FELADATKIÍRÁS

Az elektronikusan beadott változatban ez az oldal törlendő. A nyomtatott változatban ennek az oldalnak a helyére a diplomaterv portálról letöltött, jóváhagyott feladatkiírást kell befűzni.



Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Rittgasszer Ákos

Dungeon Generátor Unreal megjelenítéssel

Rajacsics Tamás

BUDAPEST, 2024

Tartalomjegyzék

[Összefoglaló 6](#_Toc162291829)

[Abstract 7](#_Toc162291830)

[1 Bevezetés 8](#_Toc162291831)

[1.1 Játékpálya 8](#_Toc162291832)

[1.2 Dungeon 9](#_Toc162291833)

[1.3 Dolgozat szerkezete 11](#_Toc162291834)

[2 Pályagenerálás 12](#_Toc162291835)

[2.1 Kétdimenziós pályák 13](#_Toc162291836)

[2.2 Háromdimenziós pályák 13](#_Toc162291837)

[3 Procedurális generálás 15](#_Toc162291838)

[3.1 Procedurális algoritmusok a játékfejlesztésben 15](#_Toc162291839)

[4 Unreal Engine 17](#_Toc162291840)

[4.1 Játékmotorok általánosságban 17](#_Toc162291841)

[4.2 Játékmotorok összehasonlítása 18](#_Toc162291842)

[4.2.1 Unreal Engine 18](#_Toc162291843)

[4.2.2 Unity 18](#_Toc162291844)

[4.2.3 Godot 19](#_Toc162291845)

[4.2.4 raylib 19](#_Toc162291846)

[4.3 Unreal Engine 19](#_Toc162291847)

[4.3.1 Procedural Mesh 20](#_Toc162291848)

[5 Specifikáció 21](#_Toc162291849)

[5.1 Generátor 21](#_Toc162291850)

[5.1.1 Algoritmus 21](#_Toc162291851)

[5.1.2 Felhasználói felület 22](#_Toc162291852)

[5.2 Megjelenítő 22](#_Toc162291853)

[5.3 Közös adatszerkezet 22](#_Toc162291854)

[6 Tervezési elvek és minták 24](#_Toc162291855)

[6.1 Clean Code elvek 24](#_Toc162291856)

[6.2 Objektumorientált tervezési elvek 25](#_Toc162291857)

[6.3 Objektumorientált tervezési minták 25](#_Toc162291858)

[7 Tervezés 29](#_Toc162291859)

[7.1 Generátor 29](#_Toc162291860)

[7.2 Fájl formátum 31](#_Toc162291861)

[7.3 Megjelenítő 32](#_Toc162291862)

[8 Algoritmusok 34](#_Toc162291863)

[8.1 Kruskal algoritmus 34](#_Toc162291864)

[8.2 Delaunay háromszögelés 35](#_Toc162291865)

[8.3 Generáló algoritmus 36](#_Toc162291866)

[9 Dungeon generátor 38](#_Toc162291867)

[9.1 Base modul 38](#_Toc162291868)

[9.2 Math modul 38](#_Toc162291869)

[9.3 Geometry modul 38](#_Toc162291870)

[9.4 Data modul 38](#_Toc162291871)

[9.5 Generator modul 38](#_Toc162291872)

[9.6 Dungeon Generator 38](#_Toc162291873)

[10 Dungeon megjelenítő 39](#_Toc162291874)

[11 Továbbfejlesztés 40](#_Toc162291875)

[11.1 Generátor továbbfejlesztése 40](#_Toc162291876)

[11.2 Megjelenítő továbbfejlesztése 40](#_Toc162291877)

[12 Összegzés 42](#_Toc162291878)

[13 Irodalomjegyzék 43](#_Toc162291879)

[14 Függelékek 46](#_Toc162291880)

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. 23.

...…………………………………………….

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 játékok készítése során egyre csökken a szoftverfejlesztők szerepe és egyre nagyobb hangsúly helyeződik a tervezők munkájára.

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 tanulását is.

A másik a környezet. A környezet két fő része 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 unalmasak lehetnek.

## 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 változatosan 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 sokszor csapdák és szörnyek nehezítik a felfedező dolgát, ugyanakkor a túlélők értékes jutalmakhoz juthatnak.

A képen vázlat, rajz, térkép látható

Automatikusan generált leírás

1. ábra Dungeon alaprajz [2]

Dungeonokat már a nagyon korai játékokban is előszeretettel használtak. Az első ilyen játék az 1975-ös pedit5 [3] volt. Egy másik korai játék pedig a Rogue [4] volt. Ez azért bírt nagy jelentőséggel mert ez képezi az alapját a már említett roguelike játéktípusoknak, ami 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.



2. 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 kétdimenziós megjelenítést alkalmaznak. Általában ezeknél a játékoknál kevésebb hangsúlyt fektetnek az élethű megjelenítésre. Ilyen játék a felülnézetes The Binding of Isaac [7], amely napjaink egyik legelterjedtebb roguelike játéka.

A legtöbb esetben 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.

