# Pályageneráló algoritmusok

Ebben a fejezetben három darab generálási módszert fogok bemutatni, amelyek a Perlin zaj, a celluláris automata, valamint a Random Walk algoritmusok. Mindhárom algoritmust a Unity keretrendszerben implementáltam és vizsgáltam meg, hogy melyik módszer illeszkedne legjobban a játékomhoz.

## A térképgenerálás megtervezése

Egy olyan dungeon-stílusú térképet szeretnék létrehozni, amelynek a külső szélei mindig falak, amelyek nem engedik leesni a játékost a pályáról, így a játék határai egyértelműen megmaradnak, valamint lehetővé teszi a játékosok számára, hogy szabadon navigáljanak anélkül, hogy áthatolhatatlan akadályokba ütköznének, például olyan falakba, amelyek megakadályozzák őket a tárgyak begyűjtésében. Tehát olyan pályageneráló algoritmus illeszkedne a játékomhoz, amely nem generál elzárt tereket, szobákat.

A térképgenerálás logikája, valamint a menete a következőképpen történik:

Bináris használatával különböztetjük meg, hogy mi a tile és mi nem az. Az 1 bekapcsolva, a 0 pedig kikapcsolva. Az összes térképünket egy 2D-s egész számtömbben tároljuk, amelyet minden egyes funkció végén (kivéve, amikor renderelünk) visszaküldünk a felhasználónak.

Az új térkép generálása előtt a térképen a meglévő tile-ok törlődnek. Ez biztosítja, hogy az új térkép generálása üres vászonnal kezdődjön, megakadályozva az új adatok átfedését vagy összeolvadását a régi tile-okkal. Ezután egy metódus segítségével legenerálunk egy N\*N méretű tömböt, amiben az értékek a 0 vagy 1 értéket vehetik fel. Miután a tömb legenerálása befejeződött, meghívjuk a procedurális mapgeneráló algoritmusunkat, majd egy renderelő függvényt, amely az általunk kiválasztott tiletípust felfesti a térképre. A térképgenerálás folyamatábrája a 3.1.1. ábrán megtekinthető.

A képen szöveg, képernyőkép, diagram, sor látható

Automatikusan generált leírás

1.2.1.1. ábra. A térképgenerálás folyamatábrája

## A térképgenerálás logikáját megvalósító algoritmusok bemutatása

Ebben a fejezetben a térképek generálásáért felelős metódusokat fogom részletesen bemutatni.

### A GenerateArray() metódus

A HardMapGenerator osztály GenerateArray() függvényének célja egy 2D-s int[,] típusú tömb létrehozása és inicializálása, amely a térképrácsot képviseli a procedurális generáláshoz. Ez a tömb szolgál alapként a procedurális mapgeneráló algoritmusok alkalmazásához a dungeon alaprajzának generálásához.

**Paraméterek:**

* int width, int height : Ezek a paraméterek határozzák meg a térkép méreteit, specifikusan a szélességét és a magasságát.
* bool empty: Ez a bool típusú változó határozza meg, hogy hogyan inicializáljuk a tömböt. Ha az értéke igaz, akkor üresként inicializáljuk, ha false, akkor pedig teliként.

Az int[,] map = new int[width,height] sorral egy új 2 dimenziós tömböt inicializálunk. A tömb minden eleme egy egész számot reprezentálhat.

A beágyazott for ciklusok a tömb minden egyes elemén végigmennek, az x a szélességen, az y pedig a magasságon iterál.A map.GetUpperBound(0) és a map.GetUpperBound(1) a tömb dimenzióinak felső határainak megadására szolgál. A GetUpperBound(0) az első dimenzió (szélesség) maximális indexét adja vissza, a GetUpperBound(1) pedig a második dimenzió (magasság) maximális indexét.

A GetUpperBound használata általában a tömb megadott dimenziójának utolsó érvényes indexét adja vissza. Egy szélesség és magasság dimenziójú tömb esetében a GetUpperBound(0) valójában a width - 1, a GetUpperBound(1) pedig a height - 1 értéket adná vissza, mivel a tömbindexek 0-nál kezdődnek.

A belső cikluson belül az üres paraméter feltételes ellenőrzése dönti el, hogy a tömb helyét 0-val (ami üres vagy nyitott helyet jelez) vagy 1-gyel (ami kitöltött vagy blokkolt helyet jelez) töltse-e ki.

**if (empty) { map[x, y] = 0; } else { map[x, y] = 1; }**: Ez gyakorlatilag beállítja a térkép minden egyes cellájának kezdeti állapotát. Egy üres térkép (empty = true) azt jelenti, hogy minden cellát 0-ra inicializálunk, egy nem üres térkép (empty = false) pedig azt, hogy minden cellát 1-re állítunk.

**return map;:** Miután a tömb teljesen feltöltődött, visszakerül a hívó számára. Ez a tömb most a kezdeti rácsállapotként szolgál a további feldolgozáshoz, például a RandomWalk algoritmus alkalmazásához.

A GenerateArray() metódus az 1.2.1.1. ábrán látható.

