Studiu de Caz  
Cave Generation cu Cellular Automata

## **Cuprins**

[**Cuprins** 1](#_Toc155535308)

[Introducere 2](#_Toc155535309)

[**Prezentare generală** 2](#_Toc155535310)

[**Prezentarea categoriei** 2](#_Toc155535311)

[**Scopul studiului** 3](#_Toc155535312)

[Descrierea algoritmului 3](#_Toc155535313)

[**Particularități ale algoritmului nostru** 4](#_Toc155535314)

[Setul de date ales 5](#_Toc155535315)

[Implementarea Algoritmului 5](#_Toc155535316)

[Teste și conculzii 9](#_Toc155535317)

[Bibliografie 11](#_Toc155535318)

# Introducere

## **Prezentare generală**

Algoritmii de generare procedurală se referă la tehnici și metode utilizate pentru a crea conținut în mod automat, folosind un set de reguli și proceduri matematice. Acești algoritmi sunt adesea utilizați în domeniul graficii computerizate, jocuri video, design procedural, generare de terenuri, modele 3D și altele. Ideea principală este să se creeze conținut complex și variat prin intermediul unor reguli și proceduri, în loc să fie creat manual.

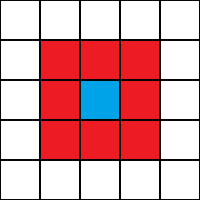
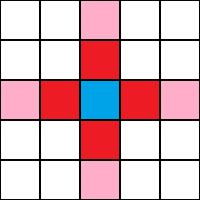
Utilizări ale algoritmilor de generare procedurală:

· **Generarea de terenuri**: În jocurile video, algoritmii de generare procedurală pot fi utilizați pentru a crea hărți și terenuri variate, oferind astfel o experiență mai dinamică și diversificată.

· **Generarea de orașe și peisaje urbanistice**: Pentru mediile urbane din jocuri sau simulări, algoritmii pot fi folosiți pentru a genera străzi, clădiri și peisaje urbane realiste.

## **Prezentarea categoriei**

O automată celulară e un model computațional studiat în teoria automatelor. Automatele celulare se mai numesc și spații celulare, structuri omogene/celulare etc. Acestea sunt folosite în fizică, biologie teoretică, și modelarea de microstructuri.

Automata celulară este în sine o matrice de celule, fiecare celulă avand un număr finit de stări (cum ar fi 0 sau 1 de exemplu). Matricea poate avea orice dimensiune atâta timp cât este finită. Pentru fiecare celulă exista un set de celule în vecinătatea ei. O stare inițială a fiecărei celule este stabilită la început (generația 0), stare care apoi se schimbă pe măsură avansează generațiile. Generatiile avansează după o regulă fixă (de obicei o funcție matematică) care determină starea fiecărei celule și a vecinătății ei. În mod normal, regula de schimbare a stării este aceeași pentru fiecare celulă, nu se modifică în timp și se aplică simultan fiecărei celule din întreaga matrice.

🡨ex. Tipuri de vecinătăți – se verifică vecinii stricți(sus, jos, stânga, dreapta), sau toți din zona 3x3 formată în jurul celulei

Automatele celulare se clasifica in 4 categorii:

* Automate care de obicei se stabilizează și devin omogene
* Automate în care modelele evoluează în structuri stabile sau oscilante
* Automate în care modelele evoluează haotic, structurile schimbându-se radical de la generație la generație
* Automate în care modelele evoluează în structuri extrem de complexe și care rezistă timp de multe generații

## **Scopul studiului**

Studiul algoritmului de cave generation cu cellular automata are ca scop principal să ofere o metodă eficientă și flexibilă pentru generarea procedurală a peșterilor și structurilor subterane în mediile virtuale. Acest algoritm se bazează pe principiul automatelor celulare pentru a simula procesul de formare a peșterilor într-un mod realist și diversificat. Prin înțelegerea și aplicarea acestui algoritm, dezvoltatorii de jocuri, designerii de medii virtuale și cercetătorii pot crea spații subterane complexe și interesante, oferind astfel utilizatorilor experiențe variate.

