# Virtuális színterek procedurális generálása és optimalizálási problémáinak vizsgálata

Bordás Milán

2023. december 8.

# Tartalomjegyzék

# Ábrák jegyzéke

# 1. fejezet

# Bevezető

A színterek kulcsfontosságú szerepet töltenek be a digitális világban. Egy film, játék, animáció nem lenne teljes egy megfelelően kialakított helyszín, pálya, háttér nélkül. Emiatt szükséges olyan módszereket kialakítani, melyek effektív eszközként szolgálnak ezen elemek létrehozására. A szakdolgozatom eszközként a procedurális generálást taglalja.
A színterek procedurális generálása sok mindent magába foglal. Ha egy könyv leírása alapján gyurmából elkészítünk egy csatateret, lényegében generáltunk procedurálisan egy színteret. Az elkészítendő objektum nem is igazán lényeges, csak az, hogy miként készítettük el. Jelen esetben nem valódi tárgyakat készítünk el, hanem digitális adatokat generáltatunk a számítógéppel, amit megfelelő környezetben színtérként tudunk értelmezni.
Több generálási módszert tárgyal a szakdolgozatom, leginkább videójátékokban használatos pályák, hátterek előállítására. A célterülettől függően két külön esetet vizsgálunk : 2 dimenziós és 3 dimenziós színtereket előállító algoritmusokat.

# 2. fejezet

# Színterek típusai

A színterek egyik legalapvetőbb tulajdonsága a dimenzióinak száma.

2.1. ábra. Felülnézetes színtér
Heroes III

2.2. ábra. Oldalnézetes színtér
Terraria

Példa 2D színterekre

# 3. fejezet

# Algoritmusok

Az algoritmusok egy fontos tulajdonsága, hogy milyen helyzetben használhatóak, milyen problémákra adnak megoldást. A már említett két eset szerencsére nem különül el egymástól annyira, hogy teljesen új megoldási módszereket kelljen kialakítani.

## 3.1. Kétdimenziós színterek algoritmusai

Ezen algoritmusok egyszerűbbnek tekinthetőek a színterek kontextusában.

### 3.1.1. Wave function collapse

A "WaveFunctionCollapse" nevű algoritmust először githubra publikálta egy felhasználó [1]. Az algoritmus nevét a szerző a "hullámfüggvény összeomlásáról" nevezte el, ami egy kvantummechanikai kifejezés. Magának az algoritmusnak nincs kapcsolata ezzel a jelenséggel, de az ihlet onnan jött.
Az algoritmus eredeti ötlete, hogy egy inputként megadott bitmap pixeljeit felhasználva generálunk egy, az eredetihez lokálisan hasonló általában nagyobb bitmapet. A módszer azonban nem korlátolt csak bitmapekre: egy adott képet is fel lehet fogni egy bitmapként, amit tetszőlegesen darabolhatunk fel, így lényegében bármilyet képre elvégezhetjük a lépéseket. Ez kifejezetten hasznos több területen is. Egy ilyen felhasználási módszer amit a forrás is említ, hogy az algoritmussal csempézni is lehet a teret.

#### Az algoritmus működése

1. Beolvassuk az input kép összes pixeljét, majd kialakítunk egy paraméter által megadott $NXM$-es rácsot, feldarabolva ezzel a képet téglalapokra, hívjuk őket celláknak.

2. A cellákhoz hozzárendeljük a pozíciójukat, majd meghatározunk egy szomszédsági kapcsolatot a többi cellával. A szomszédokat az alábbiként definiáljuk:

    - Egy cella szomszédja egy másik cella, ha az az eredeti input képen a cellától Nyugatra, Keletre, Északra vagy Délre van.

[1] https://github.com/mxgmn/WaveFunctionCollapse

- A cellákat egy tóruszon képzeljük el. Ez azt jelenti, hogy a képen a balszélső celláknak a szomszédja a jobbszélső vele egy sorban lévő cella is. Persze ez oda-vissza, és fenti-lenti cellákra is igaz. Erre példa a 3.1 ábra.
- Paraméterként megadott igaz/hamis érték szerint egy cella önmagával szomszédos lehet még akkor is, ha az inputon ez nem így van.

