FELADATKIÍRÁS

A feladatkiírást a **tanszék saját előírása szerint** vagy a tanszéki adminisztrációban lehet átvenni, és a tanszéki pecséttel ellátott, a tanszékvezető által aláírt lapot kell belefűzni a leadott munkába, vagy a tanszékvezető által elektronikusan jóváhagyott feladatkiírást kell a Diplomaterv Portálról letölteni és a leadott munkába belefűzni (ezen oldal HELYETT, ez az oldal csak útmutatás). Az elektronikusan feltöltött dolgozatban már nem kell megismételni a feladatkiírást.



Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Rittgasszer Ákos

Dungeon Generátor Unreal megjelenítéssel

Konzulens

Rajacsics Tamás

BUDAPEST, 2024

Tartalomjegyzék

[Összefoglaló 6](#_Toc160286082)

[Abstract 7](#_Toc160286083)

[1 Bevezetés 8](#_Toc160286084)

[1.1 Játékpálya 8](#_Toc160286085)

[1.2 Dungeon 9](#_Toc160286086)

[2 Pályagenerálás 11](#_Toc160286087)

[3 Procedurális generálás 12](#_Toc160286088)

[3.1 Procedurális algoritmusok a játékfejlesztésben 12](#_Toc160286089)

[4 Unreal Engine 14](#_Toc160286090)

[5 Specifikáció 15](#_Toc160286091)

[6 Tervezés 16](#_Toc160286092)

[7 Algoritmusok 17](#_Toc160286093)

[8 Dungeon generátor 18](#_Toc160286094)

[9 Dungeon megjelenítő 19](#_Toc160286095)

[10 Továbbfejlesztés 20](#_Toc160286096)

[11 Összegzés 21](#_Toc160286097)

[12 Bevezetés 22](#_Toc160286098)

[12.1 Formázási tudnivalók 22](#_Toc160286099)

[12.1.1 Címsorok 22](#_Toc160286100)

[12.1.2 Képek 22](#_Toc160286101)

[12.1.3 Kódrészletek 22](#_Toc160286102)

[12.1.4 Irodalomjegyzék 23](#_Toc160286103)

[13 Utolsó simítások 24](#_Toc160286104)

[14 Irodalomjegyzék 25](#_Toc160286105)

[Függelék 26](#_Toc160286106)

Hallgatói nyilatkozat

Alulírott **Rittgasszer Ákos**, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2024. 03. 11.

...…………………………………………….

Rittgasszer Ákos

Összefoglaló

A modern játékok több bonyolult részből állnak. Az jó játékélményhez minden egyes komponensre kiemelt figyelmet kell fordítani. Az egyik ilyen alapvető elem a pálya, melyben gyakran előre megtervezett építőelemeket alkotnak kézzel, ám előfordul, hogy véletlenszerű generálásra támaszkodnak. A "dungeon" kifejezés olyan rendszert jelöl, amelyben egymástól fallal elválasztott szobákat ajtók és folyosók kötnek össze. Ezek a folyosók és ajtók egyfajta labirintust hoznak létre. Egy dungeon élvezhetősége nagyban tud nőni, ha egy véletlenszerű algoritmussal készítjük őket, mert így nehezebben unhatóak meg.

Én is egy ilyen rendszert véletlenszerűen generáló algoritmust szeretnék készíteni. Az megközelítésem abban különleges, hogy a szobák nem szabályos formájúak mint általában, hanem bármilyen sokszög alakját felvehetik. Ez a változatosság remélhetőleg fokozza az élményt a felfedezés során.

A generálás folyamatát egy Unreal Engine segítségével készült megjelenítő teszi teljessé. Ez a megoldás nemcsak segíti az algoritmus fejlesztését és tesztelését, hanem lehetővé teszi a generált dungeonok gyakorlati hasznosítását is. A motor által nyújtott fizikai alapok segítségével a létrehozott pályák valósághűen működnek, ami optimális kiindulási pontot jelent a játékfejlesztők számára.

Abstract

Modern games consist of multiple complicated components. To ensure a great gaming experience, each component requires meticulous attention. One such fundamental element is the level design, where often pre-designed elements are manually crafted, although random generation is also employed at times. The term "dungeon" refers to a system where rooms, separated by walls, are interconnected by doors and corridors. These corridors and doors create a labyrinth. The enjoyability of dungeons can greatly increased if thet are created with a random algorithm, as they become less predictable and therefore less prone to becoming boring.

I aim to develop a random generation algorithm for such systems. What sets my approach apart is that the rooms can take on irregular shapes rather than standard forms, enhancing the variety and, hopefully, intensifying the experience of exploration.

The generation process is complemented by a visualizer created with Unreal Engine. This not only helps in the development and testing of the algorithm but also enables the practical utilization of the generated dungeons. Thanks for the engine, the created levels operate realistically, offering an optimal starting point for game developers.

# Bevezetés

A játékfejlesztők igyekeznek eleget tenni a játékosok folyamatosan növekvő igényeinek. A gyors fejlődés ugyanúgy köszönhető a hardverek egyre növekvő számítási kapacitásának, mint a játékmotorok folyamatos fejlődésének.

A modern játékmotoroknak köszönhetően a modern játékok készítése egyre csökken a szoftverfejlesztők jelentősége. Emellett egyre nagyobb hangsúly helyeződik a tervezők munkájára. Itt két meghatározó tényező van a megjelenés mellett, ami leginkább befolyásolja a játékélményt.

Az egyik a játék dinamikája. Ez azt határozza meg, hogy a játékos milyen módon lép interakcióba a játékbeli környezetével és hogy milyen célok vezérlik. A dinamika kitalálásakor az elsődleges cél, hogy a játékosok úgy érezzék tudnak azonosulni a célokkal és egy kedvező fejlődési görbe mentén tudnak sikereket elérni. A fejlődés jelentheti a játékos karakter és a játékos személy tudásszerzését is.

A másik a környezet. A környezet két fő rész a világ, amiben a játék játszódik és a világot betöltő tárgyak és előlények. Az utóbbi években egyre nagyobb teret hódítottak a nyílt világú játékok. Ez azt jelenti, hogy a pályát a játékos szabadon fedezheti fel. Az ilyen játékok egyik legnagyobb kihívása, hogy ne legyen tele a világ ismétlődő mintákkal, mivel ezek unalmassá tehetik a játékmenetet.