# 

# Descrierea algoritmului

Algoritmul este unul de generare procedurală și oferă un mod eficient de a crea peșteri cu o complexitate și diversitate semnificativă. Regulile specificate pentru actualizarea celulelor pot fi ajustate pentru a obține diferite tipuri de peșteri sau structuri subterane.

1. Se începe cu o grilă bidimensională (matrice) reprezentând spațiul subteran. Fiecare celulă din matrice poate fi considerată ca fiind o parte din zid sau spațiu liber al peșterii.
2. Se inițializează aleatoriu câteva celule ca spațiu liber, creând astfel spații deschise inițiale în peșteră. Aceste celule vor reprezenta începutul galeriilor sau camerei.
3. Se aplică reguli specifice pentru a actualiza starea fiecărei celule în funcție de starea celulelor vecine. Aici, un cellular automaton (automat celular) este folosit pentru a simula evoluția spațiului subteran. Celulele pot fi între una din două stări: spațiu liber sau zid.
4. Procesul de actualizare al celulelor este repetat într-un număr de iterații. La fiecare iterație, starea fiecărei celule este actualizată în funcție de regulile setate. Astfel, spațiile libere se extind, se formează camere și galerii, iar peștera își dezvoltă forma și structura.
5. După un număr adecvat de iterații sau când anumite condiții sunt îndeplinite (de exemplu, o anumită proporție de spații libere în raport cu spațiile ocupate de zid), procesul de generare este considerat încheiat. Rezultatul este o peșteră generată procedural cu o varietate de camere și pasaje.

## **Particularități ale algoritmului nostru**

Algoritmul se folosește de o matrice (de dimensiune n\*m) bordată cu celule zid care nu-și schimbă natura indiferent de iterație, pentru a se prezenta cât mai puține iregularități și a se asemăna cât mai mult cu o peșteră. Pentru fiecare celulă se verifică de ce natură e și apoi se caută într-un grid în jurul acesteia celule de tip ‘zid’.

În funcție de vecinii ei, celula curentă devine zid/spațiu.

De asemenea algoritmul nostru îmbină 2 seturi de reguli pt stabilirea tipului unei celule:

* În primele X iterații(preferabil 5), dacă în grid-ul de 3x3 care se stabilește în jurul celulei se regasesc cel puțin 5 ziduri **sau** dacă în grid-ul de 5x5 stabilit în jurul acesteia se afla mai puțin de 2 ziduri, natura acesteia va fi de tip ‘zid’, altfel de tip spațiu.
* În următoarele K iterații(preferabil 5 sau X/2 dacă numărul de iterații este foarte mare), se verifică într-un spațiu de 3x3:
  + dacă celula respectivă este tip ‘zid’ și are mai puțin de 5 vecini ‘zid’, atunci aceasta devine ‘spatiu’
  + dacă celula este de tip ‘spațiu’ și are 5 sau mai mulți vecini de tip ‘zid’ își schimbă natura în ‘zid’
  + dacă niciuna dintre aceste condiții nu sunt îndeplinite, celula rămâne la fel

Dacă ne legăm de un singur set de reguli, cavitățile sunt fie prea mari, fie prea mici, iar noi dorind să obținem o varietate cât mai mare de peșteri, am optat pentru varianta de a le combina.

În urma tuturor iterațiilor mai urmează o etapă de “curățare” (un flood fill), care elimină celulele/clusterele zid/spații izolate care nu pot exista în mod normal.

Avem 2 corner case-uri în funcție de nivelul de umplere și generearea inițială a celulelor: putem ajunge cu matricea plină de celule tip zid sau prea goală.

# Setul de date ales

Studiind generarea procedurală a peșterilor create folosind Cellular Automata, nu contează setul de date ales, pentru că dorim să obținem cât mai multă varietate în structurile create.