Az output cellákat úgy helyezzük el, hogy egy cella csak olyan helyre kerülhet, ahol a körülötte lévő cellák vagy üresek, vagy mindegyike a szomszédja.

3. Kiválasztjuk, hogy milyen cellát szeretnénk letenni a rácsra. Ez a kiválasztás egy súlyozott táblával történik, ahol minden cella súlya, az input képen való előfordulásának száma.

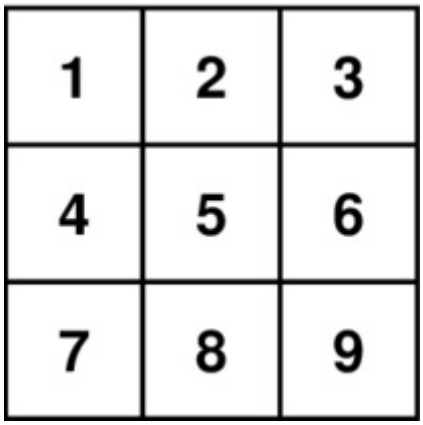
4. A cellákat az alábbi eljárások egyikével letesszük a cellákat egy $KXL$-es outputra:

   - Horizontálisan / vertikálisan, azaz elindulunk egy soron / oszlopon és folytonosan tesszük le a szabályok szerint a cellákat.
   - Véletlenszerűen, azaz minden cellát egy olyan véletlenszerűen választott helyre teszünk, ahol még szabad a hely
   - Entrópiát számolunk, és mindig a legalacsonyabb entrópiájú helyre tesszük le a cellát. Ezt az entrópiát az üres cellákra számoljuk ki amit a

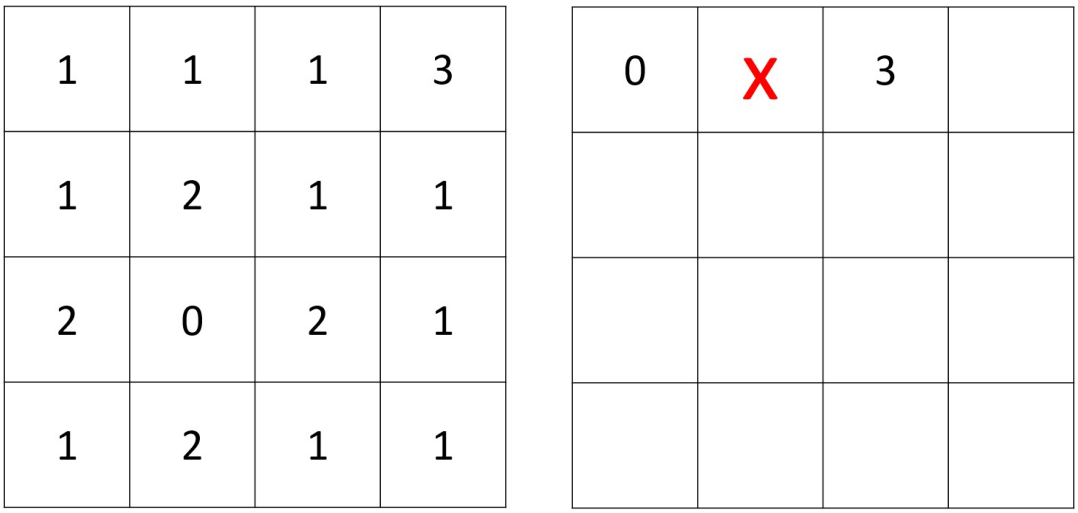
     $$\sum_{i=1}^{n} P_i * \log 1/P_i$$

     képlettel kapunk, ahol $P_i$ az adott helyre letehető $i$-edik cellának a letehetési valószínűsége. A letehetési valószínűség: az adott cella súlya, osztva az összes cella számával.
     Ennek az eljárásnak nagy a költsége, így ennek optimalizálásával később majd foglalkozunk.

5. Addig ismételjük a 3. ponttól az algoritmust, míg le nem fedtuk az outputot, vagy el nem akadtunk.
   Az elakadás egy probléma ami akkor jelenik meg, ha olyan üres rácshoz érünk, melynek szomszédjai kizárják egymást. Ekkor az algoritmust újrakezdjük, vagy leállunk. Egy ilyen elakadási helyzetre példa a 3.2 ábra.