## Dolgozat szerkezete

A dolgozat első részeiben kitérek a későbbiekben használt fogalmakra és összefoglalom azokat az ismeretek, amelyekre nekem is szert kellett tenne a fejlesztés és tervezés megkezdése elött. Összeszedem azokat játékokat és ötleteket is, amelyek legnagyobb hatással voltak rám a specifikáció meghatározásakor. Kitérek a játékmotorokra és megmagyarázom miért az Unreal Engine-re esett választásom. A részt a feladat specifikációja zárja le.

A második rész a tervezésről szól. Ismertetem azokat az elveket, amelyek alapján terveztem a programot és felvázolom a magasszintű struktúráját. Összefoglalom az architektúrát és hogy milyen gondolatok mentén csoportosítottam a programokat.

Az utolsó rész magáról az implementálás folyamatáról szól. Itt részletezem a dungeon generátor és a megjelenítő részleteit. Kitérek az érdekesebb problémákra és azok megoldására. Összefoglalom az implementáció fontosabb mozzanatait és a tervezéshez képest alacsonyabb szinten vázolom a program felépítését.

# Pályagenerálás

A pályagenerálás futási ideje szempontjából két nagy csoportot tudunk megkülönböztetni. Az egyik esetben a generálás futásidőben történik, a másikban pedig a játék indítása elött.

A futásidejű vagy valós idejű generálás többféleképpen valósulhat meg. Az egyik eshetőség, hogy az egész pályát a játék indításakor legeneráljuk. Ebben az esetben a játékosnak ki kell várnia ezt a folyamatot, ezért a hossza nem nagyon lehet több mint pár 10 másodperc, de 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.

A játékidő előtti generálás általában olyan esetekben hasznos, ahol nem cél, hogy a játékos mindig más, véletlenszerű pályát kapjon. Ekkor az algoritmus használatának az az előnye, hogy a tervezőnek nem kell kézzel lemodelleznie az egész játékterületet. Van, hogy 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 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 tulajdonság, hogy egy algoritmus determinisztikus vagy sem. A nem determinisztikus algoritmusokat nevezzük véletlenszerű vagy random algoritmusoknak is. 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 véletlen 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 szimulálják 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.

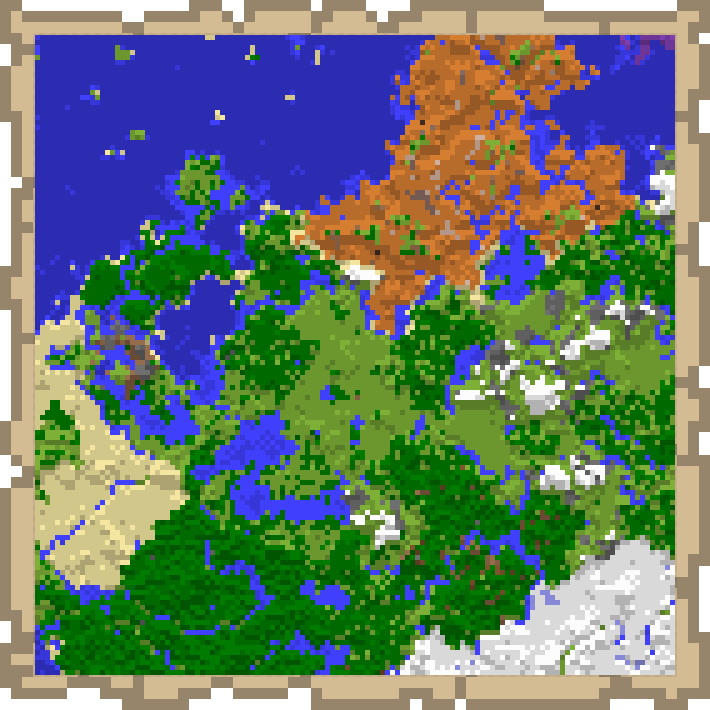
## Kétdimenziós pályák

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 nézet [8]. A tapasztalatok alapján a véletlenszerű pályagenerálás a felülnézeti játékoknál a legelterjedtebb.

Gyakori, hogy a kétdimenziós játékok azért véletlenszerű algoritmusokat használnak, hogy többször újrajátszható legyen a játék. Ezt úgy kell elképzelni, hogy ugyanazokkal a szabályokkal többször neki lehet futni a játéknak de a pálya és annak feltöltése mindig változik.

## Háromdimenziós pályák

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



3. 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 modellezni. 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. Olyan általános eszközöket biztosít, amelyekre több játéknak is szüksége lehet.

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 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, amelyeken keresztül akár programozni is lehet.

Sokszor jellemző a játékmotorokra, hogy nem egy könyvtárként kell elképzelni őket, hanem egy saját interface-t nyújtanak, amin keresztül lehet őket használni. Az ezen az interface-en 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
* Fejlesztés nyelve
* 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 [17] egy saját felhasználásra ingyenes, részben nyílt forráskódú játékmotor. Használható 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 [18] 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 fejlesztést egyáltalán nem támogatja. Hátrányai ellenére folyamatosan fejlődik és nő a felhasználóinak száma.