## Játékpálya

Két gyakori megoldás van a játékterületek készítésére. Az egyik, hogy kézzel modellezik az egész térképet és az azon elhelyezett tárgyakat és élőlényeket. A másik megoldás, hogy az említett részeket részben vagy egészben véletlenszerűen generálják. sokszor a legjobb eredmény érdekében a két módszert ötvözik.

A kézzel tervezett pályának előnye, hogy tetszőleges részletességgel és fantáziával lehet megtervezni. Hátránya, hogy a nagyon nagy területek esetén ez rengeteg időt vesz igénybe és könnyen előfordulhatnak ismétlődő minták.

A generált pályáknál, mivel egy algoritmus készíti el a térképet nehezebb fantáziadúsan megvalósítani. Hátránya továbbá, hogy a nem megfelelő algoritmus használatakor szintén előfordulhatnak unalmas ismétlődések, vagy akár hibák is, amiket a véletlenszerű generálás miatt nehezebb megtalálni. A gyakorlatban a véletlenszerű pályákat inkább kisebb játékoknál használják. Például a roguelike [1] játékok szinte kivétel nélkül ezt a módszer használják. Ugyanakkor sokszor használnak előre elkészített építőelemeket, amik segítik a folyamatot.

A pálya elkészítése nem elegendő, szükséges annak berendezése is. A berendezés általában előre megtervezett tárgyak és karakterek segítségével történik. Vannak olyan játékok, ahol a változatosság fenntartásának kedvéért véletlenszerű generálással színesítik a tájat.

## Dungeon

Egy számos játékban elterjedt pályaelem az úgynevezett dungeon, a szónak talán a kazamata a legjobb magyar megfelelője. Ezt úgy kell elképzelni, hogy vannak fallal elválasztott szobáink, amelyek folyosók és ajtók kötnek össze. Általában az építmény a föld alatt helyezkedik el. További jellemzője még, hogy általában csapdák és szörnyek nehezítik a felfedező dolgát, ugyanakkor sokszor értékes jutalmak vannak elrejtve.

A képen vázlat, rajz, térkép látható

Automatikusan generált leírás

. ábra Dungeon alaprajz [2]

Dungeon már nagyon korai játékokban is megjelent. Az első ilyen játék az 1975-ös pedit5 [3] volt. Másik korai játék a Rogue [4] volt. Ez azért bírt nagy jelentőséggel mert ez képezi az alapját a már említett roguelike játéktípusoknak. Ez a típus napjainkban is nagy népszerűségnek örvend. Míg a pedit5 még fixen megadott dungeonnal dolgozott, a Rogue már procedurálisan generálta a pályát. Ekkor még a megjelenítés elég kezdetleges volt, karakterek segítségével rajzolták ki a kívánt képet.



. ábra ASCII Dungeon a Rogue játékban [5]

Az újabb játékoknál is előszeretettel használják meg a helyszínt. Sokszor csak a pálya egyik kisebb részeként, amit a játékos felfedezhet. Ezekben az esetekben az útvesztők általában 3 dimenziósak és kevésbé bonyolultak. Ilyen például a The Elder Scrolls V: Skyrim [6], ahol a nyílt világban barangolva botlunk bele kazamatákba.

Azokban az esetekben, ahol az egész játékterületet egy dungeon jelenti sokszor 2 dimenziós megjelenítést alkalmaznak. Általában ezeknél a játékoknál kevésebb hangsúlyt fektetnek az élethű megjelenítésre. Ilyen játék a felülnézetes The Binding of Isaac [7], amiben a területből mindig csak egy szobát lát a játékos, nem az egész rendszert.

A legtöbb játéknál a szobák szabályos alakúak. A másik gyakori eset, hogy kézzel elkészítenek a tervezők néhány szobát és utána azokat felhasználva készítik el a kazamatákat. Az utóbbi esetben általában szebb, de unalmasabb eredményt kapunk. Akkor szokták ezt a módszert alkalmazni, ha a játékterület csak kis részét képezik az útvesztők.

# Pályagenerálás

A pályagenerálás szempontjából két nagy csoportot tudunk elkülöníteni. Az egyik esetben a pálya generálása futásidőben történik, a másik esetben pedig a játék indítása elött.

A futásidejű vagy valós idejű generálás többféleképpen valósulhat meg. Az egyik eshetőség, hogy az egész pályát még mielőtt elkészülne a játék legeneráljuk. Ebben az esetben a játékosnak ki kell várnia ezt a folyamatot, amiatt a hossza nem nagyon lehet több mint pár 10 másodperc, de ideális esetben ennek célszerű pár másodperc alatt megtörténnie. A másik lehetőség, hogy mindig csak azt a területet generálja le az algoritmus, ahol a játékos jár, ezt használva a játékosnak kevesebbet kell várnia, ugyanakkor lehetnek egyéb hátránya. Hátránya lehet az utóbbi megoldásnak, hogy nehezebb egész pályára érvényes szabályokat megfogalmazni.

A játékidő előtti generálás általában olyan esetekben hasznos, ahol nem cél, hogy a játékos mindig más, véletlenszerű pályát kapjon. Akkor az algoritmus használatának az az előnye, hogy a tervezőnek nem kell kézzel lemodelleznie az egész játékterületet. Sokszor pályáknak csak egy része van generálva, például a növényzet, a többi kézzel van megtervezve. Sokszor használják azt a megoldást, hogy a térkép egy része játékidő elött el van készítve akár generálással akár manuálisan, viszont másik részét futásidőben generálják. Erre egy gyakori példa, hogy a domborzat és növényzet előre el van készítve, de az ellenségeket csak amikor már ott jár a játékos akkor generálják.

Az algoritmusok szempontjából fontos tényező, hogy egy algoritmus determinisztikus vagy sem. A nem determinisztikus algoritmusokat nevezzük véletlenszerű vagy random algoritmusoknak. Ezek azt jelentik, hogy nem tudjuk az algoritmus eredményét előre meghatározni a bemenetek ismeretében. Egy algoritmus elméletileg lehet nem determinisztikus, azonban a számítógépek általában nem alkalmasak ténylegesen véletlenszám generálásra. Ez azt jelenti, hogy csak a felhasználónak tűnik úgy, hogy véletlenszerű az eredmény, az valójában nem az. Ennek oka, hogy a számítógépek úgynevezett pseudo random számok generálása képesek, amik csak modellezik a valós véletlenszám generálást. Az említett megkötések ellenére a köznyelvben a nem determinisztikusnak tervezett számítógépes algoritmusokat véletlenszerűnek szoktuk nevezni.

