**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 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 bemutatásával kezdem. Majd a Unity és a használt eszközök bemutatása után, végül a játékomról írok részletesebben.

**jatekrol par szo**

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**

[Game engine - Wikipedia](https://en.wikipedia.org/wiki/Game_engine)

Az első videójátékot egy amerikai fizikus, Willy Higinbotham készítette 1958-ban. A Tennis for Two, egy teniszjátékot szimulált le. Grafikus megjelenítésre oszcilloszkópot használt, és egy analóg vezérlővel lehetett elütni a labdát. A híres Pong elődjeként tartják számon.(labjegyzetbe spacewart)

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 helyett.

Other platforms had more leeway, but even when the display was not a concern, memory constraints usually sabotaged attempts to create the data-heavy design that an engine needs. Even on more accommodating platforms, very little could be reused between games.

Since the golden age of arcade video games, it became common for video game companies to develop in-house game engines for use with first-party software.

There were several 2D game creation systems produced in the 1980s for independent video game development. These include Pinball Construction Set (1983), ASCII's War Game Construction Kit (1983),[9] Thunder Force Construction (1984),[10] Adventure Construction Set (1984), Garry Kitchen's GameMaker (1985), Wargame Construction Set (1986), Shoot-'Em-Up Construction Kit (1987), Arcade Game Construction Kit (1988), and most popularly ASCII's RPG Maker engines from 1998 onwards. Klik & Play (1994) is another legacy offering that's still available.[11]

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

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 tervekre épültek, melyek kihasználták az extra erőforrásokat.

A legtöbb játék ennek megfelelően kemény kódolt szabálykészlettel, néhány pályaszinttel és grafikus adatokkal.

Az árkádgépek aranykora óta 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.

Az 1980-as években több kétdimenziós játékkészítő rendszert hoztak létre önálló videójáték fejlesztésre. Ezeket construction kits-nek (építőkészleteknek) nevezték. Ilyen volt például a Pinball Construction Set (1983).

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

The term "game engine" arose in the mid-1990s, especially in connection with 3D games such as first-person shooters with a first-person shooter engine. Such was the popularity of Id Software's Doom and Quake games that, rather than work from scratch, other developers licensed the core portions of the software and designed their own graphics, characters, weapons and levels—the "game content" or "game assets". Separation of game-specific rules and data from basic concepts like collision detection and game entity meant that teams could grow and specialize.[11]

Before the 1993 release of its upcoming game DOOM, id Software issued a news release. It promised that DOOM would “push back the boundaries of what was thought possible” on computers. This press release is a remarkable document. It summarized stunning innovations in technology, gameplay, distribution, and content creation. It also introduced a term, the “DOOM engine.” This term described the technology under the hood of id's latest game software. The news release promised a new kind of “open game” and sure enough, id's game engine technology became the motor of a new computer game industry. The “Invention of the Game Engine” was only half the story. John Carmack, the lead programmer at id, did not just create a new kind of software, as if that were not enough. He also conceived and executed a new way of organizing the components of computer games by separating execution of core functionality by the game engine from the creative assets that filled the play space and content of a specific game title. Jason Gregory in his book on game engines writes, "DOOM was architected with a relatively well-defined separation between its core software components (such as the three-dimensional Game Engines and Game History graphics rendering system) and the arts assets, game worlds, and rules of play that comprised the player’s gaming experience." (Gregory 11) [2014-Lowood.pdf (kinephanos.ca)](https://www.kinephanos.ca/Revue_files/2014-Lowood.pdf)

In 1993, ID software developed Doom engine , which is not a 3D engine at all, but had capacity to represent objects, characters and whole level map by 2D sprite representation. Rendering was very fast and it needs 386 based PC with standard VGA support to run. Though it was a 2D engine, the illusion created by the developer made it a 3D title. [(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)

Later games, such as id Software's Quake III Arena and Epic Games's 1998 Unreal were designed with this approach in mind, with the engine and content developed separately. The practice of licensing such technology has proved to be a useful auxiliary revenue stream for some game developers, as one license for a high-end commercial game engine can range from US$10,000 to millions of dollars, and the number of licensees can reach several dozen companies, as seen with the Unreal Engine. At the very least, reusable engines make developing game sequels faster and easier, which is a valuable advantage in the competitive video game industry. While there was a strong rivalry between Epic and id around 2000, since then Epic's Unreal Engine has been far more popular than id Tech 4 and its successor id Tech 5.[12]

Modern game engines are some of the most complex applications written, often featuring dozens of finely tuned systems interacting to ensure a precisely controlled user experience. The continued evolution of game engines has created a strong separation between rendering, scripting, artwork, and level design. It is now common, for example, for a typical game development team to have several times as many artists as actual programmers.[13]

First-person shooter games remain the predominant users of third-party game engines, but they are now also being used in other genres. For example, the role-playing video game The Elder Scrolls III: Morrowind and the MMORPG Dark Age of Camelot are based on the Gamebryo engine, and the MMORPG Lineage II is based on the Unreal Engine. Game engines are used for games originally developed for home consoles as well; for example, the RenderWare engine is used in the Grand Theft Auto and Burnout franchises.

