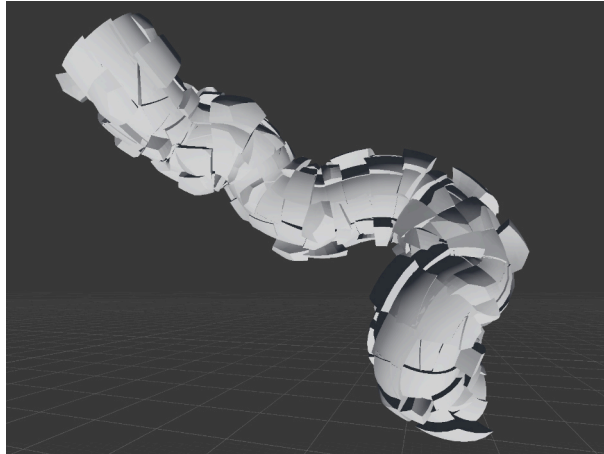
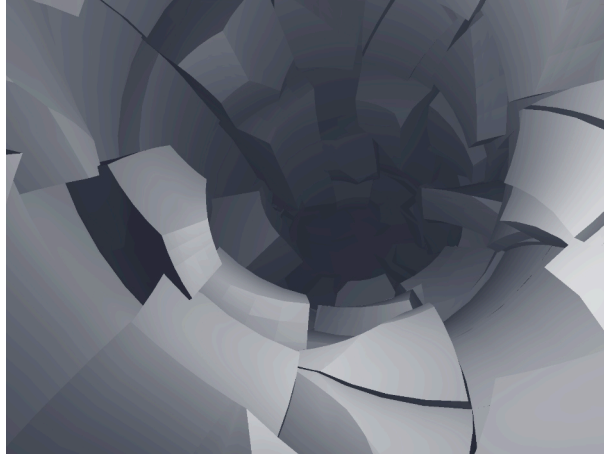


BARLANGJÁRATOK HÁROMDIMENZIÓS MODELLJEINEK PROCEDURÁLIS GENERÁLÁSA

KÉSZÍTETTE: SÁRDI CSABA
KONZULENS: DR. STROMMER LÁSZLÓ

BUDAPESTI MŰSZAKI ÉS GAZDASÁGTUDOMÁNYI EGYETEM
MORFOLÓGIA ÉS GEOMETRIAI MODELLEZÉS TANSZÉK
2024



ABSZTRAKT

A videojátékok világában egy felbecsülhetetlen fontosságú eszköz a „procedurális generálás”, amely lehetővé teszi közel végtelen világok létrehozását anélkül, hogy azt a fejlesztőnek pontról-pontra modellezve saját kezűleg kellene elkészítenie. Felszíni topográfiai elemek procedurális generálására, azaz paraméterezett függvények sorával való létrehozására már temérdek algoritmus áll rendelkezésünkre. Barlangok készítésére szolgáló hasonló módszerek azonban nemigen léteznek, valamint a néhány létező megoldás funkcionalitása is mind korlátozott. A célom egy olyan, barlangfelületek generálására képes algoritmus létrehozása volt, amely lehetővé teszi a művészi elképzelés érvényesülését és a változatosságot, mindeközben minimalizálja a befektetendő emberi munka mennyiségét. A végeredmény tehát egy olyan eszköz, amely képes – paraméterekkel szabályozható, azonban véletlenszerű vonalvezetéssel és belső topográfiával rendelkező – barlangmodellek létrehozására, ezzel új lehetőségeket nyitva meg a videojátékfejlesztés világában.

MI IS AZ A PROCEDURÁLIS GENERÁLÁS?

A legtöbb videojátéknak kulcsfontosságú eleme valamilyen világ, egy virtuális tér, amely a játékosok által átélt élmény egyik fő befolyásoló tényezője. Az ilyen terek létrehozása, azaz a tér megformálása oly módon, hogy az megfelelően valamilyen funkciónak pedig fundamentálisan építészeti feladat.

Ezeknek a virtuális tereknek a léptéke azonban gyakran meghaladja azt, ami lehetővé tenné az alapos műgonddal való pontról-pontra tervezést. Viszont egy tér létrehozása során ugyan sokszor egészen pontos formára van szükség a megfelelő funkció betöltéséhez – legyen az a játékosok mozgásának irányítása, vagy egy bizonyos érzelmi reakció kiváltása a szemlélőből – máskor elég egy bizonyos rendszer, vagy minta betartása. Ha elképzelünk egy sűrű fenyőerdőt, vagy akár egy mészkődarabokkal tarkított karsztmezőt, nem egy konkrét formára gondolunk, hanem egy szabályrendszerre, az egészet alkotó darabok viszonyára: hogy a nagyjából húsz méter magas fenyőfák hozzávetőlegesen tíz lépésnyire állnak egymástól, vagy, hogy a karsztmező lassan hullámozó göröngyeit térdmagasságú fű és helyenként fehér kőrákások véletlenszerű kompozíciója borítja. A fáradtságos kézi 3d-modellezés helyett tehát megalkothatunk egy függvényt, egy algoritmust, ami replikálva ezeket a természetes szabályokat, lehetővé teszi egyes fajta környezetek létrehozását végtelen különböző, egyedi formában. Ezt a folyamatot nevezzük *procedurális generálásnak*.

A procedurális generálás és a parametrikus modellezés nagyon hasonló és gyakran felcserélhető fogalmak. Azonban kevésbé tudományos, inkább művészeti felhasználási esetekben, ahol a folyamatok sok változót és gyakran véletlen-szám generálást tartalmaznak, a procedurális generálás a gyakoribb megnevezés a parametrikussággal szemben.

A procedurális generálásnak tehát fontos része a véletlenség. Akár komplex természeti folyamatokat, vagy kisebb tervezési döntéseket helyettesíthetünk randomizált értékekkel. Azonban fontos, hogy ilyenkor mindig tudatos döntést kell hoznunk azzal kapcsolatban, hogy mit bízunk a számítógépre és mit tartunk meg magunknak, mint tudatos tervezőnek. Például, ha egy homlokzatot tervezünk érdekes mozaik mintával, megfontolandó döntés, hogy az egyes csempék színét egy (akár véletlen elemeket is tartalmazó) algoritmusra bizzuk, hiszen gyorsan, olyan látványos eredményeket állíthatunk elő, amikre manuálisan nem lettünk volna képesek. Azonban ugyanennél a homlokzatnál az ablakkiosztás egy olyan feladat, amit érdemesebb élő, hús-vér tervezőre hagyni. Fontos megjegyezni még, hogy amennyiben valahol a procedurális megoldás mellett döntünk (például egy homlokzat mozaik-mintájánál) onnantól a végeredményt előállító algoritmus ugyanúgy tervezendő, mint, ahogy másképpen a végeredmény lenne; egy műgonddal elkészített minta tehát nem helyettesíthető véletlen kockadobások sorával.

Viszont gyakorlatban hogyan hozhatunk létre meggyőző véletlenséget egy számítógéppel?