Ha a játékokat megpróbáljuk a pályájuk alapján kategorizálni, akkor a két legnagyobb csoport a háromdimenziós és a kétdimenziós. Általában a játék nézete, dinamikája és műfaja szoros kapcsolatban állnak egymással. Elmondható, hogy a háromdimenziós játékokhoz nagyobb számítási kapacitásra van szükség. Emellett általában a kétdimenziós játékokat kevésbé részletesen kidolgozott megjelenítés jellemzi. Ezek következménye, hogy a kétdimenziós pályáknál gyakoribb a futásidejű véletlenszerű pályagenerálás.

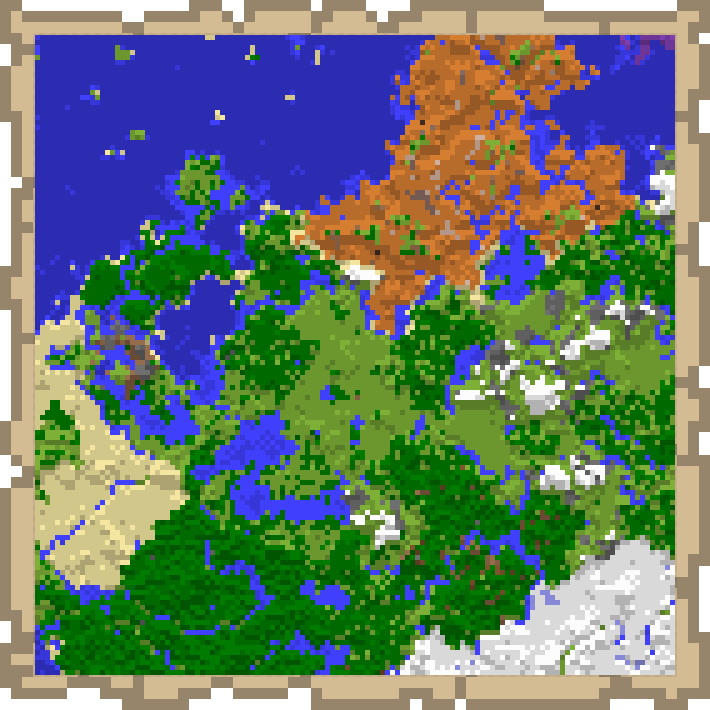
## Kétdimenziós pályák

Ha azokat a játékműfajokat nézzük amelyeknél nagyobb hangsúly van fektetve a véletlenszerű pályagenerálásra, akkor megfigyelhető, hogy ezek jellemzően kétdimenziósak. Három főbb nézetet különböztetünk meg. Ezek a felülnézet, az oldalnézeti és az izometrikus [8]. A tapasztalatok alapján a véletlenszerű pályagenerálás a felülnézeti játékoknál a legelterjedtebb.

Három gyakori rész van amit leggyakrabban generálással készítenek. Ezek a labirintusok, az ellenfelek és akadályok, illetve a jutalmak. Ezekre mind jellemző, hogy a véletlenszerűség izgalmasabbé teszi az újrajátszást. Összességében ezekben az esetekben a véletlenszerűség a játékdinamika része.

## Háromdimenziós pályák

Ezekben az esetekben a generáló algoritmusokat inkább a tervezők dolgának megkönnyítése miatt alkalmazzák. Természetesen ez alól is van kivétel, ilyen kivétel például a Minecraft [9], ami napjaink egyik legelterjedtebb játéka.



. ábra felülnézeti kép egy Minecraft világról [10]

A Minecraft jellemzője, hogy az egész térkép és az azon szereplő állatok, növények és épületek is véletlenszerűen generálódnak. Ezt úgy valósították meg, hogy a domborzatot egységnyi mérettű kocka alakú építőelemekből készítik el. Emellett a térkép régiókra van osztva és egyszerre mindig azt a régiót generálják le, amelyre belép a játékos, a már elkészülteket pedig elmentik.

# Procedurális generálás

A procedurális generálás [11] lényege, hogy véletlenszerű algoritmus segítségével szeretnénk olyan eredményt kapni, ami természetesnek tűnik. A módszert használják többek között az építészetben, gépészetben és a játékfejlesztés során is. Talán ez utóbbi a legelterjedtebb felhasználási terület.

A módszert általában úgy alkalmazzák, hogy előre elkészítenek egy vagy több alap alkotóelemet, majd ezeket használja fel az algoritmus. Az elemekkel általában egyszerű műveleteket hajtanak vége. A legyakoribb, hogy egymáshoz illesztik őket és transzformációkkal színesítik a képet.

## Procedurális algoritmusok a játékfejlesztésben

A játékoknál a módszert elsősorban domborzat, növények és textúrák generálására használják. De hangok készítéséhez is előszeretettel használják. A Rainworld [12] játékban például érdekes módon a karakter és az ellenfelek mozgásának animációjához használják a módszert, ami egy egyedülálló megjelenést kölcsönöz a játéknak.

A játékoknál legelőször pályagenerálásra használtak procedurális algoritmusokat. Az első ezt a módszert használó játékok az 1980-as évek környéken terjedtek el. Ezek dungeon generálását végeztek ilyen jellegű logikákkal. Ezekre az volt a jellemző, hogy megadott alakú csempékből építették fel a helyszíneket. Leggyakrabban négyzet alakú csempéket használtak, de hatszögek alkalmazására is van példa. Az első játékok mind felülnézetesek voltak.

Az óriási térképpel rendelkező játékoknál nincs arra lehetőség, hogy minden egyes apró részletet megtervezzenek a tervezők. Az ilyen esetekben gyakori, hogy a térkép domborzatát megtervezik kézzel, viszont a tereptárgyakakat és növényzetet már más módszerrel készítik. Elterjedt szokás, hogy elkészítenek néhány tárgyat és növényt, majd azokat használják fel sok helyen, esetleg átméretezve vagy áttextúrázva. Sajnos az ilyen ismétlések könnyen észrevehetőek és nagy területek esetén ronthatják a játékélményt. Egyre elterjedtebb módszer, hogy a növényzetet procedurális algoritmusokkal készítik el, ezzel változatos természetet lehet teremteni. Sokat segít, hogy az említett esetekben nincs szükség arra, hogy futás időben történjen a pályagenerálás. Ez azért jó, mert így az algoritmusnak nem kell gyorsnak lennie és így egyszerűbb jobb minőségű eredményt kapni.