Threading is taking on more importance due to modern multi-core systems (e.g. Cell) and increased demands in realism. Typical threads involve rendering, streaming, audio, and physics. Racing games have typically been at the forefront of threading with the physics engine running in a separate thread long before other core subsystems were moved, partly because rendering and related tasks need updating at only 30–60 Hz. For example, on PlayStation 3, physics ran in Need For Speed at 100 Hz versus Forza Motorsport 2 at 360 Hz.

Although the term was first used in the 1990s, there are a few earlier systems in the 1980s that are also considered to be game engines, such as Sierra's Adventure Game Interpreter (AGI) and SCI systems, LucasArts' SCUMM system and Incentive Software's Freescape engine (in 1986[14]). Unlike most modern game engines, these game engines were never used in any third-party products (except for the SCUMM system which was licensed to and used by Humongous Entertainment).

As game engine technology matures and becomes more user-friendly, the application of game engines has broadened in scope. They are now being used for serious games: visualization, training, medical, and military simulation applications, with the CryEngine being one example.[15] To facilitate this accessibility, new hardware platforms are now being targeted by game engines, including mobile phones (e.g. Android phones, iPhone) and web browsers (e.g. WebGL, Shockwave, Flash, Trinigy's WebVision, Silverlight, Unity Web Player, O3D and pure DHTML).[16]

Additionally, more game engines are being built upon higher level languages such as Java and C#/.NET (e.g. TorqueX, and Visual3D.NET), Python (Panda3D), or Lua Script (Leadwerks). As most 3D rich games are now mostly GPU-limited (i.e. limited by the power of the graphics card), the potential slowdown due to translation overheads of higher level languages becomes negligible, while the productivity gains offered by these languages work to the game engine developers' benefit.[17] These recent trends are being propelled by companies such as Microsoft to support Indie game development. Microsoft developed XNA as the SDK of choice for all video games released on Xbox and related products. This includes the Xbox Live Indie Games[18] channel designed specifically for smaller developers who don't have the extensive resources necessary to box games for sale on retail shelves. It is becoming easier and cheaper than ever to develop game engines for platforms that support managed frameworks.[19]

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

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

Back then, video and arcade games were considered by most adults to be nothing more than toys, and the software that made them tick was highly specialized to both the game in question and the hardware on which it ran.

Today, games are a multi-billion-dollar mainstream industry rivaling Hollywood in size and popularity. And the software that drives these now-ubiquitous three-dimensional worlds—game engines like id Software’s Quake and Doom engines, Epic Games’ Unreal Engine 4, Valve’s Source engine and the Unity game engine— have become fully featured reusable software development kits that can be licensed and used to build almost any game imaginable.

While game engines vary widely in the details of their architecture and implementation, recognizable coarse-grained patterns are emerging across both publicly licensed game engines and their proprietary in-house counterparts.

Virtually all game engines contain a familiar set of core components, including the rendering engine, the collision and physics engine, the animation system, the audio system, the game world object model, the artificial intelligence system and so on. Within each of these components, a relatively small number of semi-standard design alternatives are also beginning to emerge

The term “game engine” arose in the mid-1990s in reference to first-person shooter (FPS) games like the insanely popular Doom by id Software. Doom was architected with a reasonably well-defined separation between its core software components (such as the three-dimensional graphics rendering system, the collision detection system or the audio system) and the art assets, game worlds and rules of play that comprised the player’s gaming experience.

The value of this separation became evident as developers began licensing games and retooling them into new products by creating new art, world layouts, weapons, characters, vehicles and game rules with only minimal changes to the “engine” software. This marked the birth of the “mod community”—a group of individual gamers and small independent studios that built new games by modifying existing games, using free toolkits provided by the original developers.

Towards the end of the 1990s, some games like Quake III Arena and Unreal were designed with reuse and “modding” in mind. Engines were made highly customizable via scripting languages like id’s Quake C, and engine licensing began to be a viable secondary revenue stream for the developers who created them. Today, game developers can license a game engine and reuse significant portions of its key software components in order to build games.

While this practice still involves considerable investment in custom software engineering, it can be much more economical than developing all of the core engine components in-house.

**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.

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

**Unity**

Az olyan általános osztályok összességét, amelyek vélhetoleg sok különféle játék- ˝ ban felhasználhatók, együttesen játék könyvtárnak, vagy játékmotornak (game engine) nevezzük.

A Unity a Unity Technologies által fejlesztett 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.