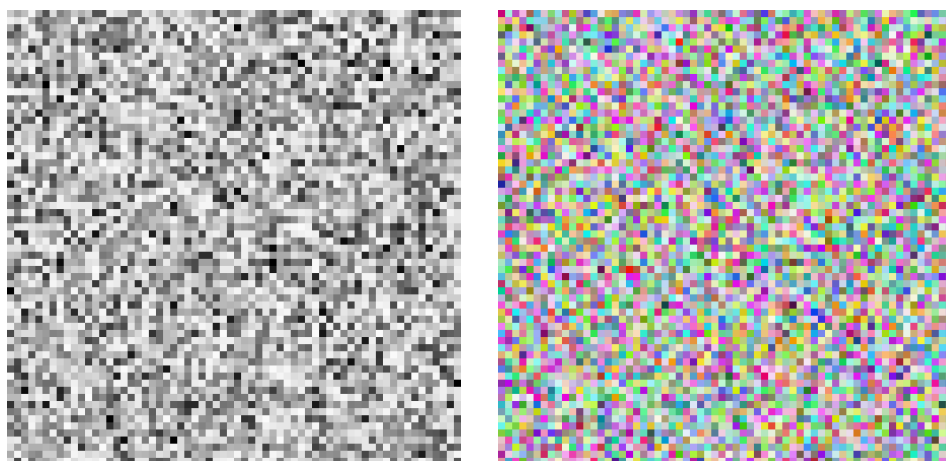
Ugyanez a probléma foglalkoztatta a korai kompjútergrafikával foglalkozókat a 80-as és 90-es években. Ugyanis amikor a valóvilágot valamiféle modell segítségével leképezzük (például egy 3D modellel*), a világban jelenlévő véletlenszerűség, zaj elveszik. És sokszor pontosan ez is a célunk (hogy a bonyolult valóságot egyszerűsített modellek segítségével értsük meg). Azonban, ha ezt a folyamatot visszafelé szeretnénk csinálni, azaz a semmiből létre szeretnénk hozni valamit, ami a lehető legjobban utánozza a valóságot, valahogy vissza kell adnunk ezt a zajt, tökéletlenséget, ami körbevesz minket a természetben. Tehát ha például a valóságtól megkülönböztethetetlen képi anyagot szeretnénk csinálni, például filmekhez, elengedhetetlen, hogy létrehozzunk egy módszert, amivel létrehozható ahhoz hasonló zaj, mint amit elvesztettünk a modell megalkotásakor.

* 3D modell alatt egy pontokból, éllekből és lapokból előállított testet, szaknyelven 3D mesh-t értek míg egy modell akármilyen absztrakt reprezentáció lehet, például egy statikai modell.

PSZEUDORANDOM-SZÁMGENERÁLÁS ÉS ZAJFÜGGVÉNYEK

Véletlen számok generálása számítógépen szinte mindig *pszeudorandom-szám-függvényekkel* történik. Egy véletlen szám létrehozására gondolhatunk úgy, mint egy dobásra egy kockával. Ezt az analógiát követve egy *pszeudorandom* függvény egy olyan kocka, aminek az eredménye például a pontos idő függvénye. Tehát két egymás utáni dobásom között nincs kapcsolat, azonban, ha két nap egymás után pontosan ugyanabban a másodpercben dobnék a kockával, garantáltan ugyanazt az eredményt kapnám. Tehát egy pszeudorandom függvény eredményeinek sora mindig megismételhető, azonban korábbi eredményekből nem tudok következtetni a következőkre.

Random értékek sorára gondolhatunk egyfajta jelként is, egy, kettő, vagy akár több dimenzióban, így beszélhetünk *zaj-függvényekről*. Például két dimenzióban az elképzelhető legegyszerűbb zajfüggvény, egy függvény, ami a tér minden pontjához egy egyedi, véletlen értéket rendel (például 0 és 1 között). Ezt nevezzük fehér zajnak (vicces nevén hangyafoci).



(1.1) (1.2) Fekete-fehér és színes (RGB) fehérzaj Blender segítségével létrehozva. (64x64 pixel)

Ahhoz, hogy egy zajfüggvényből képet alkossunk egy diszkrét mintavételezést kell elvégeznünk, hiszen végtelen felbontás nem jeleníthető meg. Ez okozza a hangyafociban a hangyák megjelenését, hiszen a mintavételezés adta limitált felbontás nélkül ezek a pontok, vagy hangyák végtelenül kicsik lennének.

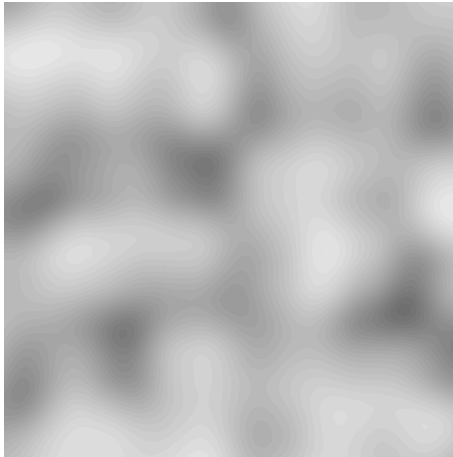
A probléma ezzel az egyszerű algoritmussal az, hogy hirtelen változásokat, ugrásokat tartalmaz, ami nem jellemző a természetre. A környezetben megfigyelhető zajok általában távolról nézve véletlenszerűen váltakozóak, azonban közelről vizsgálva egyenletes átmeneteket találunk bennük, nem pedig éles váltásokat. Ennek elkerülése érdekében egy dolog, amit tehetünk, hogy ritkásabban mintavételezzük a zajfüggvényt és a minták között valamilyen módon (lineáris, köbös, stb.) interpolálunk. Ezt nevezzük *gradient-noise*-nak.



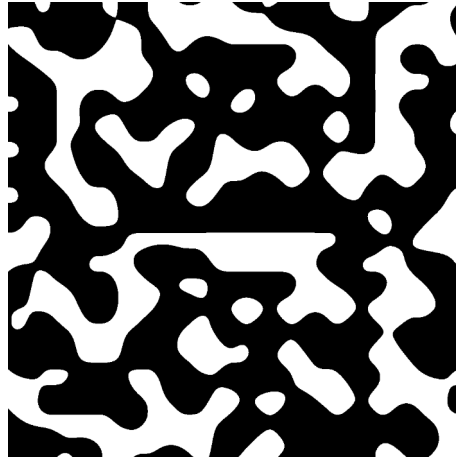
(1.3) Gradient noise Blender és Photoshop segítségével létrehozva.

PERLIN ZAJ

Azonban a manapság (kompjútergrafikai és procedurális generálási célokra) leggyakrabban alkalmazott zajfüggvény a Perlin zaj (és fejlesztett verziói, például *simplex noise*). Ezt az algoritmust egy Ken Perlin nevű amerikai informatikus fejlesztette ki 1983-ban, a természetben megjelenő véletlenszerű minták minél közelebbi utánzásának céljával.