Algoritmul folosește 5 date de input, primite de la utilizator: Height si Width (dimensiunile matricei pe care se va lucra), Noise (între 0 si 100 care reprezintă factorul de zgomot - cât % aproximativ din matrice va fi umplută cu celule de tip ‘zid’), Generations (prin cate generatii va trece matricea) si Mode (programul să ruleze în debug mode - să vedem fiecare iterație ce produce, sau să afișeze doar structura finală).

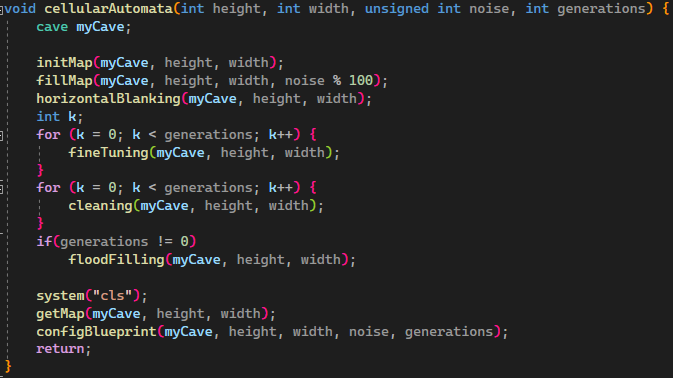
Cu toate acestea fiind spuse, pentru teste ne limităm la dimensiuni de 50x100, noise = 45 și generații = 5, pentru a păstra o anumită regularitate.

# Implementarea Algoritmului

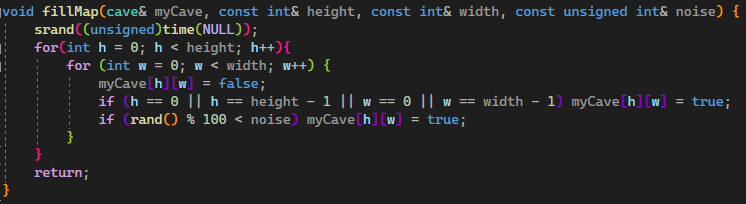
Limbajul de programare ales este C++, deoarece suntem foarte familiarizați cu el.

Functia main() nu este de interes, așa că aceasta nu va fi prezentată în detaliu. În main() doar se va face citirea datelor și se va apela programul de generare în sine.

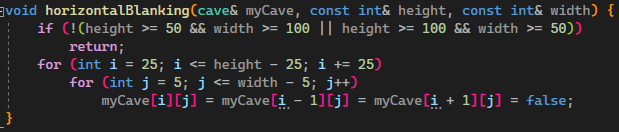
Functia cellularAutomata() este apelată în main și această constituie tot algoritmul în sine, și primește ca parametri toate datele de intrare precizate. Functiile initMap() si fillMap() au rol în configurarea matricei care va fi folosită.

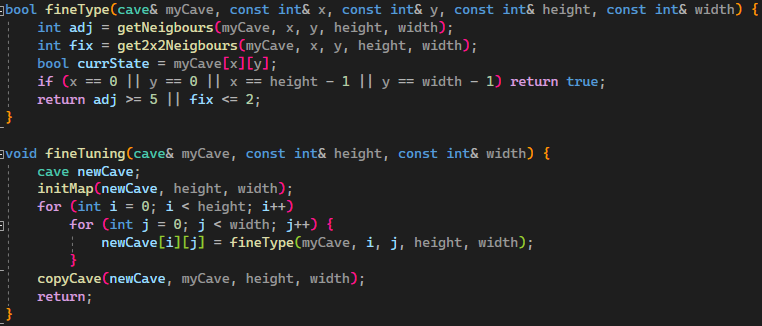
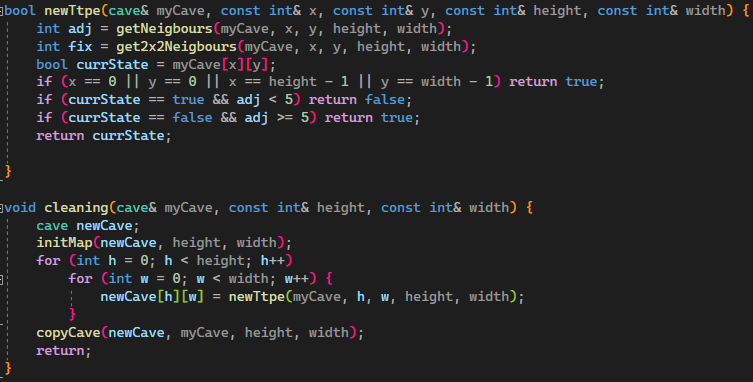