Az egyik legjobb példa procedurálisan generált játékterületekre a No Man’s Sky [13]. Ennél a játéknál a térkép több galaxist foglal magában, ezeket már lehetetlen lenne kézzel lemodellezni. A probléma megoldásaként a galaxisok bolygóit procedurálisan generálják. Ennek köszönhetően minden bolygó úgy lesz egyedi, hogy közben természetes marad és ideális esetben a játékos nem vesz észre ismétlődő mintákat.



4. ábra Procedurálisan generált bolygó a No Man's Sky játékban [14]

Összességében elmondható, hogy egyre elterjedtebbek a procedurális generáló algoritmusok nem csak az egyszerű kétdimenziós játékok körében, hanem a kidogozottabb, bonyolultabb háromdimenziós társaik között is.

# Unreal Engine

Ebben a fejezetben szeretném bemutatni általánosságban a játékmotorokat és részletesebben az Unreal Engine [15] motort, és hogy miért erre esett a választásom. Ezen kívül összehasonlítok néhány közkedvelt játékmotort.

## Játékmotorok általánosságban

Játékmotornak egy olyan környezetet nevezünk, aminek elsődleges célja a játékfejlesztők segítése a fejlesztés során.

Sok különböző terület van, amiben támogatást tudnak nyújtani a motorok. Ezek nem mindegyikében segít mindegyik motor. A leggyakoribb része a környezeteknek a megjelenítő és fizikai motor. Ezek mellett gyakoriak a hang, ruházat, pályaszerkesztő és hálózati komponensek. Sokszor nem teljesen általános célú motorokat készítenek, elterjedt például a kettő- és háromdimenzió mentén történő szétválasztás.

A modern, nagy motorok esetében elterjedt az a törekvés, hogy olyan felhasználók is tudják használni a terméket, akik nem értenek a szoftverfejlesztéshez. Ennek elősegítéseként szoktak grafikus felhasználói felületet biztosítani a fejlesztőknek.

Sokszor jellemző a játékmotorokra, hogy nem egy könyvtárként kell elképzelni őket, hanem egy felületet nyújtanak, amin keresztül lehet őket használni. Az ezen a felületen történő felhasználást nevezik scripting-nek. Ezt úgy kell elképzelni, hogy a játékmotor rendelkezik egy egyedi parancskészlettel, amin keresztül ki lehet használni a tudását. Ez lehet egy egyedi egyszerű programozási nyelv, egy már létező nyelv testreszabott parancskészlettel vagy egy grafikus felület is. A megoldás segítségével nem kell ismerni a motor bonyolult belsős szerkezetét és működését ahhoz, hogy használni tudjuk. Ugyanakkor hátránya, hogy ha valami olyat szeretnénk, amit nem tudunk megoldani a motor által biztosított eszközökkel, akkor nincs feltétlenül lehetőségünk a belső működés kiegészítésére.

Összességében, ha játékfejlesztéssel szeretnék foglalkozni, akkor mindenképp érdemes egy kész játékmotort használni. Ez nem csak könnyebbé és gyorsabbá teszi a munkát, de a sok felhasználó miatt kevésbé kell aggódni a hibák miatt.

## Játékmotorok összehasonlítása

A motorválasztás és összehasonlítás során összeszedtem olyan jellemzőket, amik alapján könnyen ki tudom választani a céljaimhoz legjobban illeszkedő terméket.

Az alábbiakat vizsgáltam:

* Ingyenes
* Nyílt forráskódú
* Scripting és annak nyelve vagy felülete
* Platformok

### Unreal Engine

Az Unreal Engine [16] egy nyílt forráskódú nagyjából ingyenes játékmotor, amit 3D játékok fejlesztésére találtak ki. A motor legnagyobb előnye, hogy nagyon élethű háromdimenziós megjelenítésre képes. Ezen kívül a számos felhasználónak köszönhetően rengeteg erőforrás és útmutató érhető el hozzá. A motor szinte minden népszerűbb platformmal kompatibilis.

A motor C++ nyelven íródott és programozása is ezen a nyelven történik. Ezen kívül készült hozzá egy Blueprint nevű grafikus scripting eszköz, aminek köszönhetően a programozáshoz nem értők is tudják használni. Amennyiben a motor nem elégítené ki a fejlesztő igényeit meg van a lehetősége annak átírására és kiegészítésére.

### Unity

A Unity [16] egy sajátfelhasználásra ingyenes részben nyílt forráskódú játékmotor. Hasnálható 2D és 3D játékokhoz is. Szinte minden népszerűbb platformra lehet a segítségével fejleszteni. Nagy előnye, hogy óriási felhasználói bázisa van, ezért rengeteg mások által készített, szabadon felhasználható erőforrás létezik. Ezen kívül nagyon sok útmutató lelhető fel.

A programozása és scriptelése C# nyelven történik. Ezen kívül a felhasználóknak biztosit egy sokrétű és kényelmes grafikus felhasználói felületet. Kezdő játékfejlesztőknek kifejezetten ajánlott.

### Godot

A Godot [17] egy teljesen nyíltforráskódú és ingyenes motor. Fejlesztése több nyelven lehetséges, ilyen a C++ és C#, de van egy saját Python-hoz hasonló GDScript nevű nyelve is. Segítségével 2D és 3D játékokat is lehet készíteni. Rendelkezik saját grafikus felhasználói felülettel és fejlesztési környezettel is, de könyvtárként is felhasználható.

Hátránya, hogy a nagyobb motorokhoz képest le van maradva tudásban és hatékonyságban, főleg 3D játékok esetén. Továbbá a játékkonzolokra való fejlesztéset egyáltalán nem támogatja. Hátrányai ellenére folyamatosan fejlődik és nő a felhasználóinak száma.

### raylib

A raylib [18] a többi említett motorral ellentétben egy könyvtár, nem rendelkezik semmilyen grafikus felhasználói felülettel. C nyelven íródott és OpenGL [19] felhasználásával jelenít meg grafikus elemeket. Használata mélyebb programozói tudást igényel, ugyanakkor használata nem kifejezetten nehéz. Előnye, hogy a megfelelő programozási tudás segítségével jól testreszabható és sokféleképpen fel lehet használni. Hátránya, hogy egy manapság a játékfejlesztésben már keveset használt nyelven íródott.