A képen szöveg, képernyőkép, szoftver, képernyő látható

Automatikusan generált leírás

1.2.1.1. ábra. A GenerateArray() metódus

### A RenderMap() metódus

A HardMapGenerator osztály RenderMap() függvénye felelős a generált térkép vizuális megjelenítéséért egy Unity Tilemap-en egy adott TileBase segítségével. Ez a függvény a numerikus térképadatokat (egy 2D-s egész számtömbben tárolva) a játéktérképen lévő tényleges tile-okká alakítja át.

**Paraméterek:**

* int[,] map: Egy két dimenziós tömb, amely a dungeon alaprajzát reprezentálja, ahol az egyes cellák értéke határozza meg, hogy üres vagy tele van-e az adott cella.
* Tilemap tilemap: Ez a Unity Tilemap, amelyre a tile-ok fognak felrajzolódni.
* TileBase tile: Ez a térkép kitöltött területeinek vizuális ábrázolására szolgáló tile. (Esetemben RuleTile)

A tilemap.ClearAllTiles() paranccsal, mielőtt az új térképet lerenderelnénk, töröljük az összes meglévő tile-t a térképről. Ez kulcsfontosságú annak biztosítása érdekében, hogy a korábbi tile-ok ne maradjanak láthatóak. Ez a metódus is egymásba ágyazott for ciklusokat használ ahhoz, hogy bejárja a map tömb minden elemét. A ciklusok a tömb minden egyes tengelyén végigmennek a map.GetUpperBound(0) segítségével a szélesség (x) és a map.GetUpperBound(1) segítségével a magasság (y) esetében.

A cikluson belül a függvény minden egyes koordinátánál (x, y) ellenőrzi az értéket. Az if(map[x,y] == 1) feltétel ellenőrzi, hogy az aktuális koordinátáknál lévő cella tele van-e ( 1 az érték). Ha tele van, akkor a Tilemap megfelelő pozíciójára egy új tile kerül. A tilemap.SetTile(new Vector3Int(x,y,0),tile) parancs egy tile-t helyez el a Tilemap (x,y) pozíciójában.

A RenderMap() metódus az 1.2.2.1. ábrán látható.

A képen szöveg, képernyőkép, szoftver, Betűtípus látható

Automatikusan generált leírás

1.2.2.1. ábra. A RenderMap() metódus

## A Perlin-zaj (Perlin Noise)

A Perlin-zaj egy algoritmus, amelyet Ken Perlin hozott létre az 1980-as évek elején, és széles körben használják a játékfejlesztésben bármilyen hullámszerű anyag vagy textúra létrehozásához. Például a Perlin-zajt használhatjuk procedurális domborzati alakzatok (Minecraft szerű domborzati térkép hozható létre a Perlin-zaj algoritmus segítségével), tűzeffektek, víz és felhők létrehozásához. Ezek a hatások főleg a második és harmadik dimenzióban tükrözik a Perlin-zajt, de kiterjeszhető a negyedik dimenzióra is. Ezen kívül az algoritmus használható még az 1 dimenziós térben is, mint például egy „side-scroller” terep létrehozásához, vagy kézzel írt vonalak illúziójának megteremtésére.

Sőt mi több, ha az algoritmust a 2. vagy a 3. dimenzióra is kiterjesztjük, valamint az extra dimenziókra úgy tekintünk, mint az időre, akkor meg is tudjuk a kreált alakzatokat animálni. Az alábbiakban néhány képet láthatunk a különböző méretű zajokról és néhány felhasználási módjukról futás közben:

|  |  |  |
| --- | --- | --- |
| Zaj dimenziószáma | A nyers zaj (szürkeárnyalatos) | Felhasználási mód |
| 1 | A képen 1D perlin-zaj látható | A képen kör, vázlat, minta látható  Automatikusan generált leírás  A zaj offsetként való használata kézzel írt vonalak létrehozásához. |
| 2 | A képen szürke, képernyőkép, természet látható  Automatikusan generált leírás | A képen minta, csomagolópapír, térkép, Színesség látható  Automatikusan generált leírás  A zajt szigetek létrehozásához is lehet használni |
| 3 | A képen vázlat, fekete-fehér, minta, táj látható  Automatikusan generált leírás | A képen térkép látható  Automatikusan generált leírás  Egy módosított Perlin-zaj implementációval dombok, völgyek és barlangok hozhatók létre |

Amint láthatjuk, a Perlin-zaj számos természetben előforduló jelenségre alkalmazható.

A Perlin-zaj gradiens zajgenerálási technikát alkalmaz, ami a pontok közötti természetesebb és simább átmenetet eredményez. Ez a megközelítés élethűbbnek tűnő tájképet hoz létre. Az algoritmus egy rácshálós keretrendszerben működik, ahol a rácsháló minden egyes metszéspontjához egy gradiensvektor tartozik. Ezek a vektorok döntő fontosságúak a zaj mintázatának és irányítottságának kialakításában.