3.1. ábra. Példa a szomszédos cellákra
Az 1-es cella szomszádja a 2-es, 3-as, 4-es, 7-es cella.

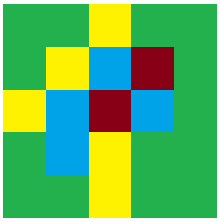



Kiinduló tábla Elakadás az outputon

3.2. ábra. Példa elakadási helyzetre
A hármas cella mellé csak egyes cella, a nullás cella mellé csak kettes cella kerülhet, így a kettő között nem lehet semmilyen cella.

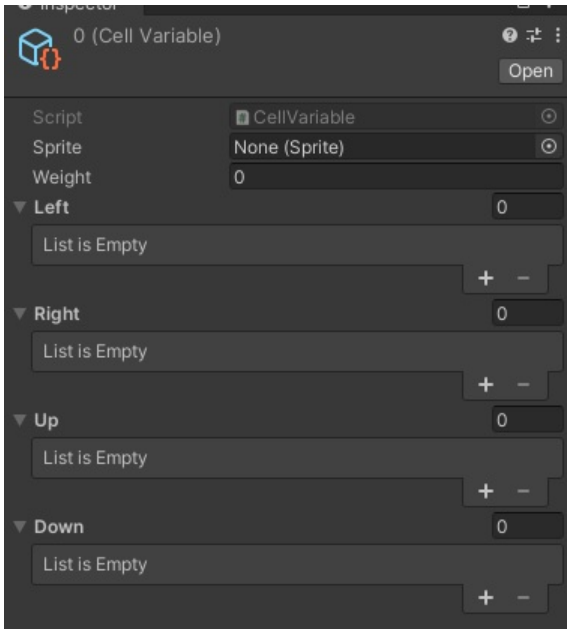
### 3.1.2. Implementáció

Az implementációhoz először választanunk kell egy képet. Válasszunk egy egyszerű, pár különálló blokkból álló képet például a képen láthatót. Ha nem jól elválasztható blokkokból áll a kép, akkor az algoritmusunkat nem tudjuk használni. Látható, hogy

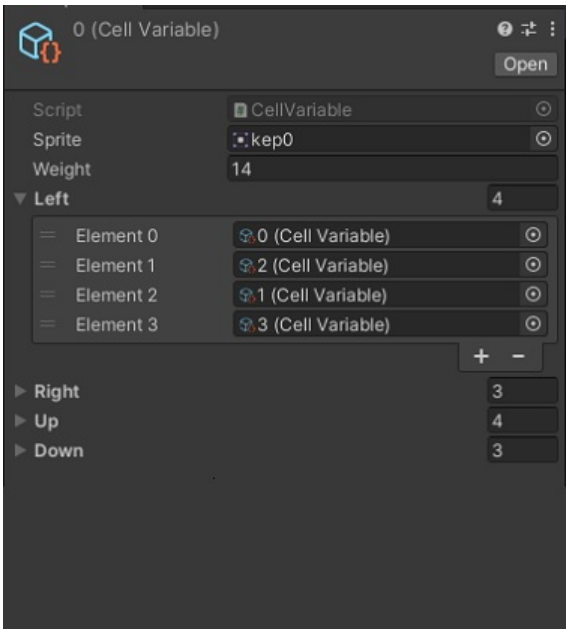


3.3. ábra. Egyszerű input kép

a kép 4 elkülöníthető blokkból áll, zöld, sárga, kék, piros blokkokból. Gondoljunk ezekre a blokkokra, úgy mintha virágok lennének egy mezőn. A zöld blokkok jelölik a füvet, a színesek pedig a virágokat. Az elvárásunk az lehet, hogy egy nagyobb mezőt alkossunk, és a virágok aránya nagyjából megegyező legyen az input képben látható arányokkal.
A következő lépésben feldaraboljuk ezt a képet, és kialakítjuk a cellákat. A feldarabolt cellákat szeretnénk megtekinteni és tetszőlegesen paraméterezni. Ehhez használjuk a Unity egyik felxibilis eszközét, a ScriptableObject-et. Ebben az osztályban tároljuk le a cellának a képét, a súlyát és a szomszédjait. Ebből az objektumból annyit kell automatikusan létrehoznia a beolvasónknak, ahány cellára daraboltuk fel az inputot. Miután megtötént a létrehozás, feltöltjük az objektumokat.