## Unreal Engine

A választásom az Unreal Engine játékmotorra esett. Ennek elsődleges okai, hogy nyílt forráskódú és C++ nyelven íródott. Ezen kívül volt már egy kevés tapasztalatom a motorral.

A cél, amire használni szeretném a motort koránt sem átlagos. Futásidőben szeretnék vele megjeleníteni futás időben generált alakzatokat. Ilyenre egy átlagos játéknál nincsen szükség, úgyhogy a legtöbb játékmotor nem is biztosít erre megoldást. Szerencsére az Unreal motorban van egy Procedural Mesh nevezetű technológia, amivel meg tudom oldani a problémát, viszont nem túl kényelmesen.

Ezen a ponton, hogy egyik motor sem támogatja készen, amit szeretnék elgondolkodtam azon, hogy saját magam írom meg a motort. Végül azért nem ezt a megoldást választottam, mert igaz, hogy a megjelenítésnél nem kapom meg azt a segítséget, amit szeretnék más területen mégis sokat tud nyújtani a motor. Többek között nem nekem kell foglalkoznom az ütközésdetektálással, fizikával vagy textúrázással. Ezek fényében végül úgy döntöttem megpróbálkozok az Unreal Engine használatával.

### Procedural Mesh

A Procedural Mesh technológia futásidőben számolt alakzatok megjelenítésében segít. Hátránya, hogy nem túl jól dokumentált, valamint használata sem túl egyszerű és kényelmes.

Ahhoz, hogy használhassam ezt a megoldást nekem kell az alakzat pontjaiból elkészítenem a háromszöghálót. A háromszögháló ahhoz szükséges, hogy a grafikus kártya ki tudja rajzolni a kívánt alakzatot. Az ehhez használt algoritmusra majd a későbbiekben térek ki.

A képen ég, képernyőkép, művészet, tervezés látható

Automatikusan generált leírás

5. ábra Példák a Procedural Mesh használatára [21]

# Specifikáció

A generátor és a megjelenítő két egymástól független program. Annyi közös csupán bennük, hogy ugyanazon az adatszerkezeten is képesek dolgozni. Ennek megfelelően a specifikációjuk külön fogalmazandó meg.

## Generátor

Ez a rész nem csak a generátor algoritmust foglalja magában, hanem a hozzá tartozó adatszerkezeteket, matematikai és geometria fogalmakat, és egy interface elkészítését, amin keresztül az algoritmus használható.

Az algoritmust és a felhasználói felületet érdemes minél részletesebben specifikálni. A többi említett részt ezekhez igazodva kell választani vagy megtervezni és implementálni.

### Algoritmus

Az algoritmussal szemben fontos követelmény, hogy véletlenszerű legyen, ugyanakkor szép eredményt adjon. Szépnek olyan rendszereket lehet hívni, melyek jól felhasználhatóak egy játék során és a játékos nem érzi úgy, hogy valami fura. Általánosságban elmondható, hogy a szabályos alakzatokat szépnek nevezzük.

Az algoritmustól további elvárás, hogy az eredmény egy dungeon legyen. A kapható dungeon egymáshoz illeszkedő szobákból kell, hogy álljon, melyek között átjárónak kell lenni. Az átjáróknak lehetővé kell tenniük, hogy mindegyik szobába el lehessen jutni, vagyis nem lehet olyan szoba, aminek egyik falán sincs átjáró. A szobák és ajtók egy útvesztőt alkotnak.

További követelmény, hogy a szobák tetszőleges sokszög alakját felvehetik. Ez az a tulajdonság, amelyben ez a megoldás különbözik a számtalan másiktól. Ez megkötés nehezíti azt a már említett fontos követelményt, hogy a dungeon szép kell, hogy legyen.

Az implementációhoz kötött további követelmény, hogy megefelelően részekre legyen osztva és a részek egymástól függetlenek legyenek. Erre azért van szükség, hogy meglegyen a lehetőség arra, hogy több algoritmus közül tudjunk választani. Valamint azt is szeretnénk, hogy a későbbiekben bármikor új megoldásokkal lehessen kiegészíteni a már meglévőket. Ezek betartásához az a objektum orientált elvek és tervezési minták jó támpontot nyújtanak.

### Felhasználói felület

A felhasználói felület célja, hogy ezen keresztül lehessen paraméterezni és futtatni az algoritmust. A generátort kétféleképpen szeretnénk, hogy használni lehessen. Az egyik esetben önállóan futtatva azt, a másik esetben könyvtárként egy másik programból. A két felhasználás eltérő felhasználói felületet igényelhet.

Ha könyvtárként akarjuk használni akkor szükség van egy olyan felületre, amin keresztül el lehet végezni a paraméterezést és magát a generálást. Ebben az esetben a felhasználó a generátor által biztosított adatszerkezettel dolgozik, vagyis ennek kialakítása kulcsfontosságú.

Ha önállóan szeretnénk futtatni, akkor felhasználónak egyáltalán nincs szüksége arra, hogy ismerje az implementáció beli fogalmakat. Ekkor egy olyan felületre van szükség, ahol a felhasználó kényelmesen tud választani a rendelkezésre álló lehetőségek közül. Megfontolandó egy grafikus felhasználó felület készítése ebben az esetben.

## Megjelenítő

A megjelenítés Unreal Engine segítségével történik háromdimenzióban. Annak, hogy a generált eredményt szeretném grafikusan megtekinthetővé tenni két célja van. Az egyik, hogy így egyszerűbben tudom tesztelni az algoritmus működését. A másik, hogy ennek segítségével tudok egy kiindulási alapot nyújtani, amit már lehet használni játékoknál.

A legfontosabb követélmény, hogy a generált falakat, talaj elemeket és ajtókat meg tudja jeleníteni háromdimenzióban. Ezen kívül követelmény, hogy a kirajzolt alakzatokra teljesüljenek a fizika törvényei. Itt elsősorban az ütközésdetektálásra, a gravitációra és az árnyékolásra kell figyelni. További fontos szempont, hogy a motorban lehessen a megjelenés paramétereit állítani. Ilyen paraméter például, hogy milyen textúrák kerülnek az elemekre.

## Közös adatszerkezet

A generátor és megjelenítő között szükség van egy általános adatszerkezetre. Erre elsősorban azért van szükség, hogy bármelyik rész könnyen lecserélhető lehessen. Lehetőséget szeretnék biztosítani arra, hogy a rendszert bármilyen platformon és keretrendszerrel lehessen használni.