A Perlin-zaj egyik fő jellemzője a rácspontok közötti interpoláció alkalmazása, ami hozzájárul a jellegzetes simasághoz. Ez a sima átmenet éles ellentétben áll a teljesen véletlenszerű zajgenerálásra jellemző hirtelen változásokkal. A Perlin-zajt eredetileg 3D-s grafikához fejlesztették ki, de a 2D-s alkalmazásokban is széles körben használják, többek között a videojátékok terepgenerálásában és a procedurális textúrák létrehozásában.

A generált minták összetettségének fokozása érdekében az algoritmus gyakran alkalmaz rétegezési technikát, amely több "oktávnyi" zajt tartalmaz. Minden egyes oktáv külön frekvenciával és amplitúdóval működik, és amikor ezeket a rétegeket kombinálják, bonyolultabb és változatosabb mintákat hoznak létre. Az algoritmus állítható paramétereket kínál, mint például a frekvencia, az amplitúdó és a perzisztencia, ami lehetővé teszi a generált zaj megjelenésének részletes szabályozását, és a terep vagy a textúra testre szabott szimulációját.

A játékokban és a számítógépes grafikában való alkalmazásán túl a Perlin-zaj elterjedt más területeken is, mint például tudományos szimulációk készítése, ahol olyan természeti jelenségeket modellez, mint a felhőképződmények, vagy egy táj jellegzetességei.

### A Perlin-zaj algoritmus implementálása és vizsgálata

A PerlinNoiseDungeon() metódus a Perlin-zaj segítségével módosítja a rácsos térképet, hogy simább átmeneteket hozzon létre a kitöltött és üres terek között.

**Paraméterek:**

* int[,] map : A módosítandó kezdeti térképtömb.
* float modifier: Egy skálázási tényező, amely a Perlin zajfüggvény frekvenciáját állítja be. Az értéke 0 és 1 között van.

A függvény a megadott térképmező minden egyes celláján végigmegy a beágyazott for ciklusok segítségével. Ha az aktuális cella a térkép szélén van (x == 0, y == 0, vagy az x vagy y utolsó indexein), a függvény a cella értékét 1-re állítja, így biztosítva, hogy az egy fal maradjon.

A Mathf.PerlinNoise(x\*modifier, y\*modifier) függvényhívás egy Perlin-zajértéket generál (0 és 1 között) a modifier által beállított koordinátákhoz. A modifier a bemeneti koordináták skálázására hat, ami befolyásolja a zajfrekvenciát. A Mathf.RoundToInt() metódus az eredményül kapott Perlin zaj lebegő értékét a legközelebbi egész számra kerekíti annak eldöntésére, hogy a cella fal vagy üres tér legyen.

A függvény visszaadja a módosított térképtömböt az új terepjellemzőkkel együtt, készen állva a renderelésre.

A PerlinNoiseDungeon() metódus az 1.3.1.1. ábrán megtekinthető.

A képen szöveg, képernyőkép, szoftver, képernyő látható

Automatikusan generált leírás

1.3.1.1. ábra. A perlin zajt megvalósító metódus

A továbbiakban a Perlin-zaj által generált térképek vizsgálatáról lesz szó.

Mindegyik generált térkép 100x100-as mátrixnak felel meg, amelyet 1-esekkel töltünk fel. A modifier változó értékét fogom növelni a vizsgálat során, az első érték a 0,05, a második a 0,10 és ez egészen a 0,30 értékig fog növekedni. A generált térképek az 1.3.1.2 ábrán láthatóak.

A képen képernyőkép, szöveg, minta látható

Automatikusan generált leírás

1.3.1.2. ábra. A PerlinNoiseDungeon() metódussal generált térképek különböző modifier értékkel

Ahogy a generált térképeken is láthatjuk, az alacsonyabb modifier értékek általában nagyobb, összefüggőbb régiókat eredményeznek, amelyek szélesebb földtani jellemzőket, például széles barlangokat vagy nagy nyílt területeket szimulálnak. A magasabb módosító értékek széttöredezettebb és kuszább tereket hoznak létre, amelyek bonyolult alagútrendszereket vagy sűrű barlanghálózatot eredményeznek.

Ezek a különbségek úgy befolyásolhatják a játékmenetet, hogy a nagyobb, nyílt területek megkönnyíthetik a navigációt, könnyedebben teljesíthetőek lesznek a pályák, míg a bonyolultabb, magasabb modifier értékkel generált pályák nagyobb kihívást jelentőek lesznek, hosszabb játékidőt és frusztrációt eredményezhetnek.

Ezek a legenerált pályák nem feleltek meg az általam elvárt kinézetnek, mivel tartalmaznak olyan elzárt részeket, amelybe a játékos nem tud bemenni. Olyan játékoknál alkalmaznám ezt a fajta pályagenerálást, amelyeknél implementálva van az, hogy a játékos eltüntetheti, széttörheti a blokkot, mondjuk egy csákánnyal, vagy bombával.