3.4. ábra. Üres ScriptableObject

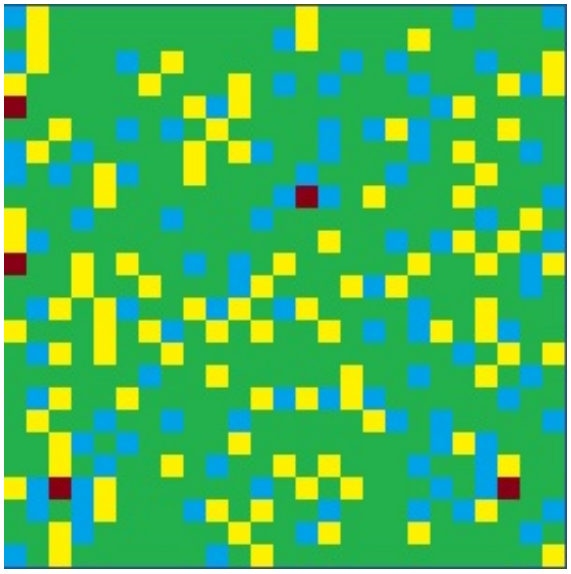



3.5. ábra. Feltöltött ScriptableObject

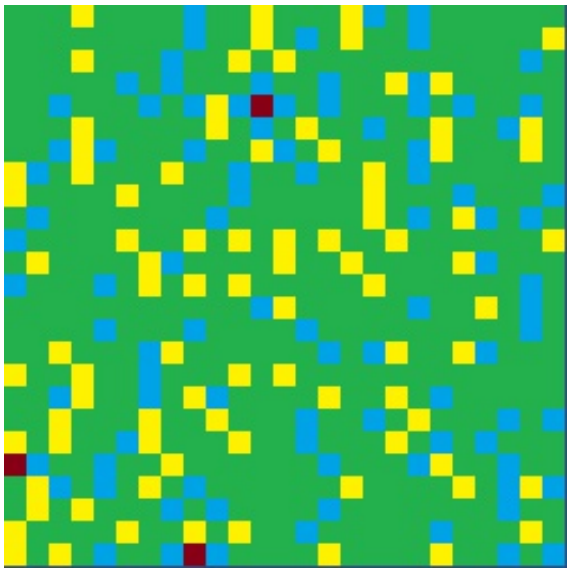
Cella-paraméter leíró objektumok

A feltöltött objektumokat pedig megadjuk az algoritmusnak, ezekből lesznek a cellák.

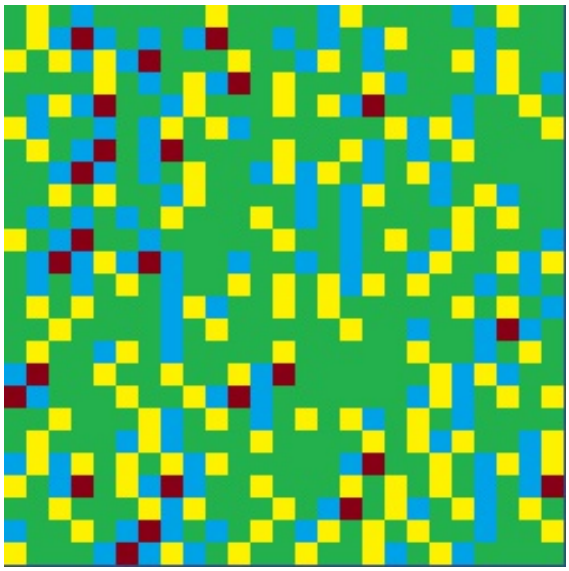
Az algoritmus futása előtt kiválasztjuk a preferált módszert a 4-es lépésben definiáltak közül, illetve hogy mekkora outputot szeretnénk generálni.
A példához egy $25X25$ nagyságú outputot választottam. A véletlenszerű generálás nem adott eredményt, mert csak elakadást produkált.



3.6. ábra. Horizontális generálás



3.7. ábra. Vertikális generálás



3.8. ábra. Entrópiás generálás

Output generálás különböző módszerekkel