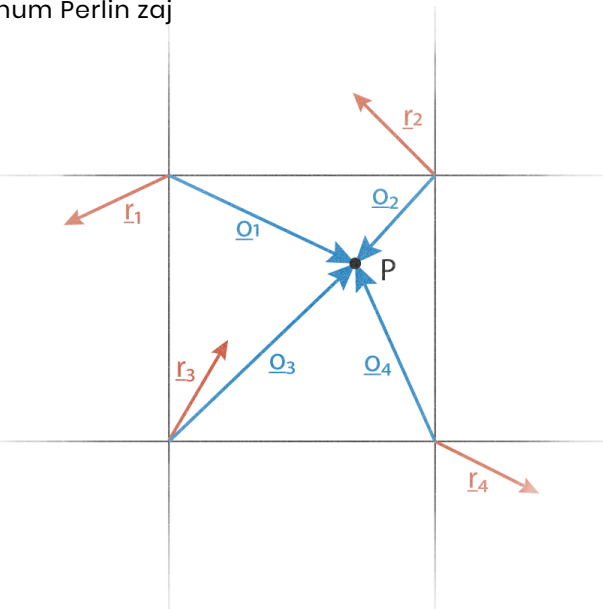


(1.4) Perlin zaj



(1.5) Signum Perlin zaj

A Perlin zaj értéke a sík egy adott pontján (P) az azt körbevevő rácspontokhoz (egész szám koordinátájú pontokhoz) tartozó random vektorok (r_1, r_2, r_3, r_4) és a rácspontokból P -be mutató *offset vektorok* skaláris szorzatának átlaga. (Az interpoláció a négy érték között általában nem lineárisan történik.) Ez azt is jelenti, hogy a Perlin zaj értéke minden rácspontban garantáltan nulla.

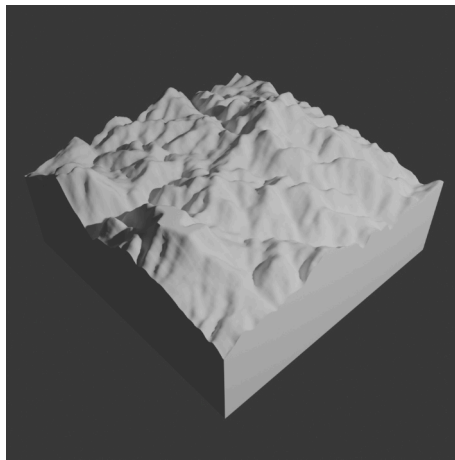
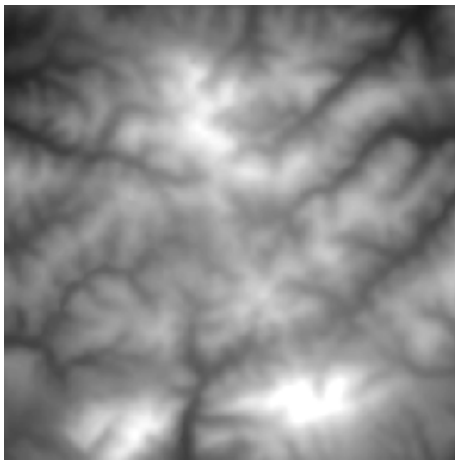
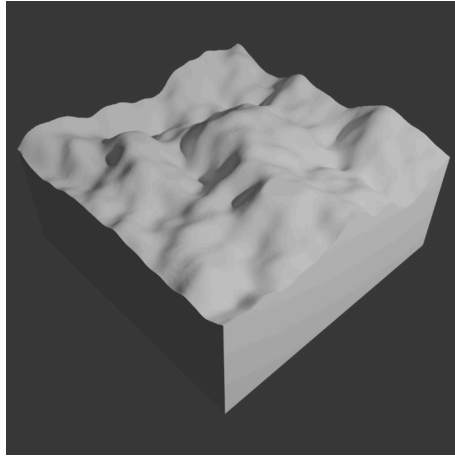
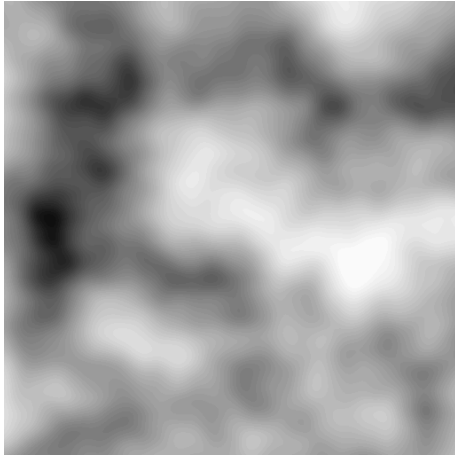


Az (1.4) ábrán demonstrált Perlin zaj mintából azonban még hiányzik egy nagyon fontos technika, ami elengedhetetlen a valóságos zajok létrehozásához: a Perlin zaj (gradient noise és más modern zajfüggvények pl.: voronoi) alkalmazható fraktál zajok létrehozására. A fraktál zaj azt jelenti, hogy több különböző frekvenciájú zajt súlyozottan összekombinálunk (ez történhet összeadással, szorzással, átlagolással); a folyamat eredményeképp egy olyan mintát kapunk, ami egyszerre tartalmaz nagyobb, valamint kisebb léptékű elemeket, hasonló módon a természetben megtalálható különböző formákhoz.



(1.6) (1.7) (1.8) 1, 2 és 3-oktávós (réteges) Perlin zaj

Az (1.6) ábrán megfigyelhető minták megtalálhatóak az (1.7) és (1.8) ábrákon is



(1.9) (1.10) Perlin zaj alapú fraktál zajjal generált topográfia
(1.11) (1.12) Valós magyarországi topográfia
(forrás: land.copernicus.eu)

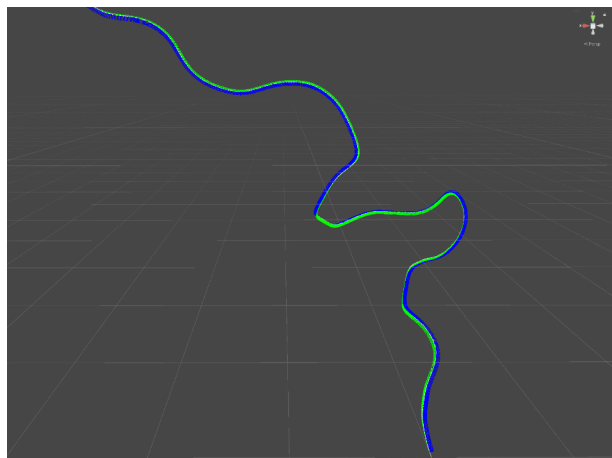
BARLANGJÁRATOK GENERÁLÁSA

Az én célom egy olyan algoritmus létrehozása volt, amely lehetővé teszi vizuálisan érdekes, azaz minél komplexebb formájú barlangok 3D modelljeinek generálását. Ugyanis míg felszíni topográfiai formák létrehozására számtalan ismert módszer és eszköz áll rendelkezésre – többek között a korábban tárgyalt technikák alkalmazásával – a barlangjáratok generálása korántsem ilyen alaposan feltárt terület annak ellenére, hogy véleményem szerint a barlangok és járatok földalatti világa legalább annyi potenciállal rendelkezik, mint a földfelszín videojáték-terek szempontjából.