## A celluláris automata (cellular automaton)

A celluláris automata, egy rácsalapú rendszerben működő számítási modell, amely egyszerűségében és összetettségében egyaránt lenyűgöző. Minden egyes sejt ezen a rácshálózaton két állapotban létezhet, amelyek az "él" vagy a "halott" állapotok. E sejtek fejlődését egyik generációról a másikra egy szabályrendszer határozza meg, amely jellemzően a szomszédos sejtek állapotán alapul. Ez a felállás, bár összetevőit és szabályait tekintve egyszerű, az azonos szabályokat követő sejtek együttes kölcsönhatása révén rendkívül bonyolult mintázatokat képes létrehozni.

A celluláris automaták egyik legismertebb példája Conway „Game of Life” című műve. Ez egy kiváló példa arra, hogy az alapvető szabályok hogyan eredményezhetnek összetett viselkedést, annak ellenére, hogy ez egy „zero-player” játék, ami azt jelenti, hogy a fejlődését a kezdeti állapota határozza meg, és nincs szüksége emberi játékostól származó cselekedetre, inputra. Az ember úgy lép kapcsolatba a játékkal, hogy létrehoz egy kezdeti konfigurációt, és megfigyeli, hogy hogyan fejlődik. A celluláris automata, valamint a „Game of Life” játék négy szabálya a következő:

1. Minden olyan élő sejt, amelynek kettőnél kevesebb szomszédja van, „meghal” (ezt nevezzük alulnépesedésnek vagy veszélyeztetettségnek).
2. Minden olyan élő sejt, amelynek háromnál több szomszédja van, „meghal” (ezt nevezik túlnépesedésnek vagy túlzsúfoltságnak)
3. Minden élő sejt, amelynek két vagy három élő szomszédja van, változatlanul tovább él a következő generációig.
4. Minden halott sejt, amelynek pontosan három élő szomszédja van, életre kel.

A kezdeti minta képezi a rendszer "magját". Az első generáció úgy jön létre, hogy a fenti szabályokat egyszerre alkalmazzák a mag minden sejtjére - a születések és halálozások egyszerre történnek, és azt a diszkrét pillanatot, amikor ez megtörténik, néha ticknek nevezik. (Más szóval, minden egyes generáció az előző generáció színtiszta függvénye.) A szabályok ismételt alkalmazása további generációk létrehozásához folytatódik.

A képen tér, diagram, képernyőkép, szöveg látható

Automatikusan generált leírás

1.3.1.1. ábra A celluláris automata szabályai szemléltetve

A celluláris automata rugalmassága a testreszabhatóságban rejlik. A fejlesztők a szabályokat és az állapotokat az egyedi igényekhez igazíthatják, befolyásolva olyan szempontokat, mint a térkép sűrűsége és az útvonalak összekapcsolhatósága. A véletlenszerűség beépítésének képessége ellenére az automata determinisztikus jellege biztosítja az azonos kezdeti feltételekből származó konzisztens eredményeket, ami különösen hasznos a reprodukálható szintek létrehozásához.

A celluláris automata nem csak a szintek strukturálásában segít, hanem a vizuális látványt is fokozza, olyan mintákat generálva, amelyek esztétikailag szépek és a játékmenet szempontjából is praktikusak.

A képen vázlat, rajz látható

Automatikusan generált leírás

1.3.1.2. ábra Az automata által generált minták

Ezzel a tulajdonságával hatékony eszközzé válik a fejlesztők számára, akik dinamikus és megnyerő környezetet kívánnak létrehozni a 2D platformer játékokban, és az egyedi, változatos szintek létrehozásával jelentősen növeli a játék újra játszásának az esélyét.

### A celluláris automata algoritmus implementálása és vizsgálata

Ahhoz, hogy a VonNeumannCellularAutomata() metódus létrehozza a randomizált térképünket, először egy módosított térképet kell kreálnunk. A módosítás pedig nem más, mint a randomizáció.

A ModifiedMapForCellularAutomata() metódus megteremti egy celluláris automata folyamat előfeltételeit egy véletlenszerű kezdeti feltételeket tartalmazó térkép létrehozásával, amely aztán a következő lépésekben a von Neumann celluláris automata szabályaival lesz fejlesztve. Funkcionalitását tekintve ugyan azt csinálja, mint a GenerateArray() metódus, de miközben inicializálja a térképtömböt, nem egyenletesen tölti ki a térképteret, hanem randomizálva. Minden belső cellához véletlenszerű állapotot rendel (0 vagy 1) a fillPercent által meghatározott valószínűség alapján. A ModifiedMapForCellularAutomata() metódus az 1.4.1.1. ábrán megtekinthető.

A képen szöveg, képernyőkép, szoftver, képernyő látható

Automatikusan generált leírás

1.4.1.1. ábra A ModifiedMapForCellularAutomata() metódus

A VonNeumannCellularAutomata() metódus a von Neumann szomszédságon alapuló celluláris automata technikát használja egy térkép fejlesztésére.

**Paraméterek**:

* int[,] map : Ez az a térkép, amelyet a függvény módosítani fog.
* int smoothCount: Az iterációk (vagy generációk) száma, amelyeken a celluláris automata folyamat keresztülmegy.