### raylib

A raylib [19] a többi említett motorral ellentétben egy könyvtár, nem rendelkezik semmilyen grafikus felhasználói felülettel. C nyelven íródott és OpenGL [20] felhasználásával jelenít meg grafikus elemeket. Használata mélyebb programozói tudást igényel, ugyanakkor nem kifejezetten bonyolult. 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 vele.

A cél, amire használni szeretném a motort korántsem 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 specifikációjukat külön fogalmazom 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.

### 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 nem megfelelő. Általánosságban elmondható, hogy a szabályos alakzatokat szépnek nevezzük.

Az algoritmustól további elvárás, hogy az eredmény egy dungeon legyen. Az elkészített dungeon egymáshoz illeszkedő szobákból kell, hogy álljon, melyek között átjárók lehetnek. Az átjáróknak lehetővé kell tenniük, hogy mindegyik szobába el lehessen jutni, vagyis nem lehet olyan szoba, amelynek egyik falán sincs ajtó. A szobák és ajtók egy útvesztőt alkotnak. Az útvesztővel kapcsolatban nincsen további megkötés megfogalmazva.

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

Az implementációhoz kötött további követelmény, hogy megfelelően 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 objektum orientált elvek és tervezési minták jó támpontot nyújtanak. Ezekre a tervezési részben fogok részletesebben kitérni

### Felhasználói felület

A felhasználói felület célja, hogy ezen keresztül lehessen paraméterezni és futtatni az algoritmust. A generátort kétféleképpen szeretném, 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 interface-re, amin keresztül el lehet végezni a paraméterezést és magát a generálást. Ebben az esetben a felhasználó a generátor által biztosított adatszerkezettel dolgozik, vagyis kialakításánál a generáló programra és a felhasználó program követelményeire is figyelni kell.

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

## 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 fontos, hogy a kirajzolt alakzatokra teljesüljenek a fizika törvényei. Itt elsősorban az ütközésdetektálásra, a gravitációra és az árnyékolásra gondolok. 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 független 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 a file, 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 ezeket elérjem szem elött kell tartani őket 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 osztályok 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 döntéshozatalban és betartásuk segít a fenntartható kód írásában.

Kiemeltem néhány fontosabb elvet, amelyeket én is igyekeztem betartani:

* Single responsibility principle: Az osztályok felelőssége legyen egyértelmű és jól meghatározott.
* 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 problémákra 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) csoportja, 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. A megfogalmazott példák általában leegyszerűsítik a valóságot, emiatt nem biztos, hogy a gyakorlatban minden esetben helyesek.

A *Builder* minta komplex objektumok kreálásánál hasznos. 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.

A *Singleton* minta két problémára is megoldást kínál. Az egyik, hogy biztosítani szeretnénk, hogy egy osztályból legfeljebb egy példány létezzen. 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.

Az *Adapter* 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 fél 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.

A *Bridge* 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.

A *Decorator* minta arra biztosít lehetőséget, hogy objektumokhoz új viselkedést csatoljunk dinamikusan. Ez úgy érhető el, hogy becsomagoljuk őket egy speciális 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.

A *Facde* minta lényege, hogy egy egyszerű interface-t biztosít egy bonyolult könyvtárhoz vagy osztályhalmazhoz. A megoldás alapja, 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.

A *Mediator* minta arra ad megoldást, ha több osztály között rendszertelen függőségek alakulnak ki. Megoldásként egy mediator osztály kikényszeríti, hogy rajta, ellenőrzött körülmények között kommunikáljanak egymással a felek. 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.

Az *Observer* 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.

A *Strategy* 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 absztrakt 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.

A *Template Method* 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

A tervezés a specifikációhoz hasonlóan két független részből áll. Ezek a generátor és a megjelenítő. Igyekeztem, ahol csak lehet külön kezelni őket, mivel azt szeretném, hogy egymástól függetlenül is tudjanak hasznosan működni. Az említett két részt még kiegészíti egy harmadik, a közöttük kapcsolatot teremtő fájl formátumának és tartalmának kitalálása.

A tervezés során igyekeztem az előző szakaszban összefoglaltakra figyelni. Alapvetően a tervezés nagyja megtörtént az implementálás megkezdése elött. Ugyanakkor a fejlesztés közben kiderülő tényezők, amikre nem gondoltam előre, utólag befolyásolták a tervezés eredményét és olykor újratervezésre is szükség volt.

## Generátor

Első lépésként eldöntöttem, hogy a fejlesztés nyelve a C++ lesz. Azért döntöttem emellett mert a játékfejlesztők körében ez az egyik legelterjedtebb és nekem is van nagyobb tapasztalatom benne.

A következő kérdés az volt, hogy hogyan szeretném tagolni a programot. Itt úgy döntöttem, hogy meghatározok adott felelősségű modulokat, amik aztán könyvtárként tudják egymást használni. Első körben az alábbi modulokat fogalmaztam meg:

* Base modul: Ebben a részben általános, mindenhol használandó dolgok kapnak helyet. Ilyenek például a logolás, a random szám generálás és a különböző guardok.
* Data modul: Ebben a modulban vannak az adatszerkezeti osztályok.
* Generator modul: Itt a generáláshoz szükséges algoritmusok és osztályok kapnak helyet.
* Math modul: Alapvető matematikai fogalmaknak ad helyet. Ilyen például a vektor, a gráf vagy a mátrix.
* Geometry modul: A szükséges geometriai fogalmak helye. Itt található például a sokszög osztály és az azon végezhető transzformációk.
* Dungeon generátor alkalmazás: Ez a rész tudja futtatni az algoritmust, itt fut össze a többi modul.