A folyamat során egy barlangjáratot, mint egy hosszú, kígyózó vonalat modellezek, mely köré egymáshoz illeszkedő, belső topográfiai elemeket – mint például cseppköveket, sziklákat, vagy vízmosásokat – helyezek el. Ehhez tehát először is létrehozandó egy véletlenszerű formájú és tetszőleges hosszúságú térgörbe.

A térgörbe generálásához, először létrehozok egy háromdimenziós vektormezőt, mely a tér minden pontjához hozzárendel egy háromdimenziós egységvektort. Ez a vektor meghatározza a térgörbe irányát az adott pontban. Ezek után kijelölök egy tetszőleges kezdőpontot. Ha a kezdőpontban mintavételezem a vektormezőt, majd végtelen kicsi lépést teszek a kapott egységvektorral megegyező irányban, majd ezt a két műveletet ismétlem, míg a lépések összeadott hossza el nem ér egy előre meghatározott értéket, egy, a vektormező és a kezdőpont által meghatározott, egyedi formájú térgörbét kapok. A valóságban ezt a folyamatot azonban, nem praktikus analitikus módon végezni, hiszen a létrehozott térgörbe by design véletlenszerű formájú, így a numerikus módszerből adódó apró pontatlanságok elhanyagolhatók. Viszont ennek ellenére említésre méltó, hogy a numerikus és analitikus megközelítések által létrehozott görbék között lényeges eltérés, várható, hiszen a végesen kicsi lépések során keletkező pontatlanságok összeadódnak a folyamat során.

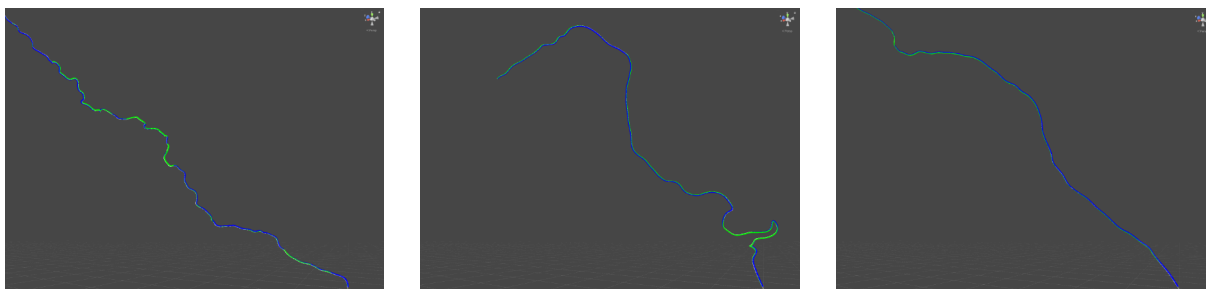
A vektormező értékeit három csatornányi (x,y,z) Perlin zaj kombinálásával, majd az eredmény normalizálásával hozom létre. Ehhez a folyamathoz persze bármilyen más zajfüggvény, vagy akár előre elkészített háromdimenziós textúrák* is használhatóak lennének.



(2.1) Egy, a leírt módszerrel létrehozott térgörbe

A leírt módszer alkalmazásával létrejövő forma egy példája a (2.1) ábrán látható. A kanyarok gyakorisága és görbülete – a görbe „érdeksége” – változtatható a generáláshoz használt Perlin zaj raszterméretének állításával. Finomhangolható továbbá a fraktál zaj rétegeinek (oktávjainak) száma és a különböző frekvenciájú rétegek súlya a kimeneti értékben.

* A számítógépteragrafiában egy textúra egy kettő, vagy ritkábban három dimenziós raszterben tárolt tetszőleges adatok halmazát jelenti. Például egy felület jellemzésekor leggyakrabban albedo (szín), roughness (fény szórási együttható), és normal (felület iránya) textúrákat használunk.



(2.2) (2.3) (2.4) Különböző paraméterekkel létrehozott térgörbék

A görbe létrehozásához használt paraméterek:

- Length : float – A görbe hossza
- Step : float – A mintavételezések közti távolság
- Offset : Vector3 – A kezdőpont koordinátái
- Layers : int – A Perlin zaj oktávjainak (rétegeinek) száma
- Falloff : float – A különböző oktávok súlyát vezérlő érték
- Noise Scale : float – Az első oktáv létrehozásához használt raszterméret (minden oktávé fele az előzőnek)
- Guidance : float – A vektormező értékeihez adott (1, 1, 1) irányú egységvektor szorzótényezője

Egy egyszerű csövet eköré a görbe köré szerkesztve már egy barlangféleséget kapunk, azonban a valódi barlangok természetesen nem rendelkeznek sima, geometrikusan szabályos falakkal. Ezen változtathatnánk oly módon, hogy a cső falainak pontjait valamilyen zajfüggvényekkel létrehozott érték alapján távolabb vagy közelebb toljuk a tengely görbéhez. Viszont annak ellenére, hogy ez a megoldás aránylag könnyen kivitelezhető és hatékony, én úgy vélem, hogy önmagában alkalmazva meglehetősen jellegtelen eredményekhez vezet. Ez persze nem jelenti, hogy az algoritmus egyáltalán ne lenne használható ezen formájában. Például a Minecraft feltehetőleg hasonló technikát használ a *spagetti* jellegű barlangjainak létrehozására^{*1}.

Én az említett cső érdes falakkal megoldás helyett, a cső belső falait kétdimenziós felületként kezelem, mely felületet különálló csempékre osztom. A csempéket pedig összeillő topográfiai elemekkel látom el (mint például cseppkövek, sziklák stb.) Mindennek elvégzéséhez szükséges egy, a görbéhez képest paraméterezett tér létrehozása. Ebben a térben egy pont a következőképpen adható meg:

- A görbe mentén megtett távolság. (d)
- Egy meghatározott nullszöghöz képesti, a görbe irányvektora körüli elfordulás. (a)
- A görbe d hossz által meghatározott pontjától vett távolság. (r)

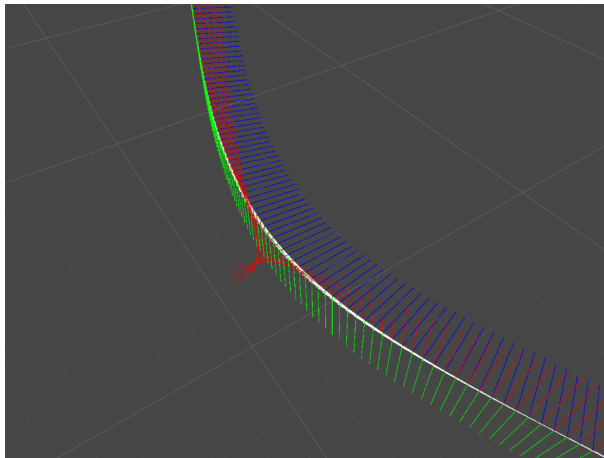
Így akármilyen x, y, z térben létrehozott 3D modell a görbe köré torzítható, ha annak a koordinátáit egy görbe által megadott d, a, r térben értelmezzük, majd visszaalakítjuk x, y, z térbe a megjelenítéshez.