A külső for ciklus smoothCount-szor iterál, és a celluláris automata folyamatot iterációnként egyszer hajtja végre. Minden egyes iteráció egy generációt jelent. A belső egymásba ágyazott for ciklusok a map tömb minden egyes celláján végigmennek a map.GetUpperBound(0) segítségével a szélességhez és a map.GetUpperBound(1) segítségével a magassághoz. A függvény meghívja a GetVNSurroundingTiles() függvényt, hogy kiszámítsa az (x, y) pozícióban lévő cellát körülvevő kitöltött szomszédos cellák számát a von Neumann-szabályok alapján.

**Cellaállapot-frissítési szabályok:**

* Kitöltött cella: Ha egy cellának több mint 2 környező kitöltött cellája van, akkor maga is kitöltött cellává válik (map[x, y] = 1).
* Üres cella: Ha egy cellának 2-nél kevesebb kitöltött cellája van, akkor üres lesz (map[x, y] = 0).
* Nincs változás: Ha egy cellának pontosan 2 szomszédja van, az állapota változatlan marad. Ez a szabály stabilitást biztosít a struktúrának, fenntartva a jelenlegi komplexitást.

A megadott számú iteráció elvégzése után a függvény visszaadja a továbbfejlesztett mapot, amely tükrözi a celluláris automata által végrehajtott változásokat. A VonNeumannCellularAutomata() metódus az 1.4.1.2. ábrán látható.

A képen szöveg, képernyőkép, szoftver, Multimédiás szoftver látható

Automatikusan generált leírás

1.4.1.2. ábra. A VonNeumannCellularAutomata() metódus

A fenti metódusban használt GetVNSurroundingTiles() metódus célja a környező "kitöltött" cellák számának kiszámítása a von Neumann szomszédság alapján, amely csak a négy szomszédos cellát veszi figyelembe a fő irányokban: felette, alatta, balra és jobbra. A von Neumann szomszédságot reprezentáló ábra az 1.4.1.3. ábrán láthtató.

A képen szöveg, képernyőkép, Betűtípus, Téglalap látható

Automatikusan generált leírás

1.4.1.3. ábra. A von Neumann szomszédság

A cellák kiszámítását végző GetVNSurroundingTiles() metódus az 1.4.1.4. ábrán látható.

A képen szöveg, képernyőkép, szoftver látható

Automatikusan generált leírás

1.4.1.4. ábra. A GetVNSurroundingTiles() metódus

A továbbiakban a celluláris automata által generált térképek vizsgálatáról lesz szó.

Mindegyik generált térkép 100x100-as mátrixnak felel meg, amelyet a ModifiedMapForCellularAutomata() metódussal véletlenszerűen töltünk fel 1-esekkel és 0-ákkal. Először a requiredFillPercent változó fixálásával és a smoothIterations változó növelésével fogom vizsgálni a legenerált térképeket, majd a smoothIterations változó értékét fixálom és a requiredFillPercent értékét fogom növelni. A generált térképek az 1.4.1.5. ábrán megtekinthetőek.

A képen képernyőkép, szöveg látható

Automatikusan generált leírás

1.4.1.5. ábra. Fixált requiredFloorPercent értékkel és változó smoothIterations értékkel generált térképek

A fentebb látható ábrán olyan térképek vannak generálva, ahol a requiredFloorPercent értéke fixen 40, a smoothIterations értéke pedig 10,20 és 30. Mint látni is lehet, ha növeljük a smoothIterations értékét, akkor a térképen a teli cellák száma növekszik, bár annyira nem látványos a különbség a legenerált térképek között. A fixált smoothIterations értékkel és a változó requiredFloorPercent értékkel generált térképek az 1.4.1.6. ábrán láthatóak.

A képen képernyőkép, szöveg látható

Automatikusan generált leírás

1.4.1.6. ábra. Fixált smoothIterations értékkel és változó requiredFloorPercent értékkel generált térképek

Az 1.4.1.6. ábrán pedig azok a térképek lettek legenerálva, amelyeknél a smoothIterations értéke fixen 20, a requiredFloorPercent értéke pedig 50,55 és 60. Itt már sokkal drasztikusabb a különbség a legenerált térképek között, minél nagyobb a requiredFloorPercent értéke, annál több olyan cella lesz, amelynek az értéke 1, azaz sokkal sűrűbb lesz a térképünk.

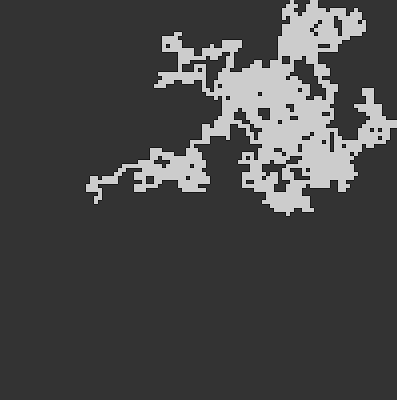
A gond ezzel a procedurális mapgeneráló algoritmussal is az, hogy nem lehet garantálni, hogy olyan térképeket generáljunk, amelyben nincsenek olyan részek, amelyek teljesen elzártak. Ezt az algoritmust inkább 3D-s, Minecraft szerű játékokban tudnám elképzelni, ahol mondjuk az értékes tárgyak elhelyezkedését lehetne meghatározni vele a föld alatt.