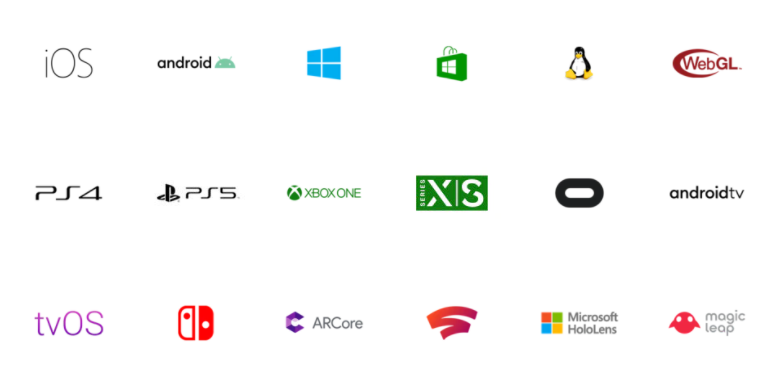
Olyan funkcionalitásokkal rendelkezik, melyek képesek a játékok különböző aspektusait kezelni, mint a

* 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 Unity 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.

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.

**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 az alábbiak:

* C#
* F#
* 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**

**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.

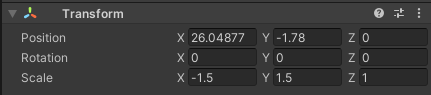
(H)A Status szalag értesítéseket jelenít meg egyes folyamatokról, illetve gyors elérést biztosít a hozzájuk kapcsolódó eszközökhöz és beállításokhoz.

**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.



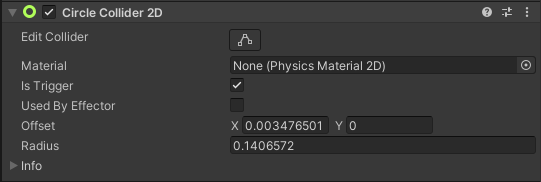
**RectTransform**

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**

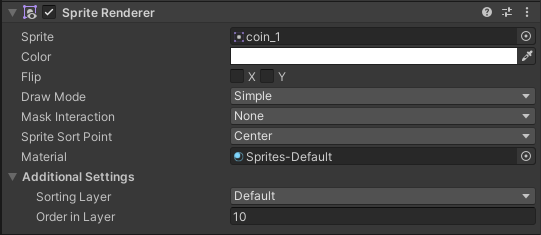
A Rigidbody komponens által vehet részt egy objektum a fizikai szimulációban. Az egyes karaktereket is ezen komponensen keresztül tudja irányítani a játékos szkript segítségével.

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.

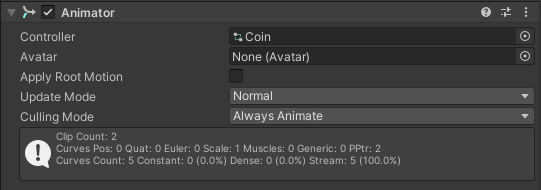


**Camera**

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.

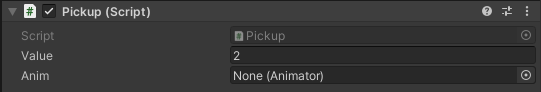
Első alkalommal szükséges letölteni, és importálni a 2D Tilemap Editor csomagot a Package Manager segítségével, mivel a Unity Editor nem tartalmazza alapértelmezetten.(kivéve, ha a unity hubba úgy hozzuk létre a projektet).

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 játék bemutatását Scene-enként végzem, megjelenési sorrend szerint.

1. **grid, tilemap**

Tulajdonléppen a kétdimenziós játéktér létrehozásának az alapja.

Létrehozáskor a rectangulart választottuk ki ezért a grid tulajdonképpen egy négyzetrácsos háló lesz, melyre pakolhatjuk a csempe elemeket.

Ahoz, 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)

\*\*kép tile palettrol

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.

Szükségességét al alábbi kép szemlélteti.

\*\*összehasonlító kép

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.

1. **karakterek**

**player**

A karakter objektumok felépítése a következő:

\*kép hierarchy-ról

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: 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 komponenssel és 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.

**enemy melee, enemy ranged**

Ezen objektumok felépítése hasonló a játékos karakterekhez, viszont itt a gyermekek között megtaláljuk még a See objektumot. Ez egy Collidert tartalmaz, melynek funkciója, hogy érzékelje az esetleges ütközést a pályaelemekkel (ground), így ugrásra késztetve a karaktert.

**npcs**

A Non Playable 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.

1. **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

1. **items**

**coin**

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

**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.

1. **background**

Háttérként az úgynevezett paralax 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 háttérelemeket és a mozgatásukat végző szkriptet tartalmazzák.

1. **UI felületek**

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.

1. **sima sprites, decor**

Egyszerű objektumok, melyek csak dekoráció céljából vannak jelen. Ezek csak a Sprite Renderert tartalmazzák megjelenítés céljából. Ilyenek például az épületek vagy a növényzet.