventory**

Olyan metódusokat tartalmaz, amelyek például hozzáadják a leltárhoz a felvett objektumot, vagy hozzáigazítja az adott rubrikához a pozícióját.

**DragDrop**

Az egyes felvehető objektumok (esetemben csak a bájitalok) tartalmazzák komponensként.

Ennek az osztálynak olyan interfészeket állítottam be szülőként, amelyekben definiált metódusok implementálásával, a Canvas objektumon, a kurzorral történő eseményeket adhatjuk meg, illetve követhetjük nyomon. Ilyen események például a kattintás, húzás, gomb felengedése.

**ItemSlot**

A leltárban lévő rubrikákban található, mint komponens. Itt az IDropHandler interfész OnDrop metódusa van implementálva, tehát ha felette engedjük fel az egérgombot, akkor meghívásra kerül.

Tekinthetünk erre, mint az adott objektum rubrikához való rendelésének backend része, az Inventory szkript pedig a frontend.

**StatSys**

**Scene Controller**

**Prefabs**

Unity’s Prefab system allows you to create, configure, and store a GameObject complete with all its components, property values, and child GameObjects as a reusable Asset. The Prefab Asset acts as a template from which you can create new Prefab instances in the Scene.

When you want to reuse a GameObject configured in a particular way – like a non-player character (NPC), prop or piece of scenery – in multiple places in your Scene, or across multiple Scenes in your Project, you should convert it to a Prefab. This is better than simply copying and pasting the GameObject, because the Prefab system allows you to automatically keep all the copies in sync.

Any edits that you make to a Prefab Asset are automatically reflected in the instances of that Prefab, allowing you to easily make broad changes across your whole Project without having to repeatedly make the same edit to every copy of the Asset.

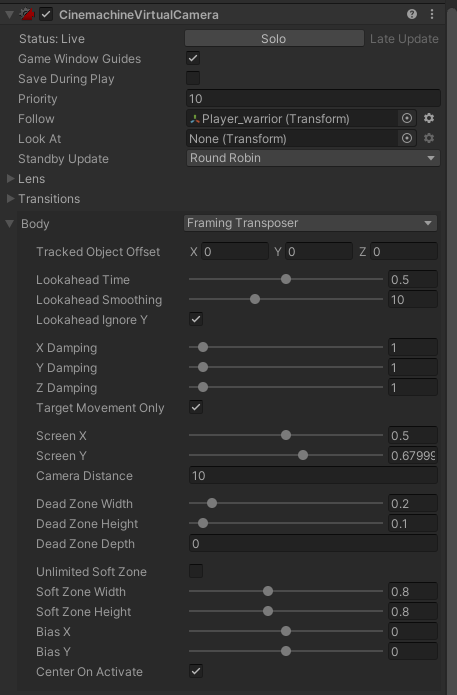
--------

A Unity prefab (előregyártott) rendszere lehetővé teszi egy objektum létrehozását, konfigurálását és tárolását az összes összetevőjével, tulajdonságértékével és gyermek objektumaival együtt, újrafelhasználható eszközként. A prefab eszköz sablonként működik, amelyből új előregyártott példányokat hozhat létre a jelenetben.

Ha egy bizonyos módon konfigurált objektumot szeretnénk újra felhasználni a jelenet több helyén, vagy a projekt több jelenetében, akkor érdemes prefab-bé konvertálni. Ez jobb, mint a GameObject egyszerű másolása és beillesztése, mert a Prefab rendszer lehetővé teszi, hogy az összes másolatot automatikusan szinkronban tartsa.

Az előregyártott eszközön végzett minden szerkesztés automatikusan megjelenik az adott előregyártott elem példányaiban, lehetővé téve, hogy könnyen végezzen széles körű változtatásokat a teljes projektben anélkül, hogy ismételten ugyanazt a szerkesztést kellene elvégeznie az eszköz minden példányán.

**Virtual Camera**

****

A Cinemachine Virtual Camera, ahogyan a neve is sugallja, egy virtuális kamerát ad a Scene-hez. Ez tulajdonképp már a meglévő Main Camera mozgását vezérli. Simább, letisztultabb hatást kelt a karakter mozgatásakor.

A sárga kis négyzet úgy viselkedik, mint egy joystick. Ahogy vezéreljük a játékost a billentyűzettel, úgy megy az is jobbra, balra, fel, le. Amíg a színtelen középső négyzetben, azaz a Dead Zone-ban van, addig nem mozdul a kamera semerre. A kék színnel jelölt terület a Soft Zone. Amikor ebbe kerül a sárga jel, akkor kezd el mozogni a kamera, a beállított átmeneti sebességgel. A piros terület a határt jelöli, amelynél már nem mehet tovább a sárga jelölő a kamera egyenértékű mozgatása nélkül.

Tesztelés.

**Összefoglalás**

Megállapítások és következtetések, tapasztalatok.