## A Random Walk algoritmus

A Random Walk algoritmus egy viszonylag egyszerű, de hatékony módszer a procedurális térképgenerálásra, különösen alkalmas kétdimenziós rácsalapú térképekhez. Közismert arról, hogy természetesnek tűnő alakzatokat hoz létre, és összetettebb procedurális generáló rendszerek első lépéseként szolgálhat. Kétdimenziós rácshálózattal összefüggésben a véletlen bolyongást néha „részeges sétának” (Drunkard’s walk-nak) is nevezik, és az elnevezés magától értetődő, ha figyelembe vesszük a működését:

1. Hozzunk létre egy N\*M méretű rácshálózatot.
2. Válasszunk egy véletlenszerű kezdő pozíciót a rácshálón.
3. Állítsuk be a pozíciót „visited”-re. (Azaz látogatottra.)
4. Válasszunk egy új véletlenszerű pozíciót az aktuális pozíciótól egyetlen cella elmozgatásával (balra / fel / jobbra / le).
5. Ha a pozíció amire érkezünk érvényes (a pozíció nem esik a rácshálón kívülre), akkor ezt az új pozíciót állítsuk be az aktuális pozíciónak.
6. Menjünk vissza a 4. ponthoz, és addig ismételjük, amíg a befejezési feltétel teljesül (például az ismétlések száma).

A modellezés alapvetően egy olyan egyed, amely minden egyes időlépésnél kiszámíthatatlanul mozog bármilyen irányba. Az entitás a korábban meglátogatott cellákba is visszamehet, így a korábbi iterációk nem befolyásolják az aktuális iterációkat, ami a „Random Walk-ot” sztochasztikus / memória nélküli folyamattá teszi. Sőt mi több, garantálja, hogy a térkép teljesen összefüggő lesz, mivel csak a szomszédos cellák között mozog. Ez az algoritmus ideális a játékok barlangjainak és túlvilágainak létrehozására, mivel képes összefüggő és terjedelmes térképeket létrehozni.



1.4.1.1. ábra. Egy Random Walk algoritmussal létrehozott összefüggő térkép

### A Random Walk algoritmus implementálása és vizsgálata

A RandomWalkCave() metódus a térképrács módosítására szolgál a Random Walk algoritmuson alapuló procedurális generálási technikával. Ez a függvény egy barlangszerű struktúrát váj ki egy adott kétdimenziós tömbben.

**Inicializáció:**

Random seed inicializálása: A seed.GetHashCode() paranccsal egy random magot inicializálunk. Ez a mag biztosítja, hogy a generált véletlenszámok reprodukálhatóak legyenek, ha ugyan azt a magot használjuk, ami hasznos a hibakeresésnél vagy bizonyos térképek újragenerálásánál

**Kiinduló pozíció kiválasztása:** A funkció véletlenszerűen választ ki egy kezdő pozíciót (floorX, floorY) a térkép határain belül, de nem a széleken, hogy biztosítsa, hogy legyen hely a barlang bővítésére. Innen fog kezdődni a véletlen bolyongás.

**reqFloorAmount:** A reqFloorAmount a térképen lévő összes cellaszám százalékaként (requiredFloorPercent) kerül kiszámításra. Ez a változó határozza meg, hogy hány cellát kell átalakítani teliből (value = 1) üressé (value = 0) a barlanggenerálás befejezéséhez.

**floorCount inicializáció:** A floorCount nulláról indul, és minden alkalommal növekszik, amikor egy cellát teliből üressé alakítunk.

A while ciklus addig megy, amíg a floorCount értéke el nem éri a reqFloorAmount értékét. A while cikluson belül egy switch utasítást alkalmazok az irányok kezelésére. Az út generálásának a logikája a következő:

* A switch utasítás minden egyes esete egy iránynak felel meg.
* Mielőtt az algoritmus bármilyen irányba mozogna, biztosítjuk, hogy a lépés nem halad ki a térkép határain kívülre.
* 0.eset: Növeljük a floorY értékét, ha a felfelé mozgás a határokon belül marad.
* 1.eset: Csökkentjük a floorY értékét, ha a lefelé mozgás a határokon belül marad.
* 2.eset: Növeljük a floorX értékét, ha a jobbra mozgás a határokon belül marad.
* 3.eset: Csökkentjük a floorX értékét, ha a balra mozgás a határokon belül marad.

Minden esetben megnézzük, hogy a jelenlegi pozíció teli-e. Ha az, akkor átkonvertáljuk üresre, és a procedúra addig folytatódik, amíg el nem érjük a kívánt cellák számát.

A RandomWalkCave() metódus az 1.5.1.1. ábrán megtekinthető.

---------------------Ide majd a LaTeX formátumban a kódot beilleszteni! -----------------

A továbbiakban a Random walk algoritmussal generált térképek vizsgálatával fogok foglalkozni.