A felsorolt modulok részletesebben ki lesznek fejtve az implementációs fejezetben. A modulok közötti függőségek a következő ábrán (6. ábra) láthatóak, a Base modul kivételével, mivel attól mindegyik másik függ.

A képen szöveg, képernyőkép, Betűtípus, tervezés látható

Automatikusan generált leírás

6. ábra A modulok függőségi diagramja

A modulok meghatározása után azon gondolkoztam, hogy milyen könyvtárakat lenne érdemes használni, ahhoz, hogy egyszerűsítsem a dolgomat.

Elsőként a logolás elősegítéséhez szerettem volna egy könyvtárat használni. Ehhez az spdlog [24] könyvtárat választottam ki, amit már régebben is használtam és pozitív tapasztalatokat szereztem vele kapcsolatban. A segítségével testreszabható formátumban színes szöveget tudunk kiírni a standard kimenetre. Sajnos az MSVC v14.38-17.8 verziójú c++ fordító környékén az spdlog könyvtárat nem tudta már lefordítani a fordító, így a végleges állapotában ezt ki kellett kötnöm.

A másik terület, ahol könyvtárakat szerettem volna használni az az alapvető matematikai fogalmak. Itt hosszasabb kutatás után úgy döntöttem, hogy csupán a gráfok kezelésére használok könyvtárat, a többit magam implementálom. A gráfokhoz a LEMON [25] könyvtárat néztem ki. A könyvtárat beépítettem a programomban és amikor elkezdtem használni akkor jöttem rá, hogy nem képes arra amire szeretném használni, találtam benne több hibát is és még elavult is. Ezután nem fogtam neki másik könyvtár keresésének, hanem elkészítettem saját implementációmat.

Az előbb említett problémákból is látszik, hogy mennyire fontos úgy megtervezni a program struktúráját, hogy annak részei függetlenek legyenek és könnyen kicserélhessem őket. Ha nem így terveztem volna akkor például az említett gráf könyvtár kikötése és helyettesítése óriási fejtörést okozott volna.

## Fájl formátum

A fájl formátumának megtervezésekor arra kellett figyelnem, hogy elég álltalános legyen ahhoz a formátum, hogy bármely megjelenítőt fel lehessen készíteni a használatára.

Első lépésként azt kellett eldöntenem, hogy bináris vagy szöveg alapon tárolom az adatokat. A bináris eset előnyei, hogy hatékonyabb, vagyis ugyanazon az adatot lehetséges, hogy kevesebb memória használatával tudja tárolni, ugyanakkor önmagában az ember által nem olvasható. A szövegalapú megoldás hátránya a lehetséges nagyobb méret, ugyanakkor előnye az olvashatóság. Ezek figyelembevételével egy szöveg alapú formátumot találtam ki. A választott fájlformátumnak a .dg kiterjesztést adtam. Utólag úgy látom jobb lett volna egy széles körben elterjedt fomátumot használni és nem sajátot kitalálni. Egy alkalmas formátum lehetett volna az XML (Extensible Markup Language) [26].

Második lépésként összeszedtem azokat az adatokat, amiket szerettem volna elmenteni. Elsőként fájl formátum verzióját tárolom. Erre azért van szükség, mert ha a fejlesztés során változtatom a formátumot akkor a verizózásnak köszönhetően a régi formátumban mentett fájlok is megnyithatók. Ezen kívül meg van a konverzió lehetőség fenn áll. A másik általános adat, amiről úgy gondoltam, hogy a későbbiekben hasznos lehet, az a random generálás seedje.

Ezután következett a dungeon-t leíró adatszerkezet. Itt két különböző féle adatot határoztam meg, a talaj és fal elemeket. Mindkettő esetén két féle információt tárolok. Ezek az alakzat csúcspontjai és a háromszögeléssel kapott háromszögek indexei, amelyek a megjelenítéshez kellenek.

Az egyes elemeket leíró részek így néznek ki:

A képen szöveg, képernyőkép, Betűtípus látható

Automatikusan generált leírásA képen szöveg, képernyőkép, Betűtípus, szám látható

Automatikusan generált leírás

7. ábra A fal és talaj elemek a .dg kiterjesztésű fájlokban

Az említett példákban jól látható, hogy a *Wall* és *Floor* kulcsszavak jelölik az elemek típusát. Ezután látható két szám egy sorban. Ezek közül az első a csúcspontok számát, a második pedig a háromszögek számát határozzák meg. Ennek megfelelően a következő sorokban találhatóak a csúcspontok koordinátai, amik három számmal írhatóak le. Az azutáni sorok a háromszögek három csúcsának indexét adják meg, az alapján indexelve, hogy az előbbiekben hányadikként szerepeltek az egyes csúcsok.