De interes dintre cele două este fillMap(), initMap() doar alocând memorie pentru matricea de tipul ‘cave’(std::vector<std::vector<bool>>); fillMap() constituie etapa de bază a algoritmului, umplând matricea random(random din C++ seed-uit cu time(NULL)) cu celule de tip true sau false, true semnificând un ‘zid’(și aflându-se în număr de aproximativ (width\*height)\*noise/100 la terminarea funcției) și false semnificând un ‘spațiu’. Pentru a evita ‘scurgeri’, marginile matricei vor fi mereu ziduri.



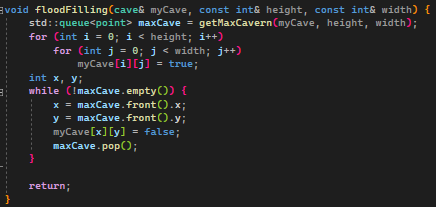
După umplerea matricei, tehnica Horizontal Blanking va fi aplicată pentru a ne asigura că vor exista structuri(spații) unite între ele în urma iterațiilor de generare prin care trece matricea. Tehnica constă în trasarea unei fâșii de spațiu orizontale de ~3 celule grosime în matricea noastră, după generarea inițială.

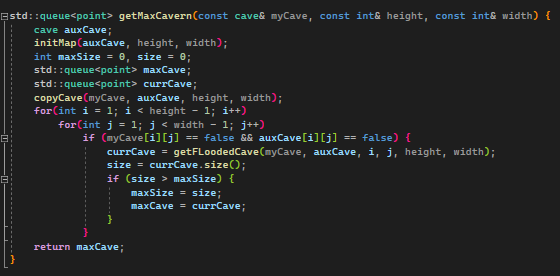


Ulterior, structura generată va trece prin 2 etape: prima o constituie trecerea din dezordine în ordine, oarecum modelând celulele formate după regulile propuse de cellular automata astfel încât să se formeze o structură care seamănă cu o peșteră. Funcția fineTuning() transformă celulele cu fiecare iterație, ajungându-se la tunele mici. Aceasta folosește prima regulă specificată pentru stabilirea naturii unei celule.  
  
.  A doua etapă face opusul, formând cu fiecare iterație spații tot mai ordonate. Fiecare iterație face fiecare celulă să asemene cu vecinii ei, astfel zonele cu multe ziduri se vor umple, iar spațiile se vor mări. Funcția pentru stabilirea naturii celulelor folosește a doua regulă menținută mai sus.