A d, a, r -ból x, y, z térbe alakító függvény létrehozásának elengedhetetlen része a normálvektorok meghatározása a térgörbe pontjaihoz, ugyanis ezek iránya adja majd meg az a paraméter értelmezéséhez szükséges nullszöget. A térgörbe binormálisa és normálisa meghatározható a görbe irányvektorainak mintavételezésével, majd a minták vektoriális szorzatának kiszámításával (az irányvektor pedig megkapható a görbe egymást követő pontjainak különbségéből), azonban az így számított normálvektor a görbe inflexió pontjainál és egyenes szakaszainál nem használható, ugyanis a kapott vektor ilyenkor átkerül a görbe egyik oldaláról a másikra vagy hossza nullára csökken. (Ez a jelenség sajnos megfelelően simított^{*2} mintavételezés mellett is túlságosan nagy torzítót eredményez az egyes csempékben).

^{*1} A *noodle* és *cheese* fajtájú barlangok létrehozásához azonban egy másik, Perlin zajra alapuló módszert használnak a Minecraft fejlesztői. Ugyanez a megoldás megtalálható még például a No Man's Sky nevű játékban is.

^{*2} Simított, azaz több pontban mintavételezett adatok átlagát használó.

Normálvektor számítható még a görbe irányvektorának egy meghatározott iránnyal vett vektoriális szorzatából is, azonban sok esetben ez is hasonló problémákhoz vezet, mint a korábbi módszer, ámbár ritkábban.

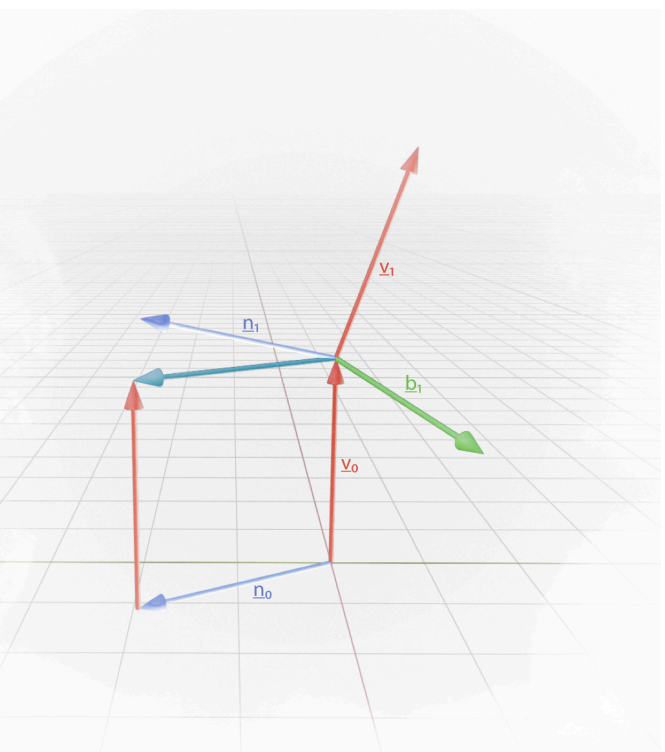


(2.5) Az ábrán látható az *adott iránnyal vett vektoriális szorzat* (piros) és *progresszív* (kék = normális, zöld = binormális) technikák eredménye közti különbség. Megfigyelhető az első technikából adódó jelentős torzió.

Megoldásképp az általam *progresszív*-nek nevezett normál meghatározási módszert fejlesztettem ki.

A *progresszív* normál meghatározási módszer az előzőkkel ellentétben nem mintavételezésen alapul, hanem a görbe pontjait tartalmazó tömbön* sorban végig haladva rendel normálvektort az egyes elemekhez. A sorrendben haladás szükséges ugyanis az eljárás a listában eggyel korábbi, már meghatározott normálvektort is használja az újabb elem kiszámításához. A normálvektorokat tartalmazó tömb n -edik elme tehát meghatározható a következő formulával:

$$\left| \begin{array}{ll} b_n = v_n \times n_{n-1} & // \text{binormális} \\ n_n = v_n \times b_n & // \text{normális} \end{array} \right.$$



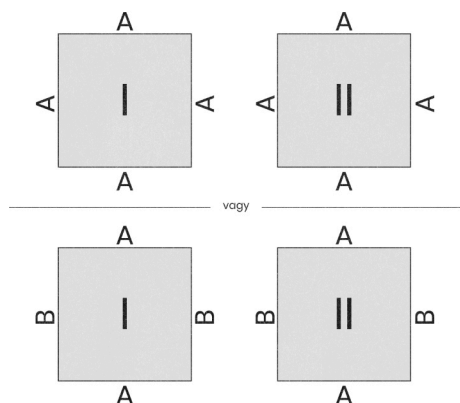
A tömb nulladik eleme egy tetszőlegesen felvett egységvektor a görbe $d=0$ pontjának irányvektora által meghatározott síkon.

A tömb értékeinek meghatározása után a görbe akármelyik pontjának normálvektora kiszámítható a tömb két legközelebbi értéke közti interpolációval.

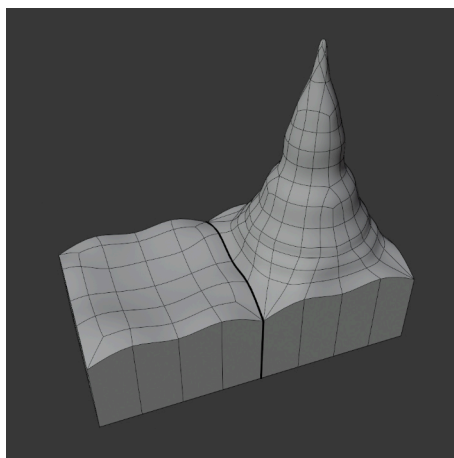
* A számítástechnikában tömbnek (angolul *array*) nevezzük azonos típusú (például lebegőpontos szám vagy vektor) adatok egy, meghatározott elemszámú rendezett sorát.

TOPOGRÁFIAI ELEMELK ELHELVEZÉSE

Az egyetlen hátralévő lépés tehát a cső felületének megtöltése különböző, előre elkészített 3D modelleket tartalmazó csempékkel. A legegyszerűbb esetben minden csempe összeilleszthető minden másikkal – ez persze nem jelenti azt, hogy a belső tartalmuk is azonos lenne, csupán, hogy külső éleik megegyezők. Például (3.1) és (3.2) ábra.