## Megjelenítő

A megjelenítő Unreal Engine-ben készült, így ennek az alapvető szerkezetét a motor határozta meg. Itt három nagyjából független részt határoztam meg, amik azért feleltek, hogy a dungeont fájlból meg tudjam jeleníteni.

Az első az adatszerkezet, ami az egész rendszert reprezentálja. A második a beolvasó rész, ami fájlból létrehozza az adatszerkezetet. A harmadik pedig az egyes fal és talajelemek megjelenítését végző rész.

Az adatszerkezeti egyrészt felelős a beolvasás során létrehozott adatok tárolásáért, másrészt ez biztosítja azt a felületet, amelyen keresztül fel tudjuk használni a dungeont. Ez a két felelősség azért mosódik össze, mivel a fájlformátum minden adatot tartalmaz, így az említett felelősségek elférnek egymás mellett. A későbbiekben megfontolandó eszek szétválasztása.

A beolvasó rész felelőssége roppant egyszerű. Ennek a résznek a feladta, hogy ismerje a fájlformátumot és beolvasás közben azt a megjelenítő által várt formátummá konvertálja.

A harmadik, az egyes elemeket megjelenítő rész a legbonyolultabb. Ezek a beolvasott adatok segítségével létrehozzák a háromdimenziós objektumokat, amiket aztán meg tud a motor jeleníteni. Itt jön képbe a megjelenítés legnagyobb kihívása, ami az, hogy nem arra lettek kitalálva a játékmotorok, hogy futásidőben számolt alakzatoka jelenítsenek meg. Szerencsére erre nyújt megoldást a már korábban említett Procedural Mesh technológia.

A technológia kényelmesebb használatához létrehoztam egy olyan interface-t, ami az általam kívánt felhasználási módot kényelmessé teszi. Ez azért is fontos, mert ez részben elfedi a belső működést, így, ha azon változtatni kellene valamit, akkor ideális esetben ez a programkód többi részének változtatása nélkül megtehető.

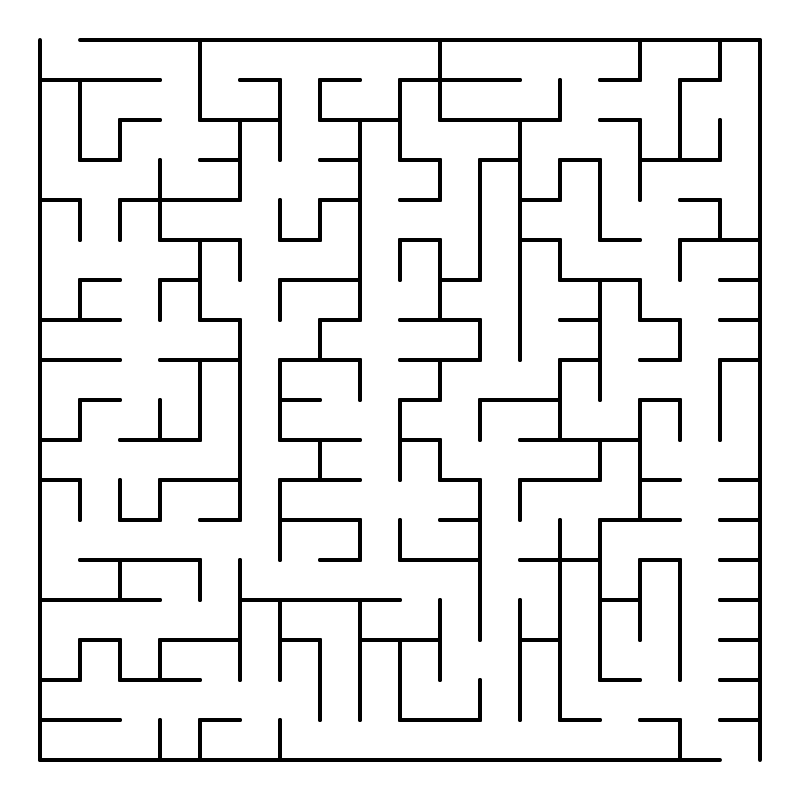
# Algoritmusok

A feladat nagy része a generáló algoritmusok kitalálása, ugyanakkor ennek kitalálásához és megvalósításához már létező algoritmusokat használtam fel. Ebben a fejezetben szeretném ismertetni a legfontosabb felhasznált algoritmusakt és végső generáló algoritmust is, az implementációs részletek nélkül.

## Kruskal algoritmus

Egy elterjed mohó gráfelméleti algoritmus a Kruskal algoritmus [27], ami egy súlyozott gráfban adja meg a minimális feszítőfát. Feszítőfának a gráf azon részgráfját nevezzük, amely a fa minden csúcsát tartalmazza. Minimális feszítőfának egy olyan fát nevezünk, amely élsúlyainak összege minimális. Az algoritmus remekül felhasználható tökéletes labirintus generálásához.

Tökéletesnek olyan labirintusokat nevezünk, amelyeknek bármely két pontja (szobája) között pontosan egy út vezet. Ezt úgy kaphatjuk meg a Kruskal algoritmus segítségével, ha a kiindulási gráf minden csúcsa az útvesztő egy mezője és a szomszédos mezők között van él a gráfban. Ezek után, ha erre a gráfra alkalmazzuk az algoritmust akkor a kapott feszítőfában nem szereplő élek helyére falakat rakunk.