A választott megoldás, ami reprezentálja az adatszerkezetet egy egyedi fájl formátum. A formátumot olvasható szöveg alapúként szeretném megalkotni. Ennek az az elsődleges oka, hogy ennek segítségével könnyebben tesztelhető és kevesebb dokumentációval is értelmezhető.

Fontos követelmény, hogy úgy legyen felépítve az adatszerkezet, hogy azt utána tetszőleges nyelven és keretrendszerrel meg lehessen jeleníteni. Ezek alapján a formátumnak nem csak a minimális információt kell tartalmaznia, ami egyértelműen leírja a rendszert. Elmentésre kerülnek a háromdimenziós megjelenítéshez szükséges adatok is. Ez azt jelenti, hogy mivel a grafikus megjelenítők háromszöghálókkal dolgoznak, ezért az adatszerkezetben tárolva a van a generálás során kiszámolt háromszögháló is.

# Tervezési elvek és minták

A specifikációban nagy hangsúllyal szerepel a kiegészíthetőség és a helyettesíthetőség. Ahhoz, hogy ezek ne sérüljenek ezekre nagy hangsúlyt kell fektetni a tervezés során. Szerencsére ezek a követelmények nem egyediek a szoftverfejlesztés világában, ezért már rengeteg ezeket támogató elvet és mintát fogalmaztak meg. Ezek felhasználni mindenképp tanácsos.

A szoftverfejlesztés egyik elterjedt irányzata az objektumorientált programozás [22]. Röviden ennek az irányzatnak az az alapja, hogy a tervezés folyamán meghatározott fogalmakból objektumokat készítünk. Ezeknek az objektumoknak ideális esetben egyetlen jól meghatározott feladatuk van. A fejlesztés során az objektumok segítségével építjük fel a programunkat.

## Clean Code elvek

A Clean Code elvek azt hívatottak segíteni, hogy a fejlesztés során írt kód minél jobb minőségű legyen. Itt a minőség elsősorban külső jellemzőkre utal, nem a kód működési logikájára. Az elvek a jól átlátható és könnyen értelmezhető kódot segítik elérni.

Néhány fontosabb elv, a teljesség igénye nélkül:

* Meaningful Names: Az elnevezések utaljanak a felelősségekre és feladatokra. Törekedjünk tömör nevek használatára.
* Comments: A jó kód magyarázza magát ezért ne legyenek fölösleges kommentek. Indokolt esetben tömör, lényegre törő kommentek segíthetik a megértést.
* Functions: A függvények legyenek rövidek és lehetőleg egy dolgot csináljanak. Ne legyen sok paraméterük és használjuk megfelelően a visszatérési értékeket.
* Classes: Az osztályoknak legyenek rövidek és legyen egyértelmű felelősségük. Kerüljük a mély öröklési láncot és figyeljünk oda az osztályok közötti függőségekre.
* Refactoring: Fontos a gyakori refaktorálás, amivel szinten tartjuk vagy akár javítjuk a kód minőségét.
* Code Smells: Olyan kódrészletek, amelyek valamilyen hibára vagy elv sérülésére utalnak. Figyeljünk rájuk és szükség esetén hajtsunk végre rafaktorálásokat.

## Objektumorientált tervezési elvek

Az objektumorientál tervezési elvek abban segítenek, hogy a tervezés és a fejlesztés során minél jobban tudjuk az osztályokat és az azok közötti kapcsolatokat felépíteni. Az elvek támpontokat adnak a tervezői döntéshozatalnál és betartásuk segít a fenntartható kód írásában.

Kiemeltem néhány fontosabb elvet, amiket én is igyekeztem betartani:

* Single responsibility principle: Az osztályok felelőssége legyen egyértelmű és jól meghatároztt.
* Open-closed principle: Az objektumok legyenek nyíltak a kiterjesztésre, de zártak a módosításra.
* Liskov substitiution principle: Minden osztály viselkedését tekintve legyen helyettesíthető a leszármazottaival.
* Encapsualtion: A komponensek belső adataikat ne mutassák a külvilág felé. Azokat csak ellenőrzött függvényeken keresztül lehessen lekérdezni és módosítani.

## Objektumorientált tervezési minták

A tervezési minták [23] gyakori helyzetekre adnak megoldást. Ezeket a megoldásokat azért célszerű használni mert széleskörben elterjedtek. Az elterjedtségnek köszönhetően már kiforrott megoldásokról van szó, amiket sokan ismernek ezért könnyen megérthetőek.

A minták három fő kategóriába sorolhatóak. Az első kategória a létrehozási minták (Creational patterns), ezek objektumok létrehozásánál biztosítják a rugalmasságot, újrahasználhatóságot és kényelmet. A második kategória a szerkezeti minták (Structural patterns), ezek arra biztosítanak lehetőségeket, hogy objektumokból és osztályokból nagyobb rendszereket alkossunk rugalmasan és hatékonyan. A harmadik a viselkedési minták (Behavioral patterns) kategóriája, ezek algoritmusokkal és a felelősségek szétosztásával foglalkoznak.

Az alábbiakban néhány hasznosnak bizonyuló mintát fejtek ki. A részletezések az alapkoncepciókat vázolják fel nyelvfüggetlenül a részletek mellőzésével. Az említett példák általában leegyszerűsítik a valóságot, emiatt nem biztos, hogy a gyakorlatban minden esetben helyesek.

### Builder

A minta komplex objektumok kreálásánál hasznos. A lényege, hogy egy bonyolult létrehozási folyamatot több kisebb részre bont. A kisebb részek lehetővé teszik, hogy többféle objektumot hozzunk létre ugyanazon logika alapján.

Használatára egy példa, ha van egy osztály, aminek a létrehozó függvényének sok paramétere van és egy bonyolult logikát tartalmaz. Ebben az esetben a minta segítségével érdemes több részre bontani a folyamatot.

### Singleton

A minta két problémára is megoldást kínál. Az egyik, hogy biztosítani szeretnénk, hogy egy osztályból legfeljebb egy példány létezik. A másik, hogy ha egy globális változót szeretnénk leváltani, a negatívumok nélkül ellenőrzött körülmények között.