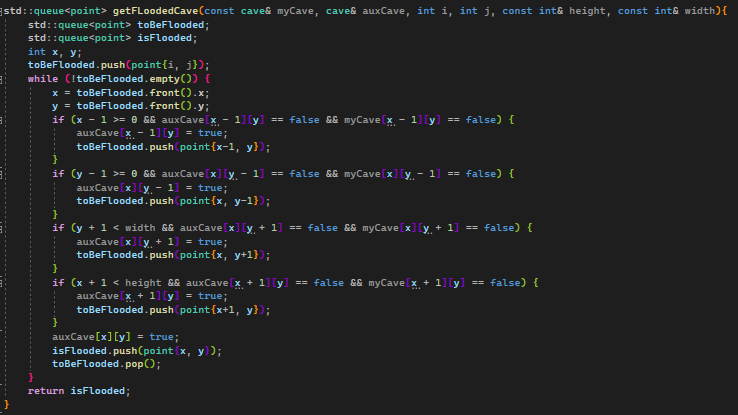
Un lucru care trebuie precizat, prelucrările nu se fac direct pe matricea noastră, rezultatele punându-se într-o matrice provizorie care va fi ulterior copiată în matricea inițială, pentru a nu ne încurca între date.

La finalul generării aplicăm un algoritm de “Flood Filling”, care presupune găsirea tuturor spațiilor izolate și umplerea tuturor în afară de cele mai mari.





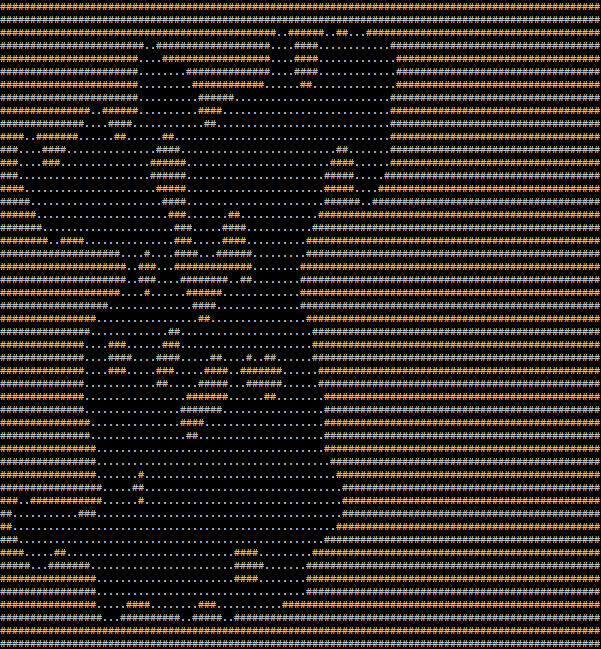
Algoritmul pune în matrice prima celulă de tip spațiu și o adaugă într-o structură de tip queue. Din acel moment, se găsesc toate celulele vecine care sunt și ele spații și se adaugă în queue. Odată ce nu se mai găsesc vecini, punctul este scos din queue și este adăugat într-un alt queue care conține punctele dintr-o zonă. În continuare se trece la următorul element din queue-ul inițial și aplică aceleași reguli, până când acesta se va goli, algoritmul de căutare și returnare a spațiilor izolate fiind un Breadth - First - Search (BFS). Toate punctele parcurse vor fi marcate în structura auxCave cu true. Queue-ul rămas va fi returnat, și algoritmul va căuta următorul punct de tip ‘spațiu’ din matrice care nu a fost încă parcurs.



# Teste și conculzii

Folosind generare procedurală, algoritmul nu va produce niciodată același rezultat, chiar și cu același set de date de intrare folosit de mai multe ori.

În următoarele 3 imagini se regăsesc generarea a 3 structuri folosind algoritmul nostru și setul de date de intrare 50 100 45 5:



Se observă cum se regăsesc spații largi și spații înguste, zidurile având un contur lin și cavitățile fiind de o varietate largă.

Din punct de vedere al eficienței, algoritmul este eficient, folosind matrici de bool în loc de int pentru a folosi cât mai puțină memorie. Timpul de execuție este stabilit de dimensiunile matricei cu care lucrăm și de câte ori vrem să o parcurgem, astfel având complexitate O(n^2).