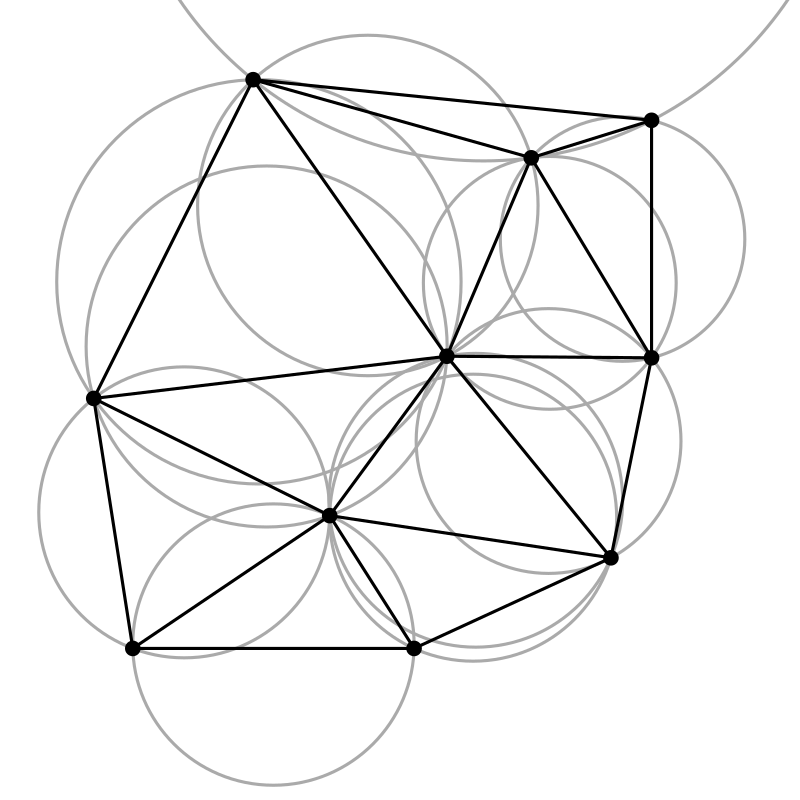
8. ábra Tökéletes labirintus [28]

Az algoritmus logikája, hogy a növekvő súly sorrendbe vesszük az éleket egyesével. Mindegyik élnél megnézzük, hogy az eddig kiválasztott élekhez adva lesz-e kör a gráfban. Ha igen akkor az élet eldobjuk, ha nem akkor hozzáadjuk a feszítőfához.

## Delaunay háromszögelés

Háromszögelő algoritmusoknak olyan algoritmusokat nevezünk, amelyek egy ponthalmazra illesztenek háromszögeket. A Delaunay háromszögelés [29] erre egy elterjedt megoldás, ami relatív szép eredményt ad. A háromszögelő algoritmusok egy elterjedt felhasználási módja az alakzatokra illeszkedő háromszögháló meghatározása, a számítógépes grafikában.

Az algoritmus lényege, hogy úgy határozza meg a háromszögeket, hogy az egy háromszöghöz tartozó pontok köré írt körében nem lehet benne semelyik másik pont.



9. ábra Háromszögelés a Delaunay algoritmussal

Az algoritmust használtam a generáló algoritmushoz és a generált alakzatok háromszögeléséhez is.