(3.1) I-es és II-es típusú csempék összeillő A és B típusú élekkel.



(3.2) Egyszerű példa két darab, négy azonos oldallal rendelkező csempére.

Ebben az egyszerű esetben a csempék kioszthatók teljesen véletlenszerűen, egy előre meghatározott mintában, vagy esetleg valamilyen zajfüggvény alapján. Például:

	Ha	$\text{perlin_zaj}(\text{pozíció}) < 0,2$	→	I-es csempe
	Ha	$\text{perlin_zaj}(\text{pozíció}) \geq 0,2 \ \& \ < 0,5$	→	II-es csempe

Az egyes csempék típusa meghatározható még a *Wave Function Collapse** (*WFC*)-nek nevezett algoritmus segítségével is. Ennek a megoldásnak különös előnye, hogy lehetővé teszi nem minden konfigurációban összeilleszthető csempék használatát. Így egyedibb, jelentősegteljesebb kapcsolatok jöhetnek létre – például egy szikla, amit gyakran apróbb kőtörmelék vesz körül, vagy egy eldőlt cseppkő, ami mindig két, egyedileg összeillő csempéből áll össze. A *WFC* algoritmus sokkal változatosabb, organikusabb, kevésbé gépies formák és elrendezések létrehozását teszi lehetővé a többi említett megoldáshoz viszonyítva.

* A *Wave Function Collapse (WFC)* Maxim Gumin 2016-ban publikált algoritmus, ami azonban szinte teljesen megegyezik Paul C. Merrell 2009-ben közzétett *Model Synthesis*-nek nevezett módszerével. Így a források közt mindekettőt feltüntettem.

WAVE FUNCTION COLLAPSE

Az általam készített *WFC* implementáció a következő modellt használja: Adott egy $x \times y$ cellás raszter, melynek méreteit a generálandó barlangjárat hossza és felbontása (azaz hány csempe tesz ki egy kerületet) határozza meg. A raszter minden cellája tartalmazza a következő adatokat:

- Azon csempe típusok listája, amelyek a jelen feltételek mellett kerülhetnek ebbe a cellába.
- A lehetséges csempetípusokhoz tartozó súly értékek, melyeknek összege mindig 1.
- Egy flag (igaz-hamis érték), ami jelzi, hogy ez a cella véglegesített csempét tartalmaz e.
- A véglegesített csempe amennyiben az előző flag igaz.

Minden csempéhez pedig a következő adatok tartoznak:

- Minden csempe mind a négy oldalához tartozik egy-egy lista, mely az adott oldallal kompatibilis csempe típusokat tartalmazza – tehát azon csempéket melyek az adott csempe mellé helyezhetők ebben az irányban –.
- Minden oldal minden csempe típusához tartozik egy súly érték, melyek összege oldalanként egy.

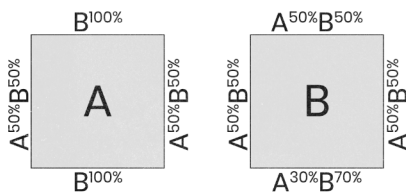
Súlyokkal ellátott csempék listájára az egyszerűség kedvéért mostantól, mint *csempe opció* fogok hivatkozni. Ezt a nevezéktant használva tehát egy raszter minden cellája tartalmaz egy csempét (amennyiben a cella véglegesített) vagy egy *csempe opció*t (amennyiben a cellába kerülő csempe még nem eldöntött). Ugyanígy egy csempe négy oldalára tekinthetünk négy *csempe opcióként*.

Kiszámítható két *csempe opció* uniója a két, csempéket tartalmazó lista egyedi elemeinek egyesítésével. Amennyiben egy csempe a két listában különböző súllyal szerepel, a két súly átlagát vesszük figyelembe. Az elemek egyesítését követően a súlyok újraszámítása (Azaz az egyes súlyok az új összeggel való elosztása) szükséges. Kiszámítható még két *csempe opció* metszete a két, csempéket tartalmazó lista közös elemeinek egyesítésével, majd a súlyok újraszámításával.

Két *csempe opció* akkor egyenlő, ha a csempéket tartalmazó listák elemei azonosak. A súlyokat ilyenkor figyelmen kívül hagyjuk.

A *WFC* algoritmus minden iterációjában kiválasztjuk a raszter legkisebb bizonytalansággal (azaz legnagyobb súly értéket tartalmazó *csempe opcióval*) rendelkező celláját és súlyozott random módon kiválasztjuk milyen véglegesített csempe töltsé ki. Ezt követően újraszámoljuk a rasztercellák *csempe opcióit* a frissen véglegesített cellától kifelé haladva. Ilyenkor egy cella *csempe opciói* a következő módon számíthatók: egy cella oldalának követelménye a cellához tartozó *csempe opció* csempéinek megfelelő oldalainak (minden oldal szintén egy *csempe opció*) uniója. Egy adott cella *csempe opciójának* értéke így a szomszédos cellák, határos oldali oldalkövetelményeinek metszete.

(3.3) A és B csempék. Az éleken jelölve a mellékük helyezhető csempék súlyozott listája, avagy *csempe opciója*



	B ^{100%}		
A ^{50%} B ^{50%}	A	A ^{50%} B ^{50%}	
	B ^{100%}		

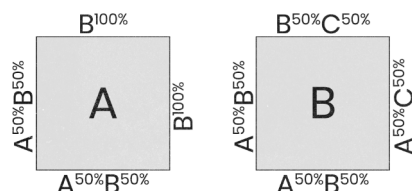
(3.4) A típusú csempe raszterbe helyezve. Feltüntetve az általa, a szomszédos cellákra kényszerített korlátozások.

AB	B ^{100%}	AB	
A ^{50%} B ^{50%}	A	A ^{50%} B ^{50%}	AB
AB	B ^{100%}	AB	
	AB		

(3.5) A szomszédos cellákban lévő csempe opciók további kényszert helyeznek az azokkal határos cellákra.

AB	B ^{100%}	AB	
A ^{50%} B ^{50%}	A	A ^{50%} B ^{50%}	AB
A ^{50%} B ^{50%}	B	A ^{50%} B ^{50%}	AB
AB	A ^{30%} B ^{70%}	AB	

(3.6) A legkisebb bizonytalansággal rendelkező cella végleges B csempét kap. Ezzel változik a környező cellákra gyakorolt kényszer.



(3.7) Az A csempe B csempe által meghatározza a tőle jobbra felfele eső cella (C) tartalmát is.