În privința scalabilității, sistemul nostru suportă procesarea unui număr mare de date cu prețul unei perioade de execuție îndelungată. Acesta se comportă în aceeași manieră (păstrează uniformitatea și diversitatea rezultatelor fără defecțiuni) când cantitatea de date prelucrate devine foarte mare. Desigur, problemele legate de procesare se pot îndepărta prin adăugarea unor resurse adiționale hardware.

Vorbind despre adaptabilitate, algoritmul poate da rezultate de o natură diferită depinzând de ce dorește utilizatorul. În acest sens pot fi făcute modificări prin adăugarea sau ștergerea a unor reguli impuse la generare, aceste modificări fiind făcute cu ușurință. Dacă vrem ca structura rezultată să prezinte un sistem de tunele, folosim doar prima regulă. Dacă vrem peșteri largi, o folosim pe a doua. Dacă vrem tunele înguste și câteva zone mari, inversăm regulile între ele, aplicând-o pe a doua mai întâi, apoi pe prima. Când lucrăm cu matrici foarte mari, astfel încât rezultatele să fie scalabile și să nu se creeze un anumit pattern în urma generării, putem schimba regulile implicite din cellular automata, adăugând mai multe nivele de căutare la numărul de ziduri dintr-o zonă, mărind astfel zona de căutare. În plus, dacă se dorește ca structura generată sa reprezinte o peșteră dar în 3D, trebuie schimbate de exemplu câte celule sunt verificate pentru stabilirea naturei unei celule la fiecare iterație(din 9 in 27 de exemplu), și adăugarea unei interfațe care poate afișa structura generată în 3D(ex. renderer), astfel algoritmul fiind ușor de modificat.

În următoarele imagini se pot observa structuri cu tunele mici, spații mari și respectiv înguste, unde codul a fost modificat.



Printre dificultățile întâmpinate se regăsește posibilitatea ca rezultatul să nu fie unul dorit. Acesta poate să însemne prea multe celule izolate ce dau impreasia unor ‘deșeuri’, spații izolate care în număr mare pot să strice forma de peșteră pe care vrem să o obținem. Aceste anomalii sunt în schimb rezolvate prin prezența metodelor de horizontal blanking și flood filling.

Un caz special este atunci când noise-ul este prea mare. În urma generării, din cauza regulilor impuse de cellular automata, toate celulele (pe lângă cele formate de horizontal blanking) vor deveni ziduri, astfel fiind imposibil să se poată forma structuri care arată a peșteri. Acest caz special poate fi rezolvat prin schimbarea regulilor, de exemplu spunându-i algortimului să transfome o celulă de tip zid în spațiu dacă are prea mulți vecini de tip zid într-o zonă foarte mare(de ex. 9x9 sau 11x11).

# 

# Bibliografie

→ Rogue Basin - Cellular automata method for generating random cave-like levels

<https://roguebasin.com/index.php/Cellular_Automata_Method_for_Generating_Random_Cave-Like_Levels>

→ White Box Dev on Youtube - Cellular Automata | Procedural Generation | Game Development Tutorial

<https://www.youtube.com/watch?v=slTEz6555Ts>

→ Acerola on Youtube - Cellular Automata: Complexity From Simplicity

<https://www.youtube.com/watch?v=t_HcBAO_Yas>

🡪Massachusetts Institute of Technology(MIT) – Procedural generation

<https://www.mit.edu/~jessicav/6.S198/Blog_Post/ProceduralGeneration.html>

🡪Wikipedia – Procedural generation

<https://en.wikipedia.org/wiki/Procedural_generation>

🡪Barney Codes on Youtube – Cave Generation on P5.js

<https://www.youtube.com/watch?v=FSNUp_8Xvqo>

→ Wikipedia - Cellular automaton

<https://en.wikipedia.org/wiki/Cellular_automaton>

→ Math n’ programming :

<https://jeremykun.com/2012/07/29/the-cellular-automaton-method-for-cave-generation/>

→Envato Tuts+:

<https://code.tutsplus.com/generate-random-cave-levels-using-cellular-automata--gamedev-9664t>