Röviden úgy működik, hogy az osztály tárol egy saját maga típusú változót és megtiltjuk, hogy az osztályt lehessen példányosítani. Amikor szeretnénk használni a singleton osztályt, akkor belül ellenőrzi, hogy létezik-e már és ha nem létrehozza. Amennyiben létezik, akkor azon hajtja végre a kívánt műveletet.

Egy lehetséges felhasználása, ha a programunk használ egy adatbázist egy adatbáziskezelő osztályon keresztül. Ebben az esetben nem biztos, hogy szeretnénk, ha több adatbáziskezelő példányon keresztül lehet elérni az adatbázist. Illetve könnyen lehet, hogy több egymástól független helyről is szeretnénk elérni.

### Adapter

A minta lényege, hogy több különböző objektumot szeretnénk együtt használni, de a felhasználási módjuk ezt nem teszi lehetővé. Ennek oka lehet például, hogy eltérő típusokat használnak ugyanannak az adatnak a reprezentálására. Megoldásként az objektumokhoz készíthetünk adaptereket, amelyek a kívánt kezelőfelületet biztosítják.

Felhasználására gyakori példa, ha több különböző ember által írt könyvtárat szeretnénk együtt használni. Ekkor valószínű, hogy nem úgy tervezték meg őket, hogy egymással jól tudjanak működni. Ebben az esetben az adapterek elkészítése megoldhatja a problémát.

### Bridge

A minta arra való, hogy egy nagy osztályt vagy több kisebb szorosan összefüggő osztályt szétszedjünk két szeparált részre. A megoldás lényeg, hogy készítünk egy absztrakciót és egy implementációt. Az absztrakció egy magasabb szintű fogalom, amin keresztül használhatóak az objektumok. Az implementációt az absztrakció tartalmazza és ez a konkrét megvalósítások interface-e.

Sokszor akkor használjuk amikor a fejlesztés közben eljutunk egy olyan állapothoz, ahol egy osztály már túl bonyolult, vagy sok osztály létezik a hibásan használt öröklés miatt.

### Decorator

A minta arra biztosít lehetőséget, hogy objektumokhoz új viselkedést csatoljunk dinamikusan, úgy, hogy becsomagoljuk őket egy speciális wrapper osztályba, ami tartalmazza az új viselkedést. A megoldás alapja, hogy decorator osztályok segítségével tartalmazás útján tudunk futásidőben viselkedést csatolni objektumokhoz.

Akkor szoktuk használni, ha ugyanazon logikához sok különböző viselkedésünk van és ezeket szeretnénk egymással kombinálni. Ekkor nem lehet minden lehetséges kombinációnak saját osztályt létrehozni, úgyhogy szeretnénk a viselkedéseket dinamikusan kezelni.

### Facade

A minta lényege, hogy egy egyszerű interface-t biztosít egy bonyolult könyvtárhoz vagy osztályhalmazhoz. A megoldás lényege, hogy a bonyolult logika egyszerű használatához írunk egy facade osztályt, ami elrejti a belső működést és csak a szükséges interface-t mutatja kifelé.

Akkor szoktuk használni, ha van egy bonyolult logikánk, aminek a belső működéséhez nem akarunk vagy nem is tudunk hozzáférni. Ekkor szeretnénk egyszerűsíteni és átláthatóbbá tenni a használatát.

### Mediator

A minta arra ad megoldást, ha több osztály között rendszertelen függőségek alakulnak ki. Megoldásként egy mediator osztály kikényszeríti, hogy rajta, ellenőrzött körülmények között kommunikáljanak egymással az osztályok. Minden osztály csak a mediatorral kommunikál és az továbbítja a kéréseket és küldi majd vissza az esetleges választ.

Akkor szokott rá szükség lenni, ha a fejlesztés során nem figyelünk a függőségekre. A bonyolulttá vált helyzeteket ilyenkor egy a mintán alapuló refaktorálás segítségével rendbe lehet tenni.

### Observer

A minta egy feliratkozásokon és értesítéseken alapuló logikát határoz meg. A lényege, hogy objektumok feliratkozhatnak másik objektumokra, amik adatt esetekben értesítést küldenek a feliratkozóknak. A megoldás lényege, hogy van egy osztály amire más osztályok szeretnének feliratkozni. A feliratkozott osztályokat tárolja az osztály és szükség esetén értesíti őket, amik már tudják, hogy mit kell ilyenkor tenniük.

A grafikus felhasználó felülettel rendelkező programok körében az egyik legelterjedtebb minta. Lehetővé teszi, hogy nem kell folyamatosan kérdezgetni a felhasználói felületet, hogy történt-e valami, hanem felhasználói interakció esetén az értesíti a szükséges osztályokat.

### Strategy

A minta segítségével algoritmusokat hozhatunk létre szeparált osztályokban, amiket aztán tetszőlegesen használhatunk azonos módon. Lényegében arra ad lehetőséget, hogy ugyanabban a környezetben különböző algoritmusokat futtassunk. A különböző algoritmusok ugyanazt az adatszerkezetet használják, de eltérhet a működésük vagy akár a céljuk is.

A megoldás úgy működik, hogy az osztály, ami különböző algoritmusokat akar használni eltárol egy algoritmus listát. Az algoritmusokat leíró osztályok rendelkeznek egy közös abstract őssel. A felhasználó osztály ugyanazon az interface-en keresztül tudja használni az algoritmusokat és ki tudja választani éppen melyiket akarja használni.

Akkor szoktuk használni amikor egy problémára különböző módon szeretnénk megoldást találni. Például útvonal tervezés esetén az egyik algoritmus a kilométerben a legrövidebb utat adja meg a másik pedig időben a legrövidebbet.

### Template Method

A minta segítségével hasonló logikájú algoritmusokat hozhatunk létre egy közös váz alapján, amiben a különböző lépéseket tetszőlegesen testreszabhatjuk. A megoldás lényege, hogy meghatározzuk az algoritmusok közös lépéseit, amikre minden esetben szükség van. A lépéseknek meg kell adni a bementének és kimeneténke a struktúráját, ezeknek minden megvalósításban azonosnak kell lenni. Ezek után létrehozhatjuk a különböző megvalósításokat, amiket aztán tetszőlegesen kombinálhatunk.

Bonyolultabb több lépésből álló algoritmusoknál hasznos. Például ha az algoritmus első lépése az adatok fájlból történő beolvasása, akkor ezt a lépést különböző fájl formátumok esetén máshogy kell implementálni, viszont ezutáni lépések akár azonosak is lehetnek.