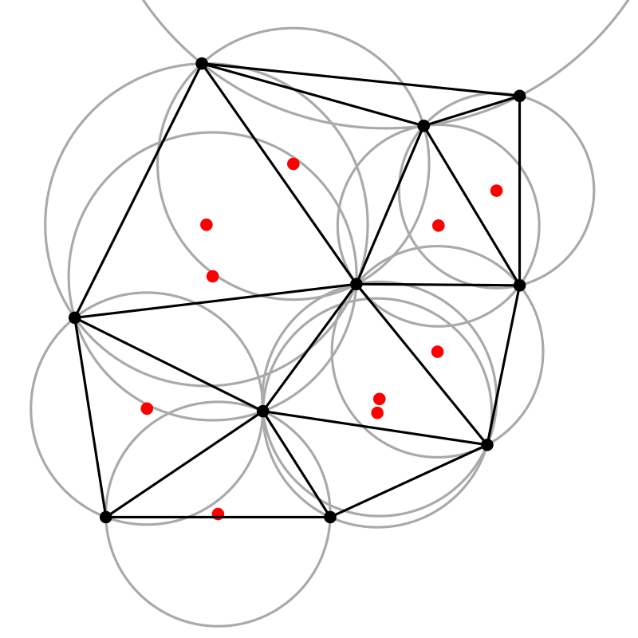
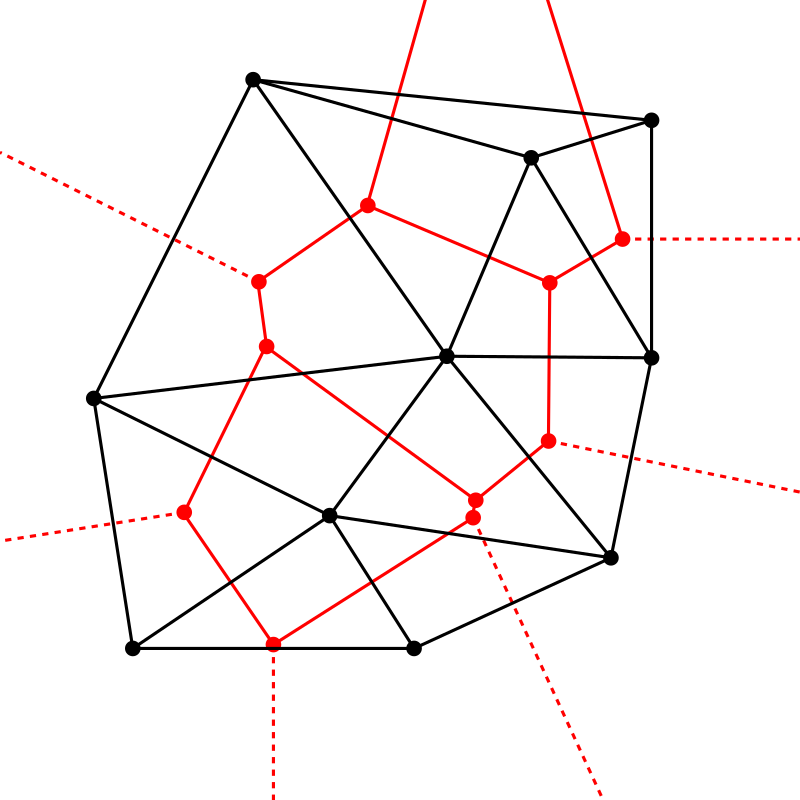
Az általam implementált megoldás lényege, hogy veszünk egy nagy háromszöget, amely magában foglalja az összes pontot. Ezután egyesével vesszük a pontokat. Mindegyik pontra igaz lesz, hogy létezik egy háromszög, amely azt tartalmazza, ekkor a háromszög csúcsait összekötjük az új ponttal. Az így kapott új háromszögelést megvizsgáljuk, ha valamelyik háromszögeknél sérül a szabály akkor azokat töröljük. Így megyünk tovább ameddig el nem készül a háromszögelés.

## Generáló algoritmus

A generáló algoritmus két részből áll. Az első rész a szobákat hozza létre, a második pedig a szobák között nyit ajtókat, úgy, hogy azok labirintust alkossanak.

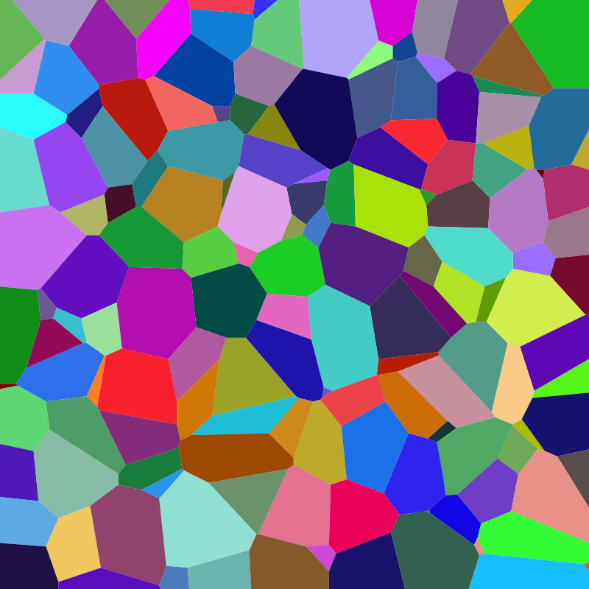
Az utóbbi, vagyis a labirintus meghatározása egy egyszerű Kruskal algoritmust használ. Itt csupán annyi a csavar, hogy szeretnénk az algoritmust véletlenszerűvé tenni, de maga a használt algoritmus determinisztikus. Ennek kikerüléseként az éleket nem súly szerinti sorrendbe vesszük, hanem véletlenszerű sorban. Másképp ezt úgy is fel lehet fogni, hogy véletlenszerű súlyokat rendelünk az élekhez.

A szobák generálásának alapjául a Delaunay háromszögelést használtam. A megoldás a Voronoi [30] diagrammal való kapcsolatán alapul. A kapcsolatot úgy kell elképzelni, hogy egy tetszőleges Delaunay háromszögeléshez egyértelműen tartozik egy Voronoi diagram és fordítva. A kapcsolat úgy működik, hogy ha vesszük a háromszögelés háromszögeinek középpontjait és ezek közül a szomszédosakat összekötjük egymással akkor egy Voronoi diagramot kapunk.

10. ábra Delaunay háromszögelés és Voronoi diagram kapcsolata

Voronoi diagramnak egy olyan formázatot nevezünk, ahol megadott kontrolpontokhoz cellákat rendelünk. Ekkor a sík bármely pontjára teljesül, hogy a hozzá legközelebbi kontrolponthoz tartozik. Az így kapott formázatot a kontrolpontok segítségével lehet paraméterezni. Ez lehetőséget ad egy szép, ugyanakkor véletlenszerű rendszer alkotására.