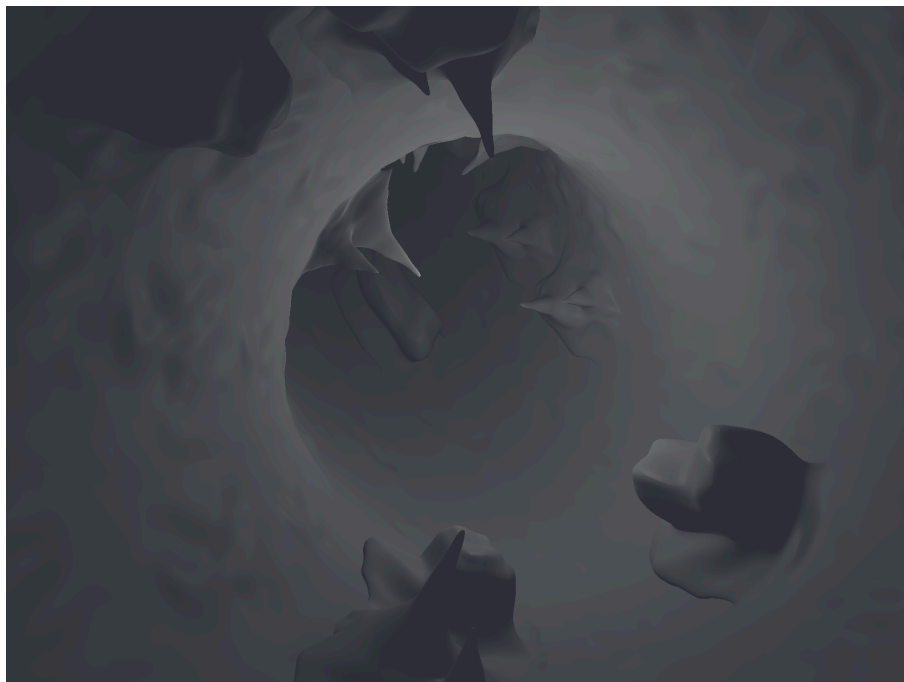
ABC	B ^{100%}	C	
A ^{50%} B ^{50%}	A	B ^{100%}	AC
AB	A ^{50%} B ^{50%}	ABC	
	AB		

(3.8) Megfigyelhető a (3.7) ábrán látható szabályok az A csempétől jobb, felfele eső cellára gyakorolt hatása.

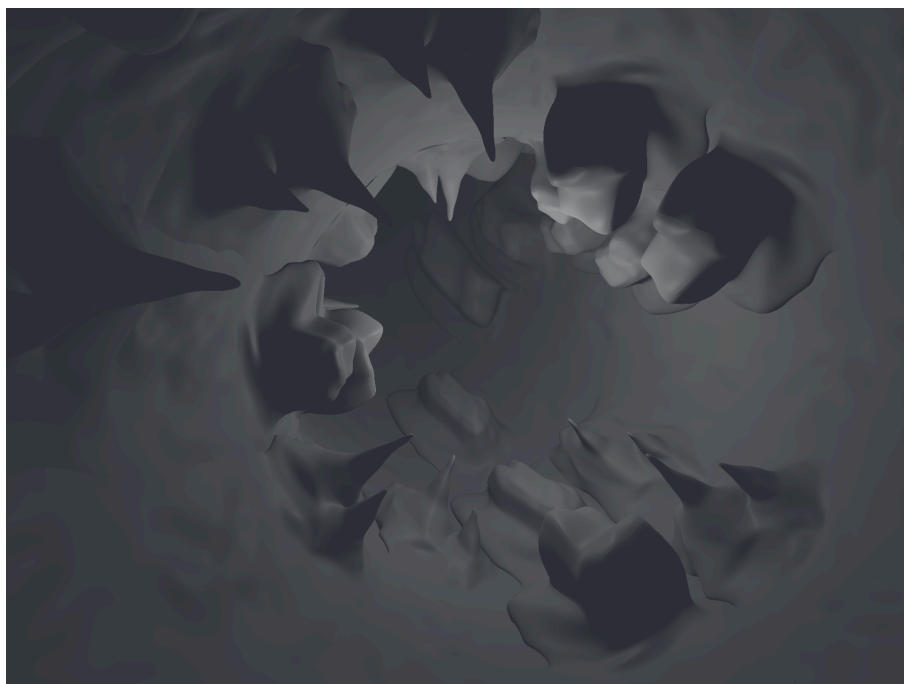
ÖSSZEFOGLALÁS

Úgy vélem, hogy két fundamentálisan különböző módszer egyesítésével sikerült valami egyedi, korábban még nem látott megoldást találnom. A bemutatott módszer néhány egyszerű csempekészletet használó példája megtekinthető a (4.1+) ábrákon.

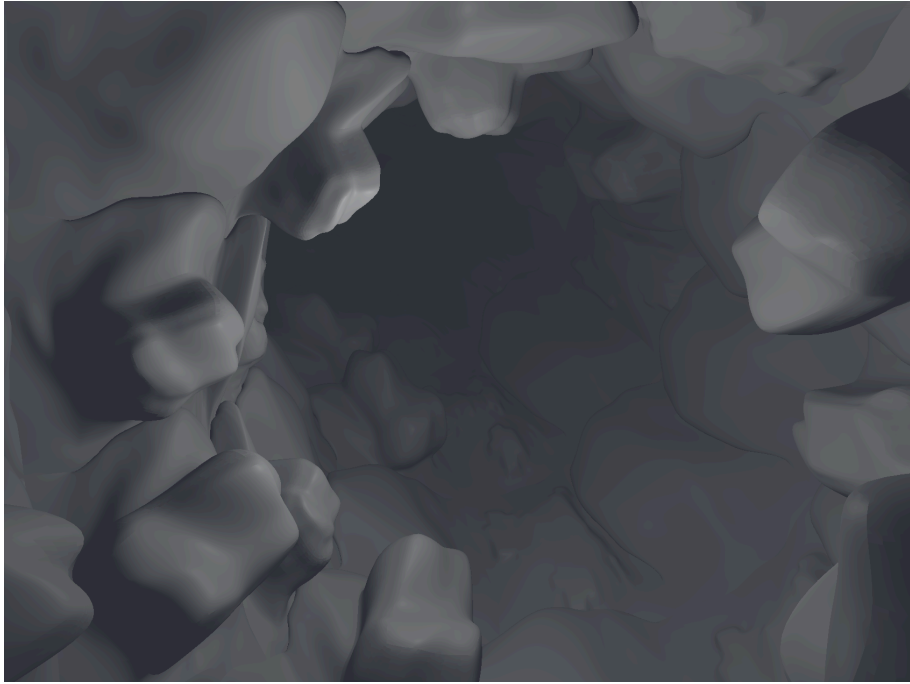
Jobb csempe készletek létrehozása és a WFC implementáció tökéletesítése és továbbfejlesztése jó következő lépések lehetnek. Hasonló módszerek alkalmazása bonyolultabb formák, például bővületek, csarnokok, elágazó barlangrendszerek létrehozására szintúgy fontos továbblépési irányok lehetnek.