Mindegyik generált térkép 100x100-as mátrixnak felel meg, amelyet 1-esekkel töltünk fel. Az első vizsgálat a requiredFloorPercent érték növelésével fog történni. Hat esetet fogok vizsgálni, melyeknél a változó értéke rendre 20,30,40,50,60 és 70 lesz. A legenerált térképek a megadott requiredFloorPercent értékkel az 1.5.1.2. ábrán megtekinthetőek.

A képen szöveg, képernyőkép, Téglalap látható

Automatikusan generált leírás

1.5.1.1. ábra. Változó requiredFloorPercent értékkel generált térképek

Ahogy az ábrán is láthatjuk, minél nagyobb a requiredFloorPercent változó értéke, annál több alagutat váj ki magának az algoritmus, azaz annál több olyan cella keletkezik, melynek az értéke 0. Azt is észrevehetjük, hogy nem keletkezik olyan rész a térképen, amely el lenne zárva, azaz teljesen összefüggő mapot kapunk minden egyes esetben.

Az algoritmus lépésszámát, valamint a futási idejét is vizsgáltam, ezek a mérések az 1.5.1.3. ábrán láthatóak.

A képen szöveg, képernyőkép, Betűtípus, szám látható

Automatikusan generált leírás

1.5.1.2. ábra. Az algoritmus lépésszámai és futási idői a requiredFloorPercent függvényében

Ahogy azt az ábráról leolvashatjuk, minél nagyobb a requiredFloorPercent értéke, annál nagyobb lesz a lépésszáma az algoritmusnak, valamint a futási ideje is nőni fog. Volt néhány olyan eset, ahol nagyobb requiredFloorPercent értéknél kevesebb futási idő, valamint lépésszám volt, de ez elenyésző mennyiségű esetben fordult elő. A lépésszám és a requiredFloorPercent korrelációja az 1.5.1.4. grafikonon megtekinthető.

A képen szöveg, Diagram, sor, diagram látható

Automatikusan generált leírás

1.5.1.3. ábra. A lépésszám és a requiredFloorPercent kapcsolata

A következő vizsgálat során változó méretű térképeket fogok generálni, viszont a requiredFloorPercent értékét 45-re fogom fixálni. Ezen térképek legenárálásának szintén a lépésszámát, valamint az idejét fogom vizsgálni. A térképek rendre 30 x 30, 40x 40 és 50 x 50 méretűek lesznek. A legenerált térképek az 1.5.1.4. ábrán láthatóak.

A képen szöveg, képernyőkép, térkép látható

Automatikusan generált leírás

1.5.1.4. ábra. Fixált requiredFloorPercent értékkel, de változó térképmérettel generált mapok

A generáláshoz szükséges futási idő, valamint a lépésszám az 1.5.1.5. ábrán látható.

A képen szöveg, képernyőkép, Betűtípus, szám látható

Automatikusan generált leírás

1.5.1.5. ábra. A fixált requiredFloorPercent értékkel és változó térképmérettel rendelkező mapok generálásának lépésszáma és futási ideje

További vizsgálatokat fogok az algoritmuson elvégezni, mivel jelenleg egyenletes eloszlást használ az algoritmus, ahol minden fő iránynak (felfelé, lefelé, balra és jobbra) egyenlő a valószínűsége, amely 0,25. Ez egy torzítatlan sétát eredményez, amely egyik irányt sem részesíti előnyben a többivel szemben. Az egyes irányok valószínűségeinek megváltoztatásával különböző hatásokat és mintákat hozhatunk létre a térképgenerálás során.

Az egyik ilyen vizsgálat során nagyobb valószínűséget fogok rendelni az előző lépéssel megegyező irányba történő előrehaladáshoz és kisebb valószínűséget az irányváltoztatáshoz. Ehhez módosítanom kell az egyes lépések kiválasztásának a logikáját.

Először is egy olyan metódust kell implementálni, amely egy nem egyenletes eloszlás alapján választ irányt, majd az egyszerű rand.Next(4) parancsot lecserélni erre a metódusra. A metódusnak a neve a ChooseDirectionBias(), amely az 1.5.1.6 ábrán megtekinthető.

A képen szöveg, képernyőkép, szoftver látható

Automatikusan generált leírás

1.5.1.6. ábra. A ChooseDirectionBias metódus

Az első lépés a valószínűségi tömb inicializálása, amely lebegőpontos számokat fog tartalmazni, és a hossza megegyezik a directions tömb hosszával. Ez a tömb tárolja az egyes irányok valószínűségét. Ezután inicializálunk egy bias nevű változót, amelynek a változtatásával kontrollálhatjuk, hogy mennyire legyen hajlamos az algoritmus ugyanabba az irányba haladni. Az első for cikluson belül számítjuk ki az úgynevezett „elfogult” valószínűséget (biased probability). Ha egy irány megegyezik a lastDirection irányával, akkor nagyobb valószínűséget kap (bias paraméter, jelenleg 70%-ra állítva). Az összes többi irány egyenlő arányban részesül a fennmaradó 30%-os valószínűségből.