11. ábra Példa a Voronoi diagramra

A diagram jellemzői és a kontrolpontok meghatározása lehetővé teszi, hogy felhasználjam a szobák generálásánál. A diagram meghatározásának legegyszerűbb módja, ha egy Delaunay háromszögelésből adom meg. Ekkor a paramétereke a háromszögelés bemeneti koordinátái. Az eredményként kapott diagramm cellái fogják megadni a generált szobákat.

# Dungeon generátor

A generátor rész fejlesztését a tervezés során meghatározott modulok mentén igyekezetem különálló fejlesztési részekre bontani. A fejezetben szeretnék kitérni a modulokra, megemlítve az érdekesebb részeket és a felmerülő problémákat és megoldásukat.

## Base modul

## Math modul

## Geometry modul

## Data modul

## Generator modul

## Dungeon Generator

# 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

Nagyon örülök annak, hogy a választott feladatnak meg tudtam találni a szépségeit és a számtalan nehézség ellenére is szívesen dolgoztam rajta. Sikerült olyan programokat alkotni, amelyek már akár a való életben is hasznosak lehetnek egy játék tervezésekor.

Nagy tapasztalatot szereztem a tervezésben és újratervezésben, valamint az Unreal Engine használatával. Több olyan helyzettel is szembe találkoztam, amelyek akár a valós munkám során is előfordulhatnak és ekkor remélem majd fel fogom tudni használni az itt szerzett tapasztalataimat.

Összességében elégedett vagyok a kétéves munkámmal és szívesen gondolok vissza a fejlesztési időszakra. Arra is érzem a kellő motivációt, hogy szabadidőmben tovább dolgozzak projekten és abból akár egy teljes játékot készítsek.

# 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] | „Unreal Engine,” [Online]. Available: https://www.unrealengine.com/en-US. [Hozzáférés dátuma: március 2024]. |
| [16] | „Unreal Engine,” [Online]. Available: https://www.unrealengine.com/en-US. [Hozzáférés dátuma: március 2024]. |
| [17] | „Unity,” [Online]. Available: https://unity.com/. [Hozzáférés dátuma: március 2024]. |
| [18] | „Godot Engine,” [Online]. Available: https://godotengine.org/. [Hozzáférés dátuma: március 2024]. |
| [19] | github, „raylib,” [Online]. Available: https://github.com/raysan5/raylib. [Hozzáférés dátuma: március 2024]. |
| [20] | „opengl,” [Online]. Available: https://www.opengl.org/. [Hozzáférés dátuma: március 2024]. |
| [21] | U. E. Forum, „Procedural Mesh Examples,” [Online]. Available: https://forums.unrealengine.com/t/procedural-mesh-examples-free/64411. [Hozzáférés dátuma: március 2024]. |
| [22] | Wikipedia, „Object-oriented programming,” [Online]. Available: https://en.wikipedia.org/wiki/Object-oriented\_programming. [Hozzáférés dátuma: március 2024]. |
| [23] | R. Guru, „Design patterns,” [Online]. Available: https://refactoring.guru/design-patterns/catalog. [Hozzáférés dátuma: március 2024]. |
| [24] | „spdlog,” [Online]. Available: https://github.com/gabime/spdlog. [Hozzáférés dátuma: március 2024]. |
| [25] | „lemon,” [Online]. Available: https://lemon.cs.elte.hu/trac/lemon. [Hozzáférés dátuma: március 2024]. |
| [26] | „XML,” [Online]. Available: https://hu.wikipedia.org/wiki/XML. [Hozzáférés dátuma: március 2024]. |
| [27] | Wikopedaia, „Kruskal algorithm,” [Online]. Available: https://en.wikipedia.org/wiki/Kruskal%27s\_algorithm. [Hozzáférés dátuma: március 2024]. |
| [28] | Wikipedia, „Maze generation algorithm,” [Online]. Available: https://en.wikipedia.org/wiki/Maze\_generation\_algorithm#/media/File:Prim\_Maze.svg. [Hozzáférés dátuma: március 2024]. |
| [29] | Wikipedia, „Delaunay triangulation,” [Online]. Available: https://en.wikipedia.org/wiki/Delaunay\_triangulation. [Hozzáférés dátuma: március 2023]. |
| [30] | Wikipedia, „Voronoi diagram,” [Online]. Available: https://en.wikipedia.org/wiki/Voronoi\_diagram. [Hozzáférés dátuma: március 2024]. |
| [31] | K. Nahtkasztlija, „Az idegen szavak toldalékolása,” június 2009. [Online]. Available: http://www.pcguru.hu/blog/kredenc/az-idegen-szavak-toldalekolasa/5062. |
| [32] | P. Koopman, „How to Write an Abstract,” október 1997. [Online]. Available: https://users.ece.cmu.edu/~koopman/essays/abstract.html. [Hozzáférés dátuma: 20 október 2015]. |
| [33] | W3C, „HTML, The Web’s Core Language,” [Online]. Available: http://www.w3.org/html/. [Hozzáférés dátuma: 20 október 2015]. |

# Függelékek

A teljes forráskód elérhető az alábbi GitHub linken: TODO