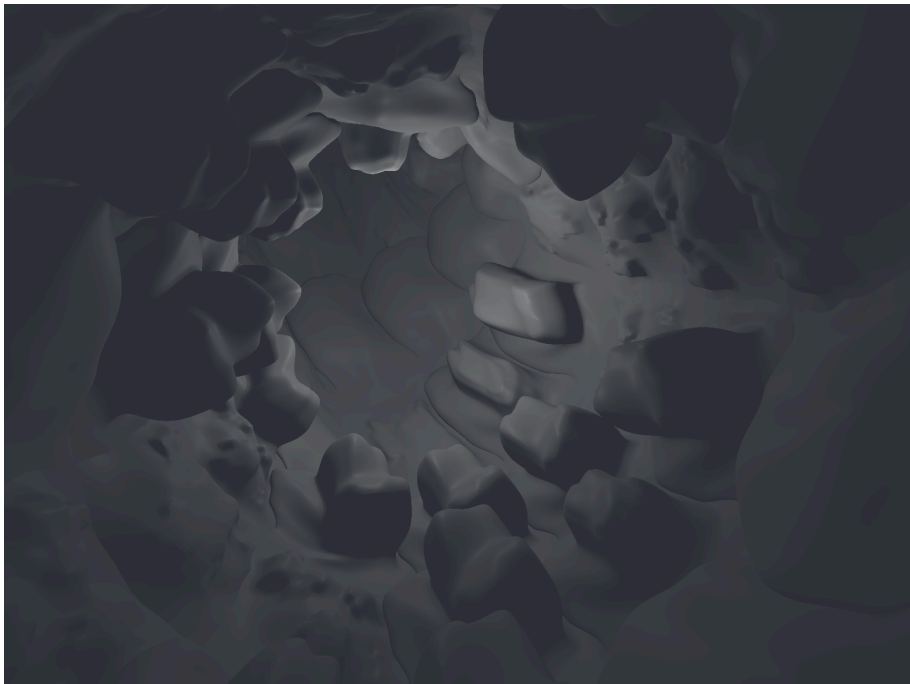
(4.1)



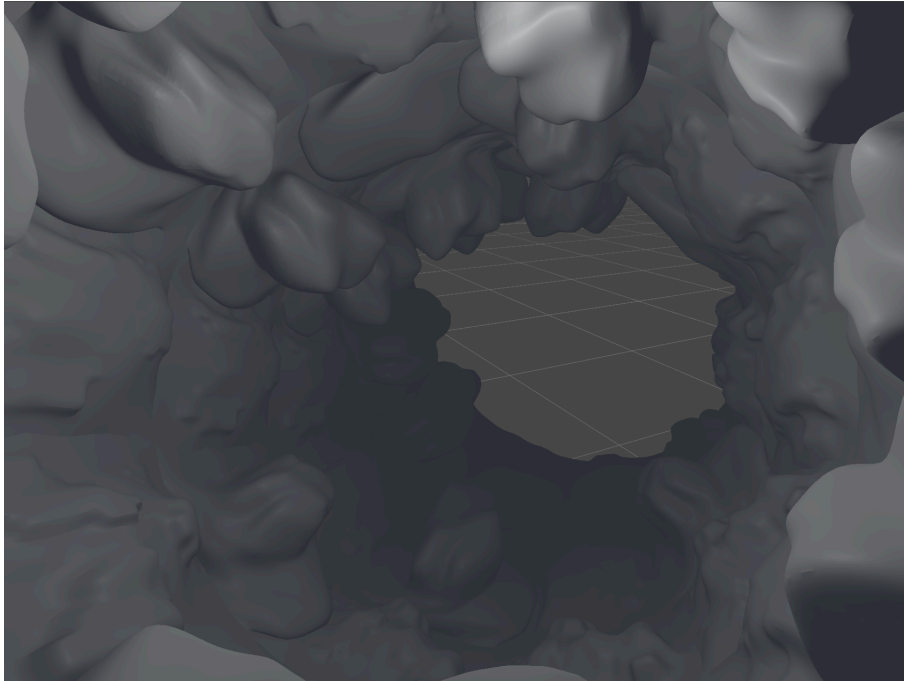
(4.2)



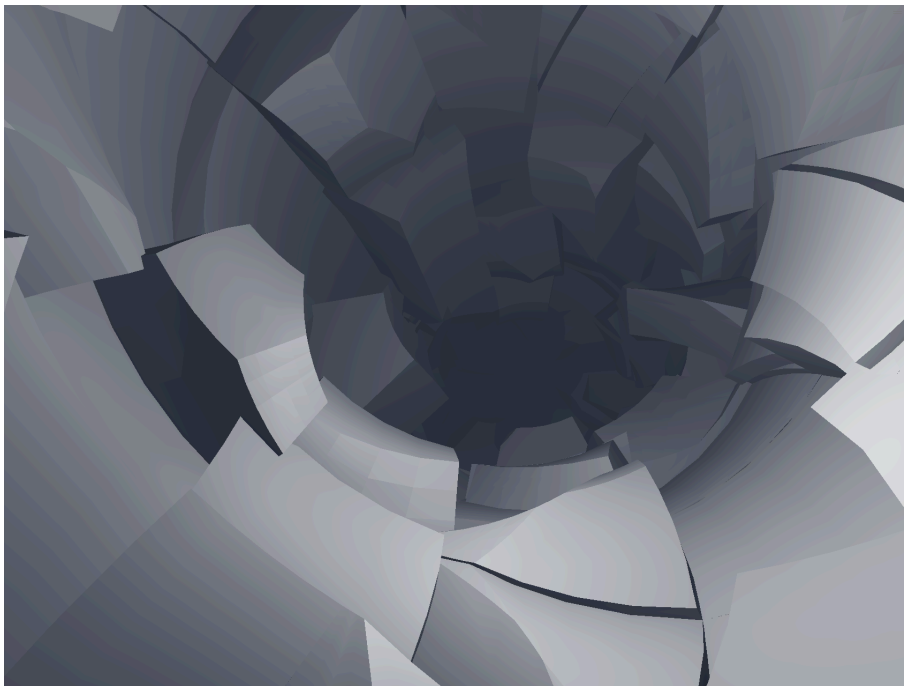
(4.3)



(4.4)



(4.5)



(4.6) Egy egyszerűsített, szögletes csempekészlettel generált barlang.

FORRÁSOK ÉS FELHASZNÁLT ESZKÖZÖK

Ken Perlin - An image synthesizer - 1985

<https://dl.acm.org/doi/10.1145/325165.325247> (u.m.: 2024.10.26)

Paul C. Merrell - Model Synthesis - 2009

<https://paulmerrell.org/wp-content/uploads/2021/06/thesis.pdf> (u.m.: 2024.10.31)

Maxim Gumin - Wave Function Collapse - 2016

<https://github.com/mxgmn/WaveFunctionCollapse?tab=readme-ov-file> (u.m.: 2024.10.31)

A leírt módszer szoftveres implementációja C# nyelven, a Unity motor felhasználásával készült.
Az illusztrációként használt zaj minták Blender regítségével készültek.