A következő lépés a random választás folyamatának az implementálása. A rand.NextDouble() parancs egy véletlenszerű lebegőpontos számot generál 0 és 1 között, amely tulajdonképpen egy „roll”-ként működik, hogy eldöntse, melyik irány lesz választva. A második for cikluson a probabilities tömbön iterálunk végig, és kiszámítjuk a kumulatív valószínűséget. Ezután az irányok kiválasztása következik annak ellenőrzésével, hogy a véletlenszerű „roll” az egyes irányok együttes valószínűségi tartományába esik-e. Az első olyan irány lesz kiválasztva, ahol a kumulatív valószínűség meghaladja a „roll” értékét.

A metódus return értékként visszaadja a kiválasztott irányt az eredeti irányok tömbjének megfelelő indexeként.

A továbbiakban a ChooseDirectionBias() metódus beépítésével generált térképeket fogom vizsgálni rögzített n x n méretű pályákon, majd változó méretű pályákon és ezek legenerálásának a futási idejét, valamint lépésszámát is vizsgálni fogom. A vizsgálat során a bias változó értékét fogom módosítani a rögzített n x n méretű pályákon. A requiredFloorPercent értéke rögzítve lesz, a 45-ös értéken.

Mindegyik generált térkép 100x100-as mátrixnak felel meg, amelyet 1-esekkel töltünk fel. Három esetet fogok vizsgálni, ahol a bias változó értékét fogom módosítani. A bias paraméter értéke rendre 0.6, 0.7 és 0.8 lesz. A generált térképek az 1.5.1.7. ábrán láthatóak.

A képen szöveg, képernyőkép, térkép látható

Automatikusan generált leírás

1.5.1.7. ábra. Változó bias paraméterekkel generált térképek

Az ábráról, amely a generált térképeket különböző bias értékekkel szemlélteti, miközben a requiredFloorPercent és a térkép mérete állandó, több következtetést is levonhatunk a bias paraméter értékének változtatásának hatásáról.

**A bias érték hatása:**

* **0.6-os érték**: Ezzel az értékkel a pályák kevesebb egyenes vonallal rendelkeznek, egy olyasmi térképet hoz létre, amely kevesebb falat és egy nagy nyitott területet tartalmaz.
* **0.7-es érték**: A bias érték növelésével összefüggőbb mintázat kezd kialakulni, az ösvények több és egyenesebb kapcsolatokat kezdenek kialakítani a területek között.
* **0.8-as érték**: Ezen a szinten a térkép sok egyenes útvonalat mutat, több kisebb nyitott területtel, amelyeket ezek a kiterjedtebb útvonalak kötnek össze.

A térképek generálásának lépésszáma, valamint futási ideje az 1.5.1.8. ábrán látható.

A képen szöveg, képernyőkép, Betűtípus, szám látható

Automatikusan generált leírás

1.5.1.8. ábra. A változó bias értékkel legenerált térképek lépésszáma és futási ideje

Mint láthatjuk, alacsonyabb bias érték esetén az algoritmus gyakrabban változtatja az irányt, ami több elfordulást eredményez. Ez összetettebb útvonalakat hoz létre, amelyek általában további lépéseket igényelnek ahhoz, hogy elérjék a térkép szükséges alapszázalékát, mivel az útvonal megfordulhat önmagában, vagy kevésbé közvetlen útvonalon haladhat.

A továbbiakban a változó méretű térképek generálásának a vizsgálatával fogok foglalkozni. Szintén megvizsgálom a futási időt, valamint a lépésszámot. A térképek mérete rendre 50x50, 60x60 és 70x70 méretűek lesznek, a bias változó értékét, valamint a requiredFloorPercent értékét is állandóra állítom. A bias értéke 0.7, a requiredFloorPercent értéke pedig 45 lesz. Az ily módon generált térképek az 1.5.1.9. ábrán, a hozzájuk tartozó lépésszám, valamint futási idő az 1.5.1.10. ábrán látható.

A képen szöveg, képernyőkép, térkép látható

Automatikusan generált leírás

1.5.1.9. ábra. Különböző méretű térképek legenerálása állando bias értékkel

A képen szöveg, képernyőkép, Betűtípus, szám látható

Automatikusan generált leírás

1.5.1.10. ábra. A különböző méretű térképek generálásához szükséges lépésszám, valamint futási idő

Ahogy az várható volt, minél nagyobb a térkép területe, annál több lépésszám szükséges a kívánt requiredFloorPercent értékének eléréséhez, valamint a futás idő is több lesz.

Úgy vélem, ez az algoritmus felelt meg leginkább az általam felállított követelményeknek, hiszen mindig olyan pályákat generál, amely összefüggő, a térkép szélei mindig falak, így a játékos szabadon navigálhat anélkül, hogy törődnie kellene a leeséssel, nincsenek olyan részek, ahová a játékos ne tudna eljutni, hiszen nincsenek elzárt terek. Ahhoz viszont, hogy a játékos eljusson mindenhová, egy bizonyos játékmechanikát fogok bevezetni, ami nem más mint, a grappling hook.