# Tervezés

# Algoritmusok

# Dungeon generátor

# Dungeon megjelenítő

# Továbbfejlesztés

A tervezés és az implementálás során is az egyik legfontosabb szempont az volt, hogy minél könnyebben lehessen kiegészíteni. Ennek megfelelően mind a generátorban, mind a megjelenítőben biztosítva vannak olyan pontokat, ahol könnyen lehet beépíteni új megoldást.

## Generátor továbbfejlesztése

A generátor esetében a legkézenfekvőbb továbbfejlesztés az új algoritmusok bevezetése. Ezekkel lehetne a hatékonyságot növelni vagy akár jobban élvezhetőbbé tenni a generált eredményeket.

Ezen kívül ki lehetne egészíteni más fajta algoritmusokkal is, amik például ellenfeleket, tereptárgyakat vagy akadályokat helyeznének el a pályán. Be lehetne vezetni különböző nehézségi szinteket, amelyeket változtatnák a pálya bonyolultságát. Az is érdekes kiegészítés lenne, ha a labirintus háromdimenziós lenne, vagyis több egymás fölött réteg létezne.

Kicsit más irányból nézve a dolgot érdemes lehetne egy független grafikus felhasználói felületet készíteni a generátorhoz. Ennek segítségével kényelmesen fel lehetne paraméterezni mindent és megtekinteni az kapott eredményt.

A programkód szempontjából a jelenlegi gráf adatszerkezet nem elég kiforrott. Ennek az implementációját el lehetne készíteni vagy akár egy könyvtárral is lehetne helyettesíteni. Erre azért lenne szükség, hogy bonyolultabb gráf algoritmusokat is lehessen használni.

## Megjelenítő továbbfejlesztése

A megjelenítés Unreal Engine segítségével lett megoldva, egy kézenfekvő továbbfejlesztés lenne ezt megvalósítani más motorokkal vagy könyvtárakkal. A jelenlegi megoldás hátránya, hogy külön történik a generálás és a megjelenítés, erre egy szép és egyszerű megoldás az lenne, ha készülne egy Unreal plugin.

A megjelenítés jelenleg elég minimalista, előnyös lenne ezt szépíteni, hogy egy kellemes látványú környezetet alkothassunk. Ezt folytatva a pályát ki lehetne egészíteni a játékoknál használatos elemekkel. Ilyen például egy kis térkép, amin látjuk a pálya térképét és hogy hol helyezkedik el a játékos.

Ezeken túl bármikor fel lehet használni a pályát egy teljes játék készítéséhez. Ehhez szükséges lehet a jelenlegi megoldások módosítása, de akár a mostani megvalósítást is fel lehet használni kiindulási alapként.

# Összegzés

# Irodalomjegyzék

|  |  |
| --- | --- |
| [1] | Wikipedia, „Roguelike,” [Online]. Available: https://en.wikipedia.org/wiki/Roguelike. [Hozzáférés dátuma: február 2024]. |
| [2] | Mike, „Sly Flourish,” [Online]. Available: https://slyflourish.com/your\_only\_dungeon\_map.html. [Hozzáférés dátuma: február 2024]. |
| [3] | Wikipedia, „pedit5,” [Online]. Available: https://en.wikipedia.org/wiki/Pedit5. [Hozzáférés dátuma: februar 2024]. |
| [4] | Wikipedia, „Rogue,” [Online]. Available: https://en.wikipedia.org/wiki/Rogue\_(video\_game). [Hozzáférés dátuma: február 2024]. |
| [5] | Wikipedia, „Rogue,” [Online]. Available: https://en.wikipedia.org/wiki/Rogue\_(video\_game)#/media/File:Rogue\_Screenshot.png. [Hozzáférés dátuma: február 2024]. |
| [6] | Wikipedia, „The Elder Scrolls V: Skyrim,” [Online]. Available: https://en.wikipedia.org/wiki/The\_Elder\_Scrolls\_V:\_Skyrim. [Hozzáférés dátuma: február 2024]. |
| [7] | Fandom, „The Binding of Isaac Wiki,” [Online]. Available: https://bindingofisaac.fandom.com/wiki/The\_Binding\_of\_Isaac\_Wiki. [Hozzáférés dátuma: február 2024]. |
| [8] | Wikipedia, „Izometrikus axonometria,” [Online]. Available: https://hu.wikipedia.org/wiki/Izometrikus\_axonometria. [Hozzáférés dátuma: március 2024.]. |
| [9] | Wikipedia, „Minecraft,” [Online]. Available: https://hu.wikipedia.org/wiki/Minecraft. [Hozzáférés dátuma: március 2024]. |
| [10] | M. Fandom, „Map,” [Online]. Available: https://minecraft.fandom.com/wiki/Map. [Hozzáférés dátuma: március 2024]. |
| [11] | Wikipedia, „Procedural generation,” [Online]. Available: https://en.wikipedia.org/wiki/Procedural\_generation. [Hozzáférés dátuma: február 2024]. |
| [12] | Rainworld, „Rainworld,” [Online]. Available: https://rainworldgame.com/. [Hozzáférés dátuma: február 2024]. |
| [13] | Wikipedia, „No Man's Sky,” [Online]. Available: https://en.wikipedia.org/wiki/No\_Man%27s\_Sky. [Hozzáférés dátuma: február 2024]. |
| [14] | B. Insider, „The 10 most beautiful planets people have found so far in 'No Man's Sky',” [Online]. Available: https://www.businessinsider.com/no-mans-sky-beautiful-planet-photos-2016-8#regardless-of-where-youre-standing-the-many-worlds-of-no-mans-sky-all-look-pretty-amazing-9. [Hozzáférés dátuma: február 2024]. |
| [15] | pcgamesn, „A Skyrim modder is enhancing its dungeons and doubling their sizes,” [Online]. Available: https://www.pcgamesn.com/the-elder-scrolls-v-skyrim/mod-dungeons-overhaul. [Hozzáférés dátuma: február 2024]. |
| [16] | Steam, „steamcommunity,” [Online]. Available: https://steamcommunity.com/sharedfiles/filedetails/?id=887211109. [Hozzáférés dátuma: február 2024]. |
| [17] | Steam, „steamcommunity,” [Online]. Available: https://steamcommunity.com/sharedfiles/filedetails/?id=2910088884. [Hozzáférés dátuma: február 2024]. |

Függelék

A generátor és a megjelenítő megtalálható GitHubon az alábbi linkeken: