UNIVERSITATEA DIN BUCUREŞTI
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ

LUCRARE DE LICENŢĂ

COORDONATOR ŞTIINŢIFIC
Lect. Dr. Banu Demergian Iulia

STUDENT
Pătru Sorin-Cătălin

BUCUREŞTI
IUNIE 2018

UNIVERSITATEA DIN BUCUREŞTI
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ

LUCRARE DE LICENŢĂ

Generare procedurală de conținut

COORDONATOR ŞTIINŢIFIC
Lect. Dr. Banu Demergian Iulia

STUDENT
Pătru Sorin-Cătălin

BUCUREŞTI
IUNIE 2018

Cuprins

1. INTRODUCERE

Dezvoltarea rapidă a calculatoarelor personale din punct de vedere al performanţei cât și scăderea drastică a preţurilor acestora a dus în mod involuntar la o problemă nouă pentru programatorii din ziua de azi și anume necesitatea consumatorilor de software de produse din ce în ce mai complexe, cu din ce în ce mai mult conținut.

Un domeniu în care s-a observat într-un mod mai evident această problemă este cel al jocurilor pe calculator și al telefoanelor mobile, unde jucătorii au așteptări din ce în ce mai mari pentru jocurile pe care le joacă, dorind din ce în ce mai mult conținut, universuri de joc mai mari și mai complexe, cât și jocuri cu durată mai mare.

În prezent există din ce în ce mai mulți oameni care joaca jocuri pe calculator sau pe mobil. Conform raportului realizat de US Entertainment Software Association (ESA) în 2010, 68% din familiile din America joaca jocuri pe calculator, vârsta medie a unui jucător fiind de 35 de ani.

Astfel, datorită acestei dorințe a consumatorilor pentru jocuri cu mai mult conținut, dezvoltatorii de jocuri au ajuns sa aibă costuri din ce în ce mai mari, necesitând să angajeze mai mulți oameni pentru a realiza un joc.

În momentul de faţă nu există o soluție bună pentru această problemă, dar un domeniu care a început sa arate rezultate promițătoare pentru rezolvarea acestei probleme este generearea procedurală de date.

În ultimii ani s-a putut observa o creștere a cantității literaturii de specialitate, unde cercetătorii din diverse domenii academice au adus un aport propriu la problemele existene în generarea de conținut pentru un joc. Acest influx de idei noi a dus la crearea de noi metode de generare procedurală, precum și reinterpretarea și combinarea a unor metode deja existente, unele dintre aceste metode încă având nevoie de mai multă dezvoltare până când vor putea fi aplicate în practică.

Putem considera algoritmii de generare procedurală ca fiind soluții pentru probleme de generare de conținut. O astfel de problema poate fi reprezentată spre exemplu de generarea de plante, cu un nivel scăzut de detaliu , care arată realistic într-un interval de timp sub 50 de milisecunde, pentru a putea fi plasate în fundal. O altă astfel de problemă ar putea fi generarea unei idei unice pentru un joc pe parcursul a câtorva zile de computare, sau finisare obiectelor dintr-un joc de către un thread în background în timp ce un designer le editează. Propietațiile dorite sau necesare a unei astfel de soluții ajung astfel să difere foarte mult în funcție de problema ce dorim să o rezolvăm. Astfel singurul lucru constant ajunge să fie faptul că vom avea mereu compromisuri de făcut între timpul de execuție și calitate, sau între diversitatea obiectelor generate și fiabilitatea lor.

Cele mai comune proprietăți dorite într-o soluție de generare procedurală sunt:

- **Viteză:** cerințele de viteză variază foarte mult în funcție de momentul în care se așteaptă să fie generat conținutul, în timpul execuției jocului sau în timpul perioadei de creare a acestuia, putând varia între milisecude până la ore.

- **Fiabilitate:** unele generatoare crează conținut fără a realiza verificări ale calității, iar altele garantează un anumit nivel de consistență în generarile lor. Acest lucru este mai important pentru

anumite tipuri de conținut, spre exemplu generarea unei case fără o ușa de intrare, lucru care ar fi dezastruos pentru joc, dar există și anuminte cazuri cum ar fi generarea unei plante, unde singura consecință a unei generări suboptime este o plantă care arată puțin ciudat dar nu afectează jocul per total.

 Nivelul de control: adesea este nevoie ca soluțiile de generare procedurală să poată fi controlate într-o anumită măsură, astfel încât un utilizator sau un algoritm (în cazul jocurilor care se adaptează în funcție de jucător) să poată specifica un anumit aspect al conținutului generat.

 Expresivitate și Diversitate: adesea este nevoie să generam un set divers de conținut pentru a evita situația în care toate obiectele dintr-un joc arată identic, ele fiind doar variații ale aceluiași obiect cu modificări minore. Un exemplu marginal ar fi un generator de nivele care generează mereu un nivel care arată identic cu excepția unui copac care are numărul de ramuri schimbat, un alt exemplu marginal din celalalt capăt al spectrului ar fi un generator de nivele care generează nivele absolut aleatorii fără nici o regulă. Astfel crearea de generatoare expresive și care crează conținut foarte divers este un lucru foarte important

 Creativitatea și Credibilitatea: în majoritatea cazurilor conținutul generat este de dorit să arate ca și cum ar fi fost generat de un om și nu de un algoritm.

Datorită faptului că domeniul generării procedurare de date poate fi considerat încă la fazele inițiale ale dezvoltării, acesta conține metode variate și nu este foarte bine definit. El poate fi aplicat la diverse domenii dar în această lucrare ne vom axa asupra problemei generării datelor pentru jocuri.

Lipsa de utilizare a metodelor de generare a conținutului procedural are cel mai probabil legătură cu lipsa de control, designerii fiind suspicioși de rezultatele ce vor fi generate de către algoritm, dar o serie de beneficii apărute recet pot schimba acest lucru.

Printre aceste beneficii se enumeră:

- Generarea rapidă de conținut ce satisface nevoile unui designer
- Diversitatea ce poate fi obținută prin ultilizarea acestor algoritmi cu schimbări mici ale codului inițial (lucru ce poate fi observat și în această lucrare în secțiunea de L-systems)
- Timpul și banii pe care o companie poate să îi economisească în procesul de dezvoltare
- Conținutul se poate adapta automat la fiecare jucător.

Generarea procedurală de conținut poate fi definită (în contextul nostru limitat) ca find crearea prin intermediul unui algoritm, ce primește un imput limitat sau indirect din partea unui utilizator, de conținut pentru un joc. Cu alte cuvinte generarea procedurală de conținut se referă la un program ce poate crea conținut pentru un joc independent sau împreună cu unu sau mai mulți jucători sau designeri.

Un termen cheie în această definiție este reprezentat de cuvântul “conținut” ce este interpretat în contextual nostru că fiind: nivele, hărți, reguli ale jocului, texturi, povești din interiorul jocului, muzică, vehicule, personaje, etc.

În domeniul jocurilor pe calculator, generarea procedurală de date poate fi folosită pe mai multe niveluri ale dezvoltării jocului:

1. **Elemente ale jocului**
2. **Spaţiul jocului**
3. **Sistemele jocului**
4. **Scenariul jocului**

1. **Elemente ale jocului**

Cea mai elementară și puțin abstractizată metodă de a folosi generarea pocedurală de conținut este să o folosim pentru generarea de conținut pentru joc, cum ar fi plante, animale, sunete și texturi. Un exemplu de algoritm pentru realizarea procedurală de plante va fi prezentată mai târziu în această lucrare de licenţă.

2. **Spaţiul jocului**

Spaţiul jocului se referă la universul în care are loc jocul, acesta este probabil unul din locurile unde generarea procedurală de date are şansele cele mai mari să înlocuiască programatorii, existând în momentul de faţă modalităţi care pot genera hărţi realistice[6] , orașe[5] sau chiar dungenuri.

3. **Sistemele jocului**

Sistemele jocului reprezintă felul în care anumite elemente din joc interacţionează întere ele, spre exemplu modul în care flora și fauna interacţionează, sau modul în care mai multe orașe din universul jocului interacţionează înte ele prin negoţ. Crearea de astfel de sisteme poate duce la realizarea unui univers mai realist și mult mai dinamic.[7]

4. **Scenariul jocului**

O alta porţiune din joc care poate fi lăsată parţial în grija calculatorului sunt poveşti în interiorul jocului. Mai precis este posibil să generăm în mod dinamic poveşti în funcție de alte date din universul jocului (dacă nu exista o resursă într- un oraş din joc, se poate emite automat o aventură pentru jucător

și se pot realiza câteva obstacole pentru acea aventură) sau chiar realizarea de povești principale ale jocului, dezvoltatorul realizând doar o schiță a ordinii în care trebuie să se întâmple evenimentele, iar calculatorul generează procedural evenimentele ce se întâmplă între punctele cheie stabilite de dezvoltator.[8]

Folosind metode de generare procedurală se poate crea în mod uşor un univers pentru un joc putând crea harta jocului[6] populând harta cu un algoritm ce folosește L-systems[5] și în final realizând dungenuri în care să se petreacă confruntări între jucător și inamicii săi din joc. Problema principală cu generarea procedurală a unui astfel de univers rămâne totuși în final faptul că universul creat ar putea să fie plictisitor din punctul de vedere al jucătorului și este necesar să se ajungă la un compromis între porțiunile generate de calculator și experiențele realizate manual de dezvoltator.

Un exemplu clar de joc care a folosit excesiv generarea procedurală rezultând un joc plictisitor este "No man's sky" care a rezultat în realizarea unui joc care a pornit o controversă imensă ducând la nevoia companiei care a realizat jocul, să returneze banii jucătorilor care au cumparat jocul.

În această lucrare vom aborda mai multe metode de generare procedurală de date la nivelul de elemente ale jocului (L-systems) și la nivel de spațiu al jocului (BSP) și vom explora avantajele și limitările generării procedurale de conținut în contextul jocurilor pe calculator.

2. BINARY SPACE PARTITIONING

2.1 SCURTĂ PREZENTARE

BSP (Binary Space Partitioning) este o metodă generică pentru a împărți un spațiu în mulțimi convexe, alegând recursiv un hiperplan .

O metodă de a implementa BSP este prin intermediul BSP Tree (Binary Space Partitioning Tree) care este un arbore binar, în cadrul căruia rădăcina reprezintă spațiul total, iar frunzele arborelui reprezintă subdiviziunile spațiului obținute la finalul partiționării.

BSP se poate folosi într-o varietate de aplicații, spre exemplu generarea de umbre și pentru a putea implementa surse multiple de lumină în jocuri.

Pentru generarea de umbre se poate folosi Shadow Volume BSP Tree (SVBSP) propus de Chin and Feiner[11] acest algoritm constă din doi pași:

- Determinarea umbrelor:
- Mărirea arborelui SVBSP

Acest algoritm poate de asemenea să fie extins pentru a putea rezolva și problema generarii de surse multiple de lumina.

2.2.ISTORIE

BSP a fost creat inițial în contextul graficii 3D **[1][2]** unde structura de arbore BSP prezenta date legate de spațiul unui joc într-un mod convenabil, cum ar fi proximitatea încăperilor unui joc din perspectiva jucătorului, arborele putând să ofere încăperile de la cea mai apropiată la cea mai depărtată.

Această metodă a fost creată din necesitatea graficii pe calculator de a desena în mod rapid scene 3D realizate cu poligoane. Un mod simplu de a desena astfel de scene este prin "Painter's algorithm".

Painter's algorithm este un algoritm the priority fill. Numele acestui algoritm provine de la o tehnică din pictură în care obiectele sunt pictate de la cel mai depărtat obiect la cel mai apropiat obiect, această metodă este folosită pentru a asigura că se colorează peste porțiunile care nu sunt vizibile.

2.3.APLICAȚII

Arborii BSP sunt adesea folosiți în jocuri 3D, de cele mai multe ori în FPS-uri (first person shooters) cum ar fi Doom, Quake și urmașii acestor jocuri. Aceștiia fiind foarte buni pentru eliminarea rapidă a suprafețelor ascunse în scene statice și pentru computarea de umbre cu una sau mai multe surse de lumină statice.[3]

Deși arborii BSP prezintă un mod ușor de a genera subpartiții această metodă prezintă o problemă din punct de vedere al timpului de execuție (fapt ce poate fi observat prin creșterea exponențială a subparțiilor 2,4,8,16), lucru ce duce la utilizarea acestor arbori în preprocescare și imposibilitatea de utilizare a acestora în timp real.

2.4.GENERARE

Modul standard de utilizare a BSP este randarea de poligoane cu "Painter's algorithm". Fiecărui poligon i se asociază o "față" și un "spate" care sunt alese arbitrar și afectează numai structura arborelui. Arborele este construit dintr-o listă neordonată a tuturor poligoanelor. [1]

Algoritmul pentru construirea în mod recursiv a arborelui din această lista neordonata este:

1. Alege un poligon P din listă
2. Construiește un nod N în arborele BSP și adaugă P la lista poligoanelor de la acel nod
3. Pentru orice alt poligon din listă
    - Dacă poligonul este în totalitate în fața planului care îl conține pe P, mută acel poligon la lista nodurilor din fața lui P.
    - Dacă poligonul este în totalitate în spatele planului care îl conține pe P, mută poligonul la lista poligonelor din spatele lui P.
    - Dacă poligonul se intersectează cu planul ce îl conține pe P, împarte poligonul în două poligoane și mută poligoanele la listele lor respective în fața sau în spatele lui P.
    - Dacă poligonul se află în planul ce îl conține pe P, adaugă-l la lista poligoanelor de la nodul N.
4. Aplică algoritmul la lista de poligoane din fața lui P
5. Aplică algoritmul la lista de poligoane din spatele lui P

2.5.PARCURGERE

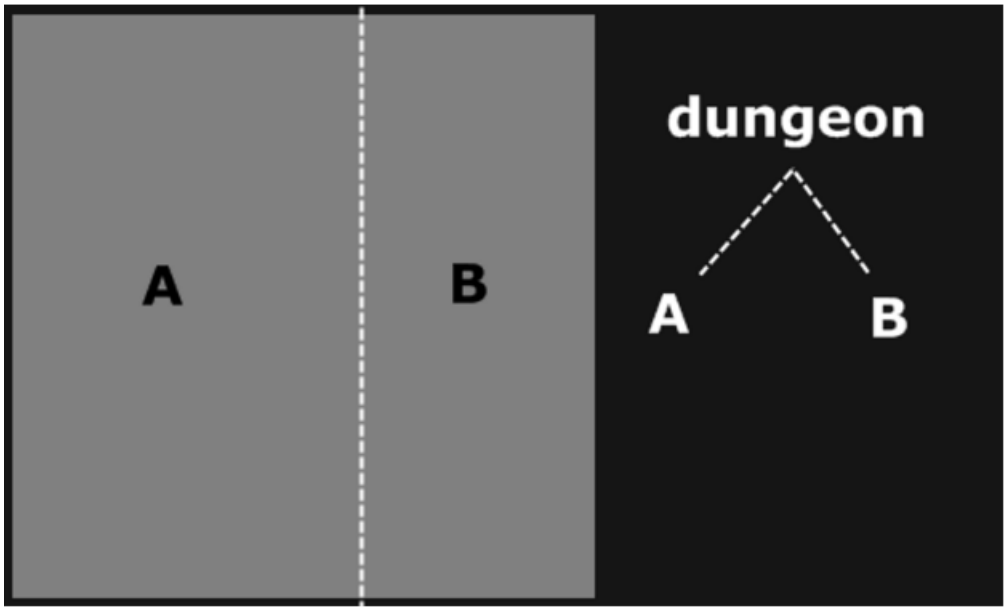
Parcurgerea unui arbore BSP se realizează în timp liniar. Presupunând că modul de implementare al arborilor BSP este cel prezentat mai sus atunci un mod recursiv de parcurgere al arborelui este:

1. Dacă nodul curent este o frunză a arborelui, randează poligoanele de la nodul curent
2. Altfel, dacă locația de vizualizare V se află în fața nodului curent:
    - Randăm arborele copil care conține poligoanele din spatele nodului curent
    - Randează poligoanele de la nodul curent
    - Randează arborele copil care conține poligoane din fața nodului curent
3. Altfel, dacă locația de vizualizare V se află în spatele nodului curent:
    - Randează arborele copil care conține poligoane din fața nodului curent
    - Randează poligoanele de la nodul curent
    - Randăm arborele copil care conține poligoanele din spatele nodului curent
4. Altfel locația de vizualizare V se află pe planul asociat cu nodul curent și trebuie să:
    - Randeze arborele copil care conține poligoane din fața nodului curent
    - Randăm arborele copil care conține poligoanele din spatele nodului curent

2.6.EXEMPLE DE UTILIZARE

Un mod intuitiv în care se pot utiliza arbori BSP este pentru partiționarea suprafeței unei case în încăperi sau realizarea hârţii unui dungeon pentru un joc de tip D&D, această abordare este un caz particular de BSP descris în paragrafele anterioare și este o versiune mai simplă, astfel :

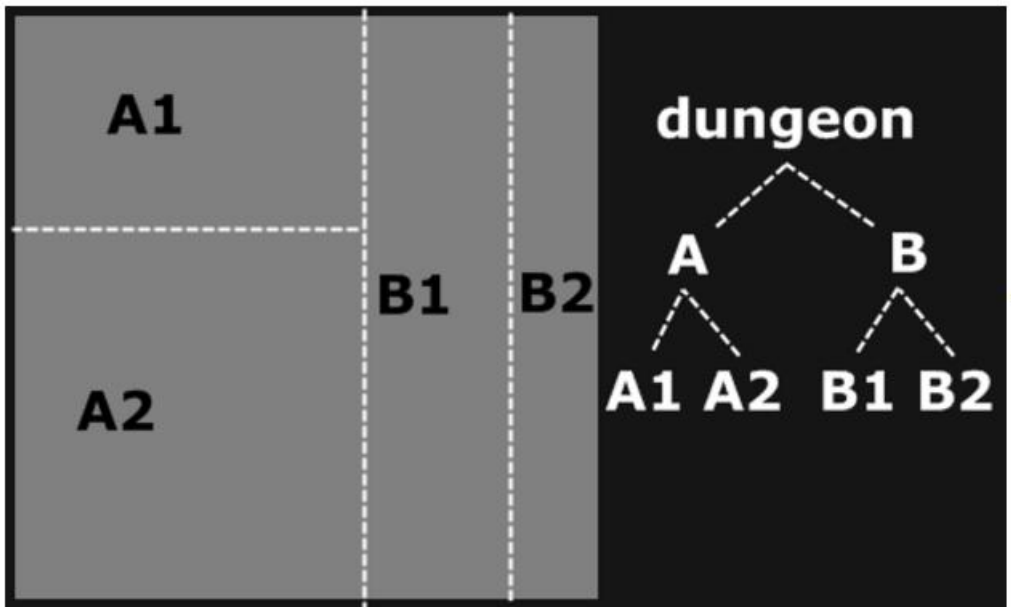
Mapăm spațiul ce dorim a fi partiționat pe un arbore binar, în cadrul căruia primul nod reprezintă spațiul total (pe care noi îl vom numi încăpere) iar frunzele arborelui vor reprezenta subdiviziunile încăperii (marcate cu A respectiv B)



[10]

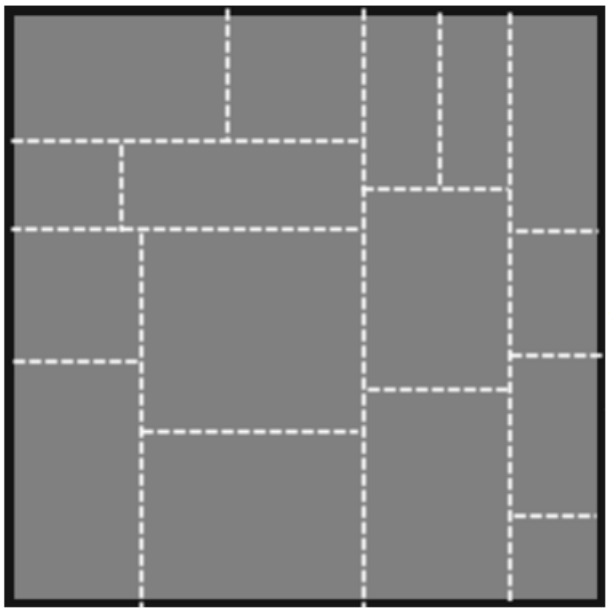
Astfel, dacă mai adăugăm un nivel de noduri la arbore vom obține 4 subdiviziuni ale încăperii, marcate cu A1, A2 pentru subdiviziunile lui A și B1 și B2 pentru subdiviziunile lui B.

Acest proces poate fi repetat până obținem un număr satisfăcător de subdiviziuni ale spațiului inițial. Cu toate acestea folosirea arborilor BSP ne restricționează din punct de vedere al flexibilității numărului de subdiviziuni generate permițând numai partiționarea încăperii într-un număr par de subdiviziunii (ca să fim mai precișsi puteri ale lui doi: 2, 4, 8, 16).



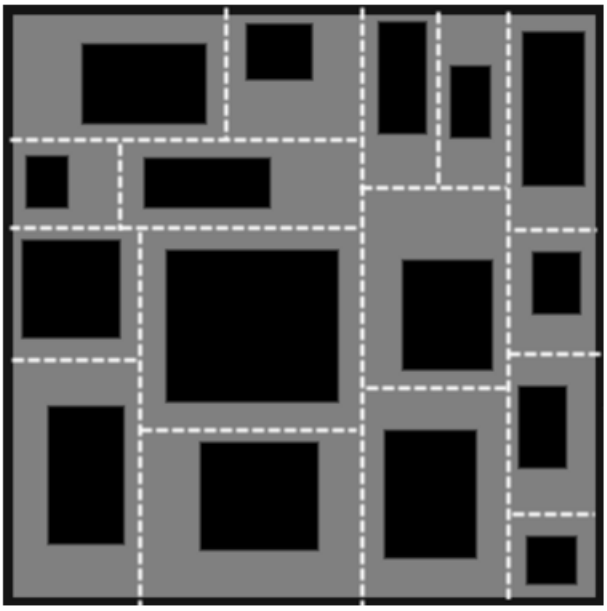
[10]

După partiționarea unui spațiu în subdiviziuni

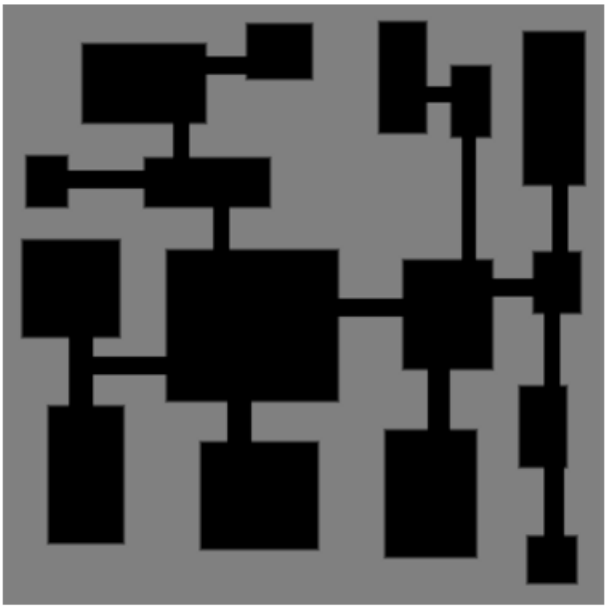


[10]

se generează încăperi în interiorul acestor subdiviziuni (este preferată această abordare în cazul în care dorim ca încăperile să nu aibă pereți care se ating) iar aceste incăperi sunt legate prin holuri.



[10]

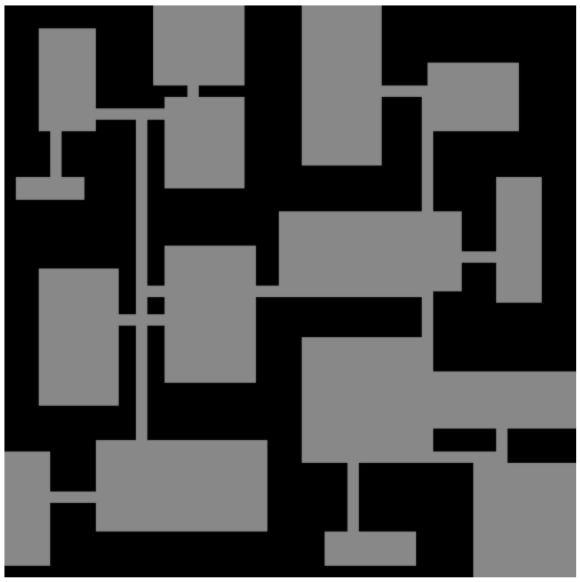
 [11]

O astfel de abordare poate fi observată în jocul "Enter the Gungeon", și este exemplificată în aplicația descrisă în paragraful următor.

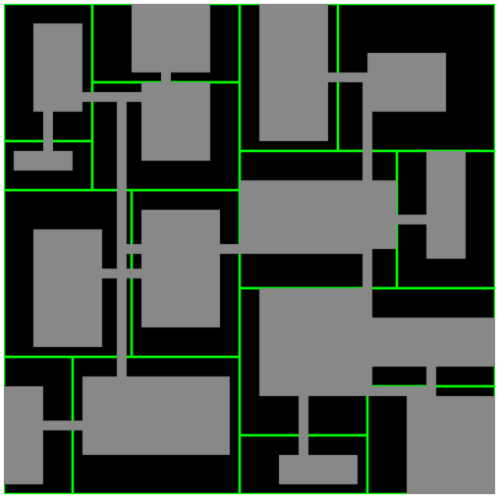
2.7.APLICAȚIE

Aplicația este realizată în Html și JavaScript și generează un exemplu de dungeon, această aplicație este o implementare a unui algoritm BSP. în mod precis aplicația divide o suprafață în două pentru un număr de iterații.

Pentru fiecare partiție din fiecare iterație dimensiunea și direcția diviziunii este stabilită la întâmplare, fie vertical fie orizontal. Mai târziu pe frunzele arborelui se pot genera încăperi în mod aleator sau după anumite reguli în caz că vrem să obținem anumite tipuri de încăperi (ex: încăperi care nu au pereți comuni cu alte încăperi), iar la final se conectează mijlocul fiecărei diviziuni cu diviziunea frate (diviziunea demarcată de un nod frate în arbore) pentru a realiza drumuri între încăperi.

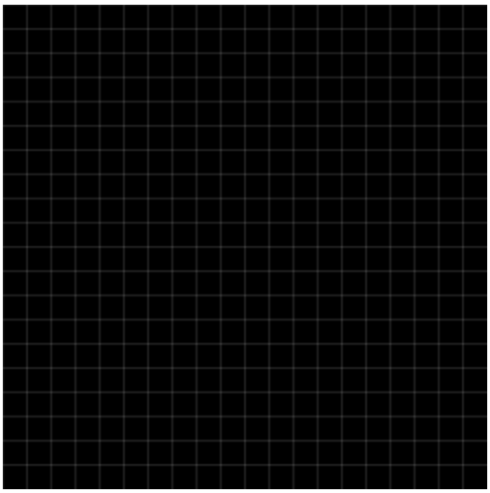


Această implementare permite camerelor generate să fie unite (sa aibă pereţi comuni) pentru a putea realiza încăperi cu forme mai complexe, în cazul în care nu se dorește ca încăperile să aibă pereți comuni trebuie stabilit un buffer între încăpere și partițiile spațiului total (partiții marcate cu verde)



Hărțile generate de aplicație au dimensiuni ce sunt reprezentate de pătrate de o lungime nedefinită X (această metodă de a defini distanțe este standard în D&D iar dimensiunea care este alocată lui X în mod normal este de 5x5 feet)

Astfel dacă setăm dimensiunea hârtiei ca fiind 20 obținem o hartă de 20x20 pătrate

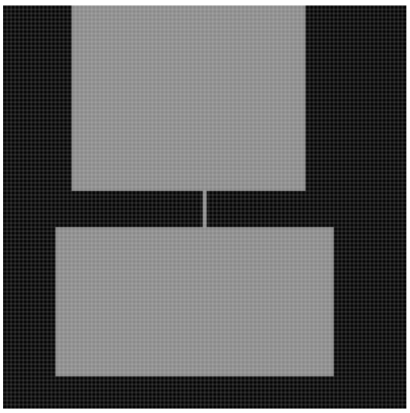
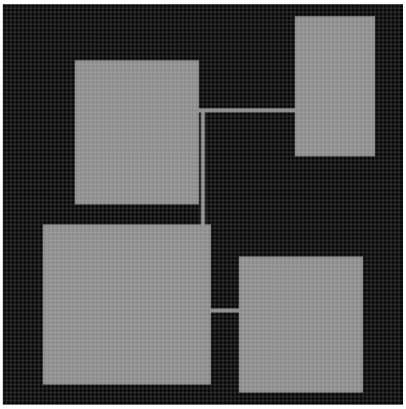
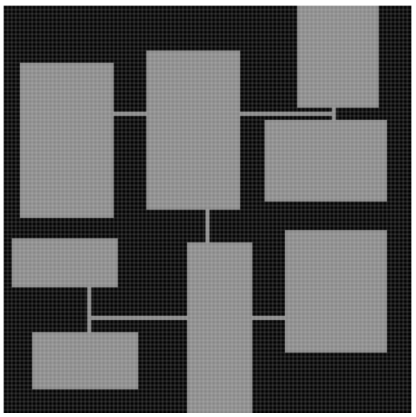


Cum a fost explicat anterior în această lucrare, aplicația folosește un algoritm ce se bazează pe arbori BSP, această implementare impunând o importanță crescută asupra dimensiunii totale a hărții, deoarece noi avem foarte puțin control asupra numărului de camere generate și a dimensiunilor acestora.

Astfel dacă dorim să obținem încăperi mai mari va trebui să creștem dimensiunea spațiului, dar acest lucru nu va garanta creșterea dimensiunii încăperilor ci doar o va favoriza.

Din punct de vedere al încăperilor, aplicația generează încăperi cu ajutorul arborelui BSP mai precis împarte spațiul în subspaţii așa cum a fost descris anterior în lucrare.

Aici se poate remarca modul în care alegerea folosirii arborilor determină numărul încăperilor

și anume faptul că numărul încăperilor este mereu reprezentat de puteri ale lui 2 (în imaginile de mai sus se observă 3 iterații ale algoritmului care produce 2, 4 respectiv 8 încăperi) această limitare poate fi rezolvată prin ignorarea unei porțiuni dintre camerele generate, dar acest lucru nu schimbă faptul că programul a generat deja camerele ocupând procesorul calculatorului, iar această metodă duce la necesitatea de a implementa verificări suplimentare pentru a ne asigura că toate încăperile sunt conectate între ele.

Pentru a putea implementa această aplicație avem nevoie de o implementare a unui arbore

```
var Tree = function( leaf ) {
    this.leaf = leaf
    this.lchild = undefined
    this.rchild = undefined
}
```

și o serie de funcții care să ne permită să ajungem la o anumită frunză sau la un anumit nivel

```
Tree.prototype.getLeafs = function() {
```

```
Tree.prototype.getLevel = function(level, queue) {
```

o serie de funcți pentru a ajuta cu folosirea punctelor dintr-un plan 2D și o definiție a încăperii

```
var Point = function(x, y) {
    this.x = x
    this.y = y
}

var Room = function(x, y, w, h) {
    this.x = x
    this.y = y
    this.w = w
    this.h = h
    this.center = new Point(this.x + this.w/2, this.y + this.h/2)
}
```

După definirea acestor funcții, putem începe implementarea efectivă a arborelui astfel realizăm o funcție care divide încăperile

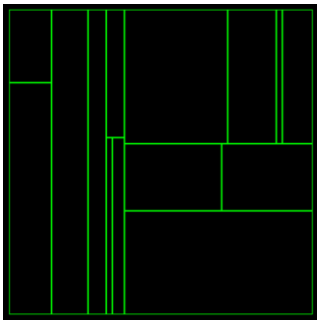
```
function split_room(room, iter) {
    var Root = new Tree(room)
    room.paint(document.getElementById('viewport').getContext('2d'))
    if (iter != 0) {
        var sr = random_split(room)
        Root.lchild = split_room(sr[0], iter-1)
        Root.rchild = split_room(sr[1], iter-1)
    }
    return Root
}
```

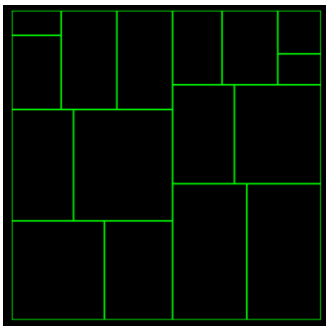
și o funcție care împarte diviziunile în subdiviziuni

```
function random_split(room) {
    var r1, r2
    if (random(0, 1) == 0) {
        // Vertical
        r1 = new RoomContainer(
            room.x, room.y,             // r1.x, r1.y
            random(1, room.w), room.h   // r1.w, r1.h
        )
        r2 = new RoomContainer(
            room.x + r1.w, room.y,      // r2.x, r2.y
            room.w - r1.w, room.h       // r2.w, r2.h
        )
        if (DISCARD_BY_RATIO) {
            var r1_w_ratio = r1.w / r1.h
            var r2_w_ratio = r2.w / r2.h
            if (r1_w_ratio < W_RATIO || r2_w_ratio < W_RATIO) {
                return random_split(room)
            }
        }
    } else {
        // Horizontal
        r1 = new RoomContainer(
            room.x, room.y,             // r1.x, r1.y
            room.w, random(1, room.h)   // r1.w, r1.h
        )
        r2 = new RoomContainer(
            room.x, room.y + r1.h,      // r2.x, r2.y
            room.w, room.h - r1.h       // r2.w, r2.h
        )
        if (DISCARD_BY_RATIO) {
            var r1_h_ratio = r1.h / r1.w
            var r2_h_ratio = r2.h / r2.w
            if (r1_h_ratio < H_RATIO || r2_h_ratio < H_RATIO) {
                return random_split(room)
            }
        }
    }
    return [r1, r2]
}
```

iar în momentul în care este realizată funcția de divizare a încăperilor trebuie să se implementeze o metodă pentru a ne asigura că încaperile vor avea proporții favorabile (în cazul implementării noastre prin H_RATIO și W_RATIO care determină dimensiuni minime pentru lungime și înaltime)

Figura din stânga fără restricții de proporții iar în dreapta cu restricții

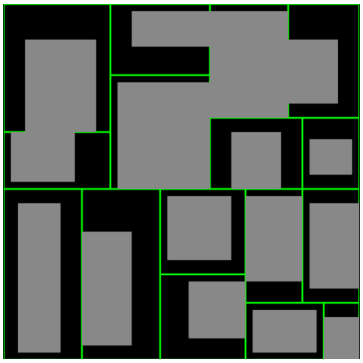




După generarea partițiilor urmează realizarea încaperilor, acest lucru se face în mod aleatoriu prin intermediul funcției:

```
RoomContainer.prototype.growRoom = function() {
    var x, y, w, h
    x = this.x + random(0, Math.floor(this.w/3))
    y = this.y + random(0, Math.floor(this.h/3))
    w = this.w - (x - this.x)
    h = this.h - (y - this.y)
    w -= random(0, w/3)
    h -= random(0, h/3)
    this.room = new Room(x, y, w, h)
}
```

După implementarea porțiunii de generare a camerei obținem:



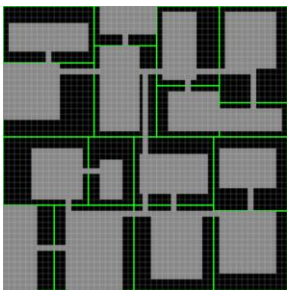
Și mai trebuie doar să unim încăperile între ele care reprezint trasarea unui drum între centrul unei diviziunii și diviziunea frate a sa.

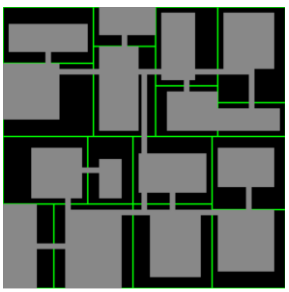
Pentru generarea acestui dungeon aplicația oferă o serie de controale ce pot fi modificate de utilizator pentru a modifică diferite aspecte ale rezultatelor generării.

Dimensiunea hartii	50
Iteratii	4
Elimina conteinere care nu indeplinesc restictiile minime de dimensiune *	☑
ratia verticala a camerei >	0.45
ratia orizontala a camerei >	0.45
Deseneaza gridul	☐
Deseneaza BSP-ul	☑
Deseneaza camerele	☑
Deseneaza coridoarele	☑

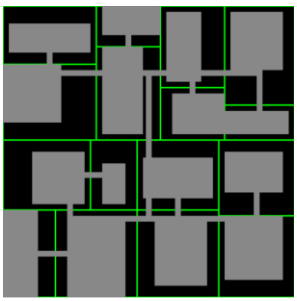
Astfel aceste controale îndeplinesc următoarele funcții:

- **Dimensiunea hărții:** modifică numărul de pătrate, ce indică dimensiunea, din hartă
- **Iterații:** modifică numărul de iterații parcurse de algoritm determină numărul de nivele ale arborelui și prin urmare și numărul de camere obținute
- **Eliminare conteinere care nu îndeplinesc restricțiile minimale de dimensiune:** oprește restricțiile pentru camre cu proporții minime stabilite
- **Rata verticală/ orizontală a camerei:** stabilește procentul de camere cu orientare orizontală respectiv verticală
- **Desenează gridul :** stabilește dacă se deseneză gridul de dimesiune a hărții





- **Desenează BSP-ul:** stabilește dacă se afiseză diviziunile facute de BSP



- **Desenează coridoarele:** determină dacă se afișează coridoarele
- **Desenează camerele:** determină dacă se afișează încăperile

3. L-SYSTEM (Lindenmayer system)

3.1.SCURTĂ PREZENTARE

L-system (Lindenmayer system) este un sistem de rescriere care constă dintr-un alfabet de simboluri care poate fi folosit pentru a creea un șir de simboluri, o serie de reguli care transformă fiecare simbol într-o serie mai lungă de simboluri, un șir de caractere "axiomă" care denotă punctul inițial de pornire și un mecanism de translatare a șirului de simboluri într-o structură geometrică.

Aceste sisteme sunt folosite în general alături de un program de interpretare care poate să transforme șirul de caractere obținut prin intermediul L-system-ului într-o reprezentare grafică.

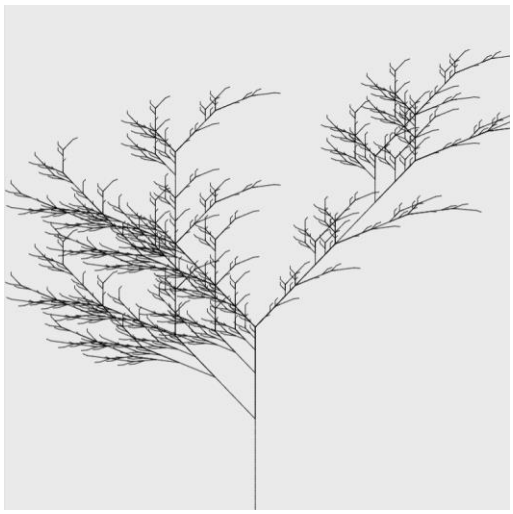
În matematică, logică și programare sistem de rescriere reprezintă un domeniu vast de metode pentru a înlocui sub-termeni a unei formule cu alți termini. Rescrierile pot fi nondeterministice o regulă de rescriere pentru un termen se poate folosii în mai multe moduri sau mai mult de o regulă se poate aplica.

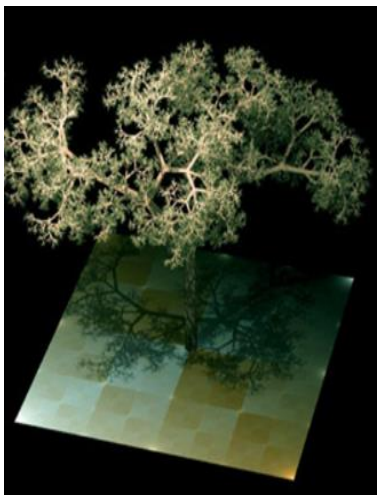
3.2.ISTORIE

L-systems au fost create în anul 1968 de Aristid Lindenmayer un biologist și botanist de la Universitatea din Utrecht. Acestea au fost folosite inițial de Lindenmayer pentru a descrie modul de dezvoltare a algelor și a altor organisme simple dar L-systems au fost modificate ulterior pentru a descrie și plante mai complexe.

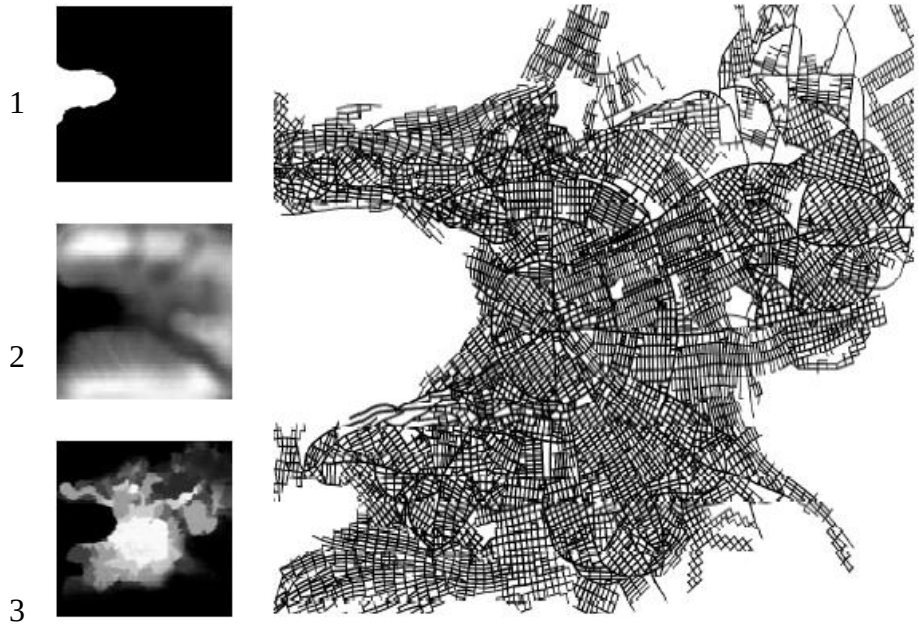
3.3.APLICAȚII

Principala aplicație a L-system este pentru modelarearea de plante[4] dar se pot folosi și pentru generarea de fractali. Plantele generate pot fi atăt în forma 2D sau 3D în funcție de ce L-system și cum acesta este interpretat ulterior. Aceste plante pot fi utilzate ulterior pentru a popula un joc economisind spaţiu de stocare pentru modelele plantelor și creând plante realistice.



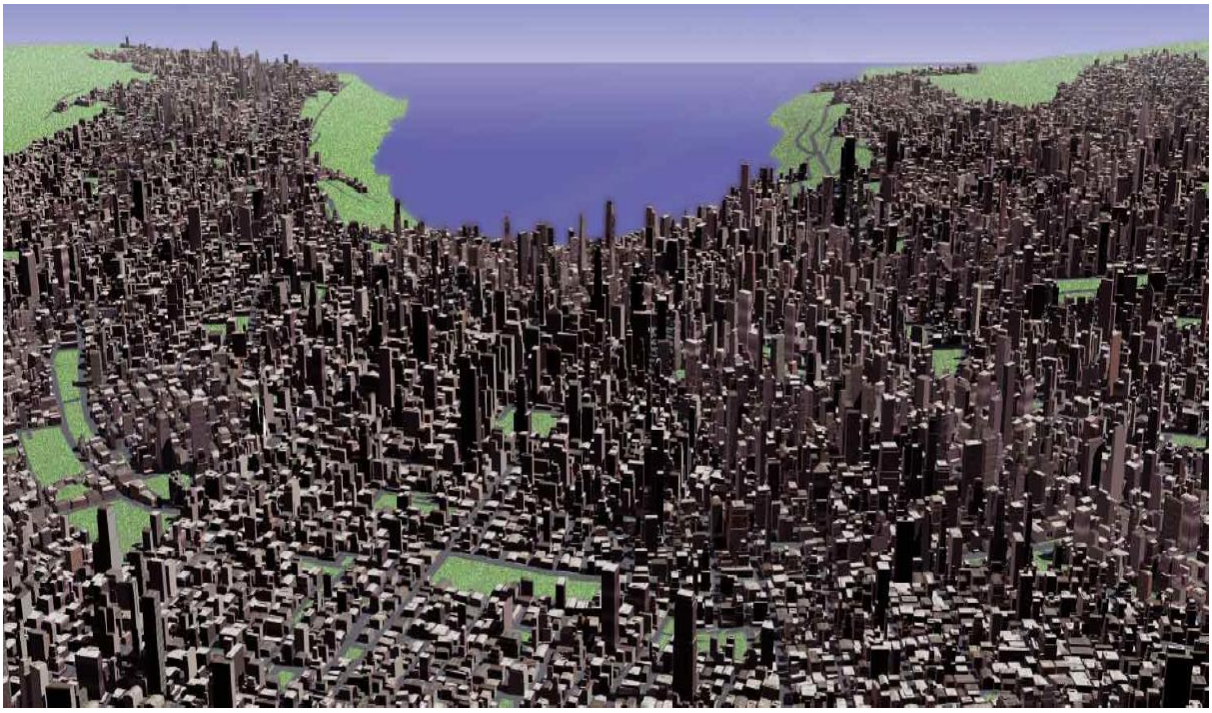


O altă aplicație practică a acestor sisteme este în generarea de hărți de orașe[5] astfel prin intermediul unei hărții de populație, care să indice locurile unde să fie generată harta orașului, și un L-system putem creea o hartă a unui oraș.



Imaginea prezentată cotine și harti pentru tipul de drumuri reprezentate de imaginile 1 și 3

Folosirea de L-systems este utilă în jocurile pe calculator pentru a putea genera un număr mare de conținut cu puțin efort, poate fi folosit aceelași șir de caractere pentru a genera plante, iar dacă se dorește variație între plante acest lucru se poate realiza la interprentarea stringului (variații în numărul de iterații și unghiurile dintre ramurile unei plante), o altă aplicație a acestor sisteme este generarea unui oraş pentru dezvoltatorii unui joc pentru a economisii timp algoritmul descris la[5] putând genera:



3.4.GENERARE:

L-systems sunt definite în mod formal ca fiind un triplet **G**= (*V*, ω,*P*) unde:

- **V** este un alfabet de simboluri care conține elemente ce pot să fie înlocuite (variabile) și elemente care nu pot fi înlocuite (constante sau simboluri terminale)
- **ω** este axiomă și reprezintă un șir de simboluri din V care definesc starea inițială a sistemului
- **P** este o serie de reguli care specifică felul în care variabilele pot fi înlocuite de la o iterație la alta a algoritmului. Acestea sunt împărțite în două parți predecesor și succesor. (predecesor → succesor)

Aceste sisteme încep de la axiomă și continuă să înlocuiască simbolurile dupa reguli în mod iterativ

Un exemplu de un astfel de sistem pentru a descrie modul de dezvoltare a algelor este:

Variabile: X, Y
Axiomă: X
Reguli: (X→ XY), (Y→X)

cu următoarele rezultate:

n= 0: X
n= 1: XY
n= 2: XYX
n= 3: XYXXY
n= 4: XYXXYXYX
n= 5: XYXXYXYXXYXXY
n= 6: XYXXYXYXXYXXYXYXXYXYX
n= 7: XYXXYXYXXYXXYXYXXYXYXXYXXYXYXXYXXY
...

acest proces continuând pentru oricâte n iterații.

Pentru interpretarea acestor șiruri de caractere se folosește o funcție care să transforme șirul de caractere în comenzi de desenat. O metodă pentru implementarea unei astfel de funcții este următoarea:

Se consideră că avem un robot pe o foaie de hârtie. Acest robot se poate mișca înainte și de asemenea poate vira (opțional poate schimba și culoarea folosită pentru a desena sau grosimea liniei etc.)

Un limbaj care poate fi folosit pentru robotul nostru este:

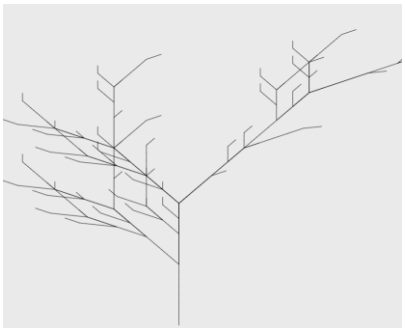
- **F** : robotul se deplasează înainte
- **+** : virare la stănga
- **-** : virare la dreapta
- **[** : pune starea curentă într-o stiva
- **]** : scoate starea curentă din stiva

Stiva se folosește pentru a "ridica" robotul de pe foaie și a-l muta la o stare initială

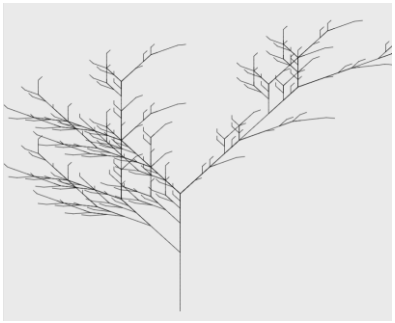
Astfel folosind acest limbaj putem obține următoarele rezultate:

x -> f-[[x]+x]+f[+fx]-x
f -> ff

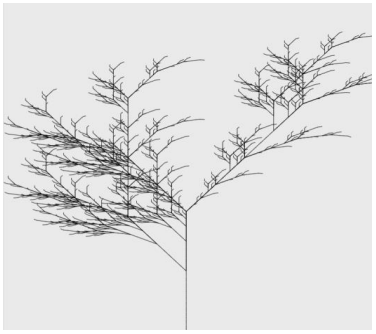
cu starea inițiala x



dupa 4 iterații



dupa 5 iterații



dupa 6 iterații

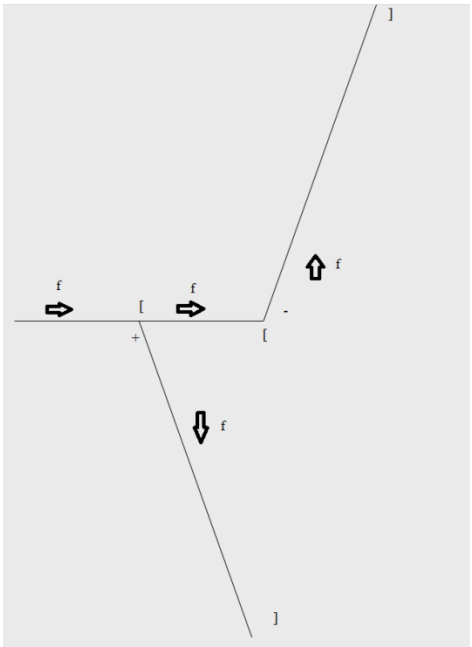
Acest algoritm va fi descris mai în detaliu în aplicația realizată pentru licenţă prezentată în continuare.

Modul în care funcționează limbajul se poate observa în următorul exemplu:

(f → f[+f]f[-f]f --- expresia folosită)

Pentru o iterație (f[+f]f[-f])

Variabile: f
Axiomă: f
Reguli: (f → f[+f]f[-f]f)



După cum se poate observa și în imaginea de mai sus “robotul” pornește cu o mișcare forward(f), după execuția acestei mișcări se notează un punct unde robotul să se întoarcă eventual ([), după marcarea acestui punct “robotul” își reorienează direcția spre dreapta (+) această mișcare putând fi observată numai în momentul în care acesta face o altă mișcare forward(f) , a se renarca că această mișcare se face cu o derivație de la un unghi drept de aceeași dimensiune cu ungiul specificat în

interfața grafică, care deseneza linia ce merge în jos. După această mișcare robotul întâlnește comada de return ([) prin care acesta își va mută poziția înapoi la punctul marcat inițial prin comanda [.

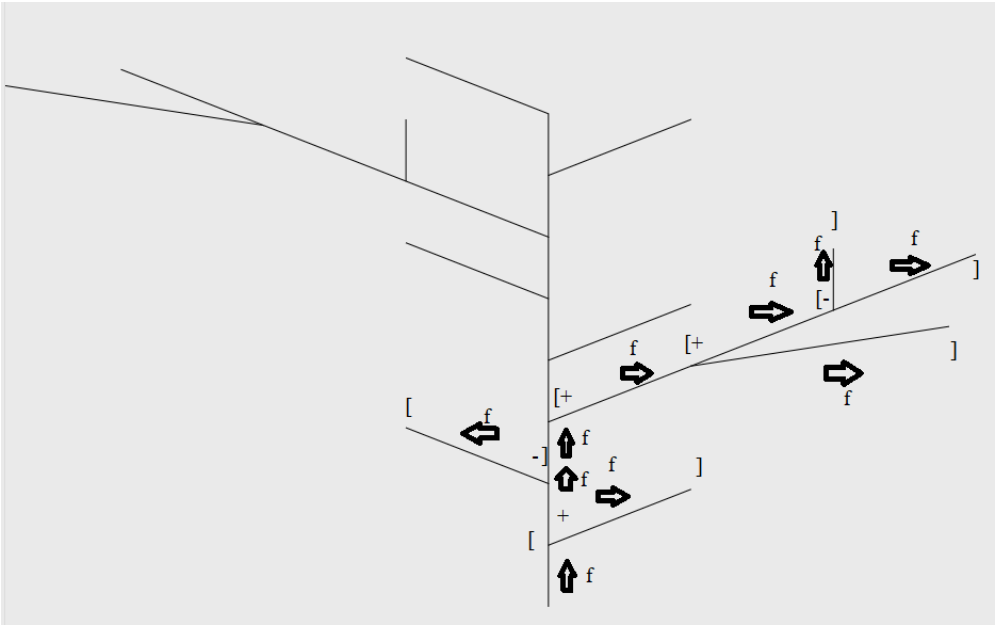
După întâlnirea simbolului] robotul se întoarce la punctul marcat inițial și continu să facă o mișcare de forward(f) după care urmează să repete aceleas operatiouni numai că în direcția opusă

Din acest exemplu simplu se poate observa importanța stivei care este reprezentată de simbolurile [și] aceasta fiind instrumentală în realizarea ramurilor plantei pe care vrem să o realizăm, oferindu-ne o metodă usoră de a reliza ramurile fără a trebuii ca persona care scrie limbajul să se gândească la drumul înapoi de la capătul ramurii până la "tulpina" plantei

Acum dacă observăm a două iterație a acestei genereari putem să observăm foarte ușor cât de rapid crește complexitatea comenzilor pe care trebuie să le realizeze "robotul " astfel noi putem realiza cât de instrumentală este abstractizarea oferită de limbajele L-systems pentru o ușoară generare și înțelegere a unui set de comnezi ce altfel ar fi foarte complexe.

La prima iterație setul de comenzi ce trebuie realizat de "robot" este următorul f[+f]f[-f]f dar în momentul în care se trece la a două iterație numărul de instrucțiuni crește drastic, "robotul" trebuind să realizeze următoarea serie de instrucțiuni:

f[+f]f[-f]f[+f[+f]f[-f]f]f[+f]f[-f]f[-f[+f]f[-f]f]f[+f]f[-f]f



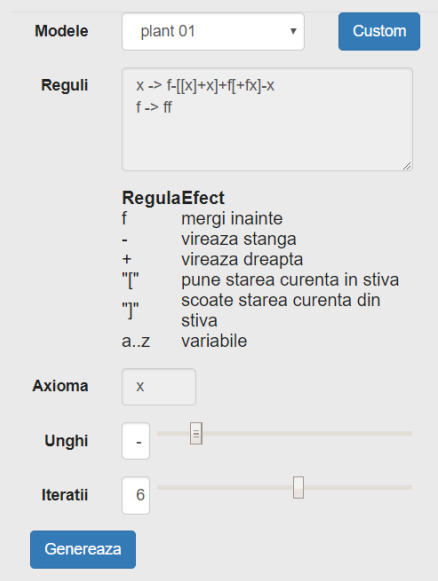
În a doua iterație se poate observa felul în care stiva se folosește pentru a realiza o subramura a ramurii ce provine din tulpină.

3.5.APLICAȚIE

Aplicația este realizată în Html și JavaScript. Această aplicație primește ca input un L-system, o dimensiune de unghi și un număr de iterații, simulând cu ajutorul acestora șirul de caractere ce s-ar obține din L-system după n iterații, urmând ca acest șir de simboluri să fie interpretat în mod grafic.

Programul oferă o interfață grafică pentru a ușura interacțiunea utilizatorului cu programul aceasta, interfața oferind următoarele opțiuni:

- **Modele:** o serie de date de intrare predefinite pentru a demonstra modul de utilizare al aplicației
- **Reguli:** o căsuță de text în care pot fi scrise regulile de rescriere ale L-sistemului
- **Axioma:** definește axioma
- **Unghi:** definește unghiul care va fi folosit în porțiunea de interpretare grafică a programului
- **Iterații:** stabilește numărul de iterații ale L-sistemului care vor fi simulate



Din punct de vedere al implementării aplicația constă din doua porțiuni importante și anume generarea șirului de simboluri după n iterații, ținând cont de regulile impuse de L-system, ce se realizează prin:

```
LSystem.prototype.generate = function (iterations) {
    var i, p;
    var prod = this.seed;
    var newProd = "";

    for (i = 0; i < iterations; i++) {
        newProd = "";
        for (p = 0; p < prod.length; p++) {
            if (this.rules[prod[p]] === undefined) {
                newProd += prod[p];
            } else {
                newProd += this.rules[prod[p]];
            }
        }
        prod = newProd;
    }

    return prod;
};
```

A doua porțiune importantă din implementare este reprezentată de interpretatorul grafic care transformă șirul de simboluri în o formă grafică mai ușor înțeleasă de utilizator:

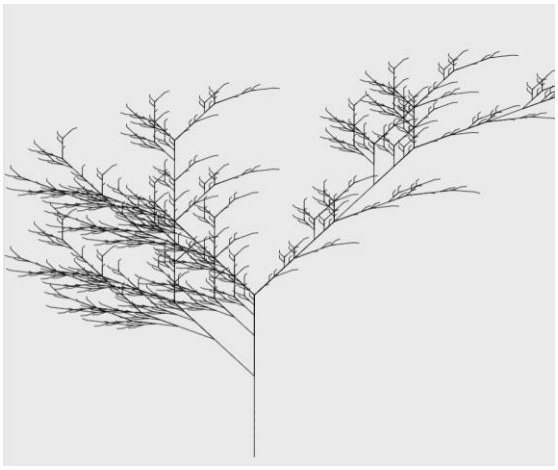
```
LSystem.prototype.compute = function (iterations) {
    var prod = this.generate(iterations);
    var a = 0;
    var i;
    var aStep = this.angle * Math.PI / 180;
    var p = { x: 0, y: 0 };
    this.maxX = this.minX = this.maxY = this.minY = 0;
    this.paths = [];
    var path = [];
    var stack = [];

    for (i = 0; i < prod.length; i++) {
        path.push({ x: p.x, y: p.y });
        switch (prod[i]) {
            case "[":
                stack.push({ x: p.x, y: p.y, a: a });
                break;
            case "]":
                this.paths.push(path);
                path = [];
                var top = stack.pop();
                p.x = top.x;
                p.y = top.y;
                a = top.a;
                break;
            case "-":
                a -= aStep;
                break;
            case "+":
                a += aStep;
                break;
            case "f":
                p.x += Math.cos(a);
                p.y += Math.sin(a);
                this.maxX = Math.max(p.x, this.maxX);
                this.minX = Math.min(p.x, this.minX);
                this.maxY = Math.max(p.y, this.maxY);
                this.minY = Math.min(p.y, this.minY);
                break;
        }
    }
    if (path.length > 0) {
        this.paths.push(path);
    }
};
```

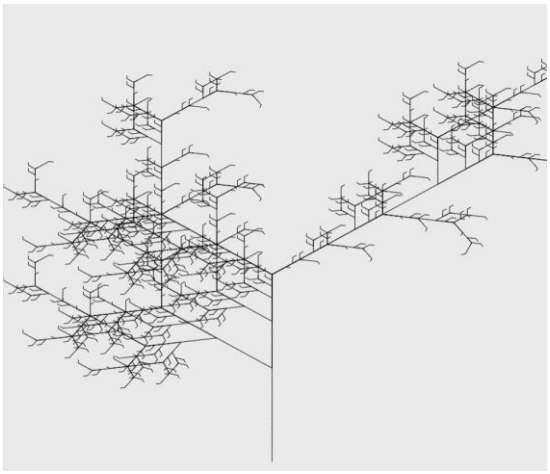
În implementarea noastră pentru a putea desena un arbore, interpretorul asociază fiecare simbol cu o comadă a "robotuli" de desenat.

Un lucru important de observat este folosirea stivei. Această stivă are o importanță deosebită deoarece ne permite să desenăm ramuri dintr-o plantă fară a trebui să desenăm porțiuni irelevante, și anume drumul de la vârful ramurii principale până la locația unde vrem să creem o subramură, astfel stiva oferă atăt avantaje din punct de vedere al esteticii desenului cât și din punct de vedere al eficenței programului.

Stiva memorează locaţia unde apare ramura și permite programului să se întoarcă la această locaţie pentru a începe să deseneze din nou din acel punct.

Aceste desene obținute prin intermediul programului ar avea aplicații practice mai reduse dacă pentru un L-system am putea obține doar un singur desen, astfel pentru a rezolva această problemă se poate modifică unghiul dintre ramuri.





o diferentă de 25^o în unghiul de desenare între cele doua desene (primul desen 25^o și al doilea desen 50^o)

Astfel, prin varierea unghiului putem obține plante cu variații mai mici sau mai mari pentru a putea popula spaţiul unui joc.

4. CONCLUZII SI APLICAȚII

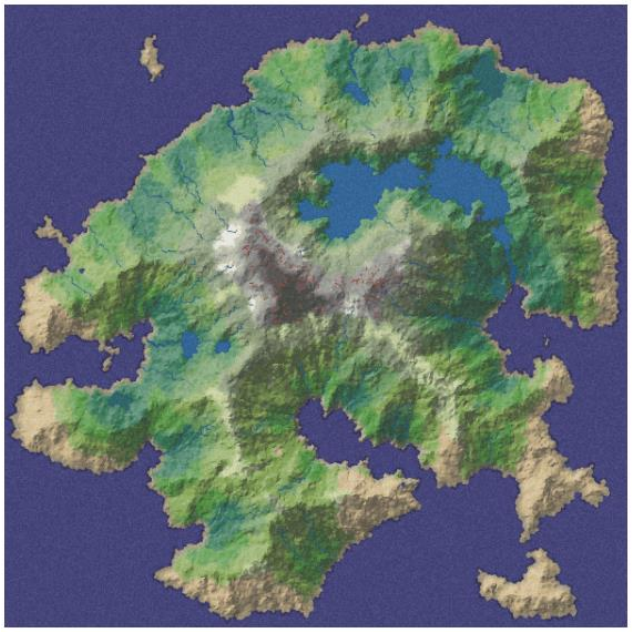
4.1. METODE DE GENERARE DE CONȚINUT PENTRU JOCURI

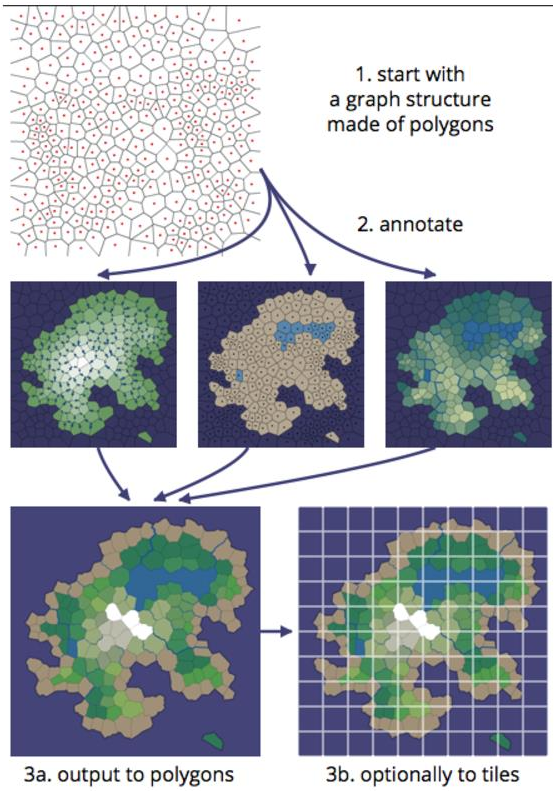
Deși în această lucrare am discutat foarte mult despre metode de generare de date și cum acestea se pot aplica în jocuri, aceste demonstrații individulale de genereare procedurală de date nu ne ofera o imagine cuprinzătoare asupra modului în care aceste elemente se pot combina pentru a crea un joc.

Din acest motiv, în această sectiune din lucrare va fi prezentat un joc ce poate fi creat în mod teoretic cu elemente deja existente în literatura din acest domeniu.

Orice joc necesită un spaţiu în care jucătorul poate să interacționeze cu elementele jocului, astfel primul lucru pe care ar trebui să îl generăm în joc este universul jocului.

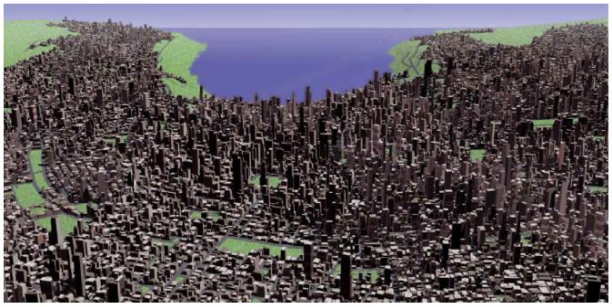
Pentru a realiza acest lucru vom începe prin generarea unui continent pentru joc o metodă pentru generearea unei harți este un algoritm care foloseste un graf de poligoane[12] acest algoritm putând genera o hartă pentru jocul nostru.





Deși acest algoritm nu ne oferă cel mai precis control asupra formei continentului, acesta poate fi mai mult decat îndeajuns.

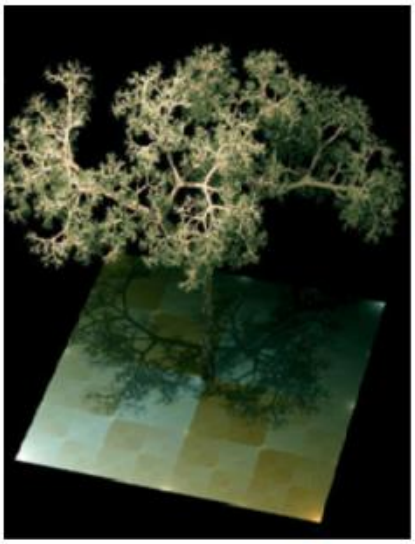
După ce obținem o hartă pentru continentul nostru va trebui să generăm orașe pentru jucător, acest lucru putând fi realizat în mod ușor prin algoritmul[5] prezentat anterior, prin intermediul acestui algoritm noi putând genera orașe ușor urmâd ca apoi să modificăm în mod manual anumite elemente din joc pentru a realiza evenimente sau locuri speciale în oraș.



Dupa ce creem orașele, acestea pot fi conectate prin intermediul unor algoritmi de găsire a celui mai scurt drum.

După ce conectăm orașele, tot ce mai trebuie să facem este să populăm universul jocului, iar în această porțiune generarea procedurală poate ajuta cu generarea de diverse tipuri de plante prin intermediul L-systems care au fost prezentate anterior, astfel putem crea tufișuri sau altfel de plante cu o structură simplă prin intermediul algoritmului prezentat anterior în lucrare

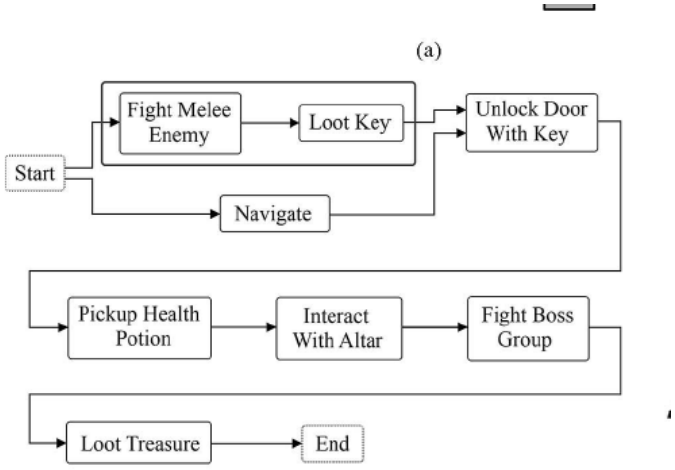
sau putem să generăm copaci sau alte plante 3D cu algoritmi mai avansați



Odată ce finalizăm această porțiune vom obține un univers robust în care să se desfășoare acțiunile jocului nostru.

Deși nu este posibil ca generarea procedurală să genereze o poveste care sa fie excepționala din punct de vedere al calității, în momentul de față este foarte posibil să generăm povești pentru porțiuni mai puțin importante din joc, așa numitele side quest-uri.

Acest lucru poate fi realizat prin intermediul unui concept prezentat în Hartsook *et al.* [12] și se bazează pe un concept de insule, unde evenimente cheie sunt prezente (găsirea unei comori, căutarea unei chei, lupta cu un inamic) și o serie de drumrui ce arată modul în care se poate ajunge de la un evemniment la altul.

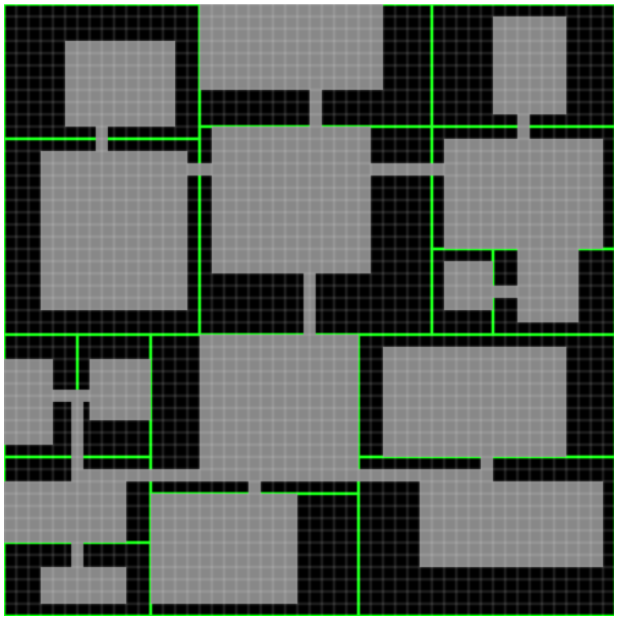


Folosind această metodă și combinând-o cu BSP (Binary Space Partitioning) care a fost prezentat anterior putem genera interacțiuni secundare, de o importanță mai redusă pentru a creea o lume mai complexă a jocului.

4.2.CONCLUZII

În cadrul acestei lucrări am abordat câteva metode de generare procedurală și anume BSP și L-systems. În urma implementării acestor programe se pot observa în mod clar avantajele cât și dezavantajele generării procedurale de conținut.

Principalul dezavantaj care se poate observa în privința generării procedurale este faptul că deși generăm foarte rapid conținutul pentru jocul nostru, acesta s-ar putea să fie plictisitor sau să nu fie foarte util.



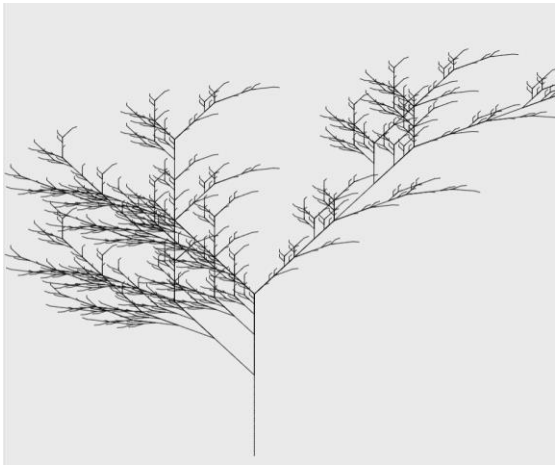
Dezvoltatorul trebuind sa își pună mereu întrebarea "Este harta generată de program în concordanță cu povestea pe care vreau să o transmit?". Această problemă este remediată prin intervenția dezvoltatorului, care poate abstractiza procesul și poate lăsa programul sa stabilească locațiile, dimensiunile și felul în care camerele se leagă între ele, dar camerele sunt foarte atent realizate de dezvoltator.

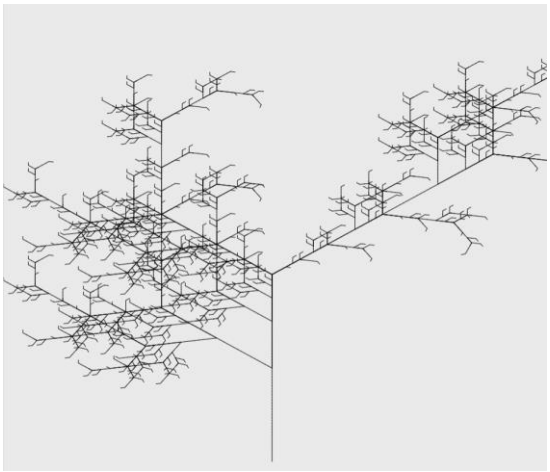
Un exemplu în acest sens este jocul "Enter the Gungeon" care are camere predefinite care sunt aranjate aleatoriu de un program pentru a realiza un nivel "nou".

O altă problemă care s-a putut observa este în aplicația de generare de plante folosind L-systems unde apare o problemă a cunoșterii limbajului L-system pentru a putea genera plante, fapt ce reprezintă un lucru nou de care dezvoltatorii trebuie sa se ocupe, cât și problema varietații plantelor.

Problemea varietații este întrebarea cât de mult pot să modific parametrii de generare ai desenului până când obțin o plantă care nu mai seamănă cu cea inițială, dar este considerată de joc ca find același obiect. Acest lucru reprezintă o problemă, mai ales dacă planta poate fi folosită pentru o activitate în joc, jucătorii putând să nu recunoască resursele de care au nevoie, acest lucru ducând la frustrare.

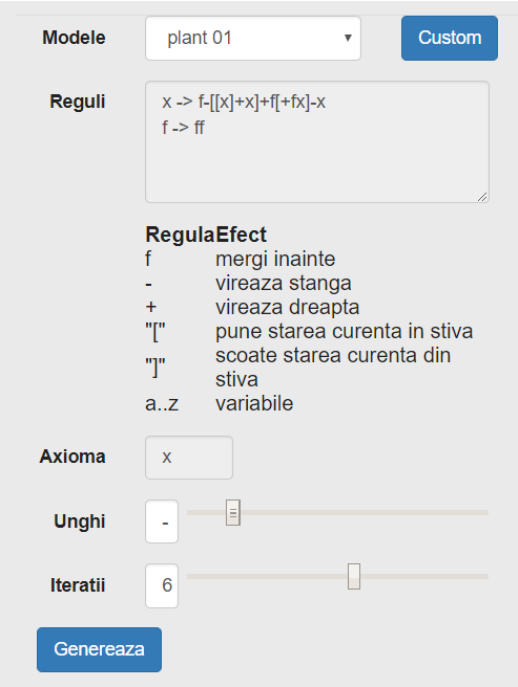
În cazul unei aplicați simple ca cea prezentată în această lucrare, o diferentă de 25^{o} între unghiurile dintre ramuri ducând la un aspect total diferit între cele doua plante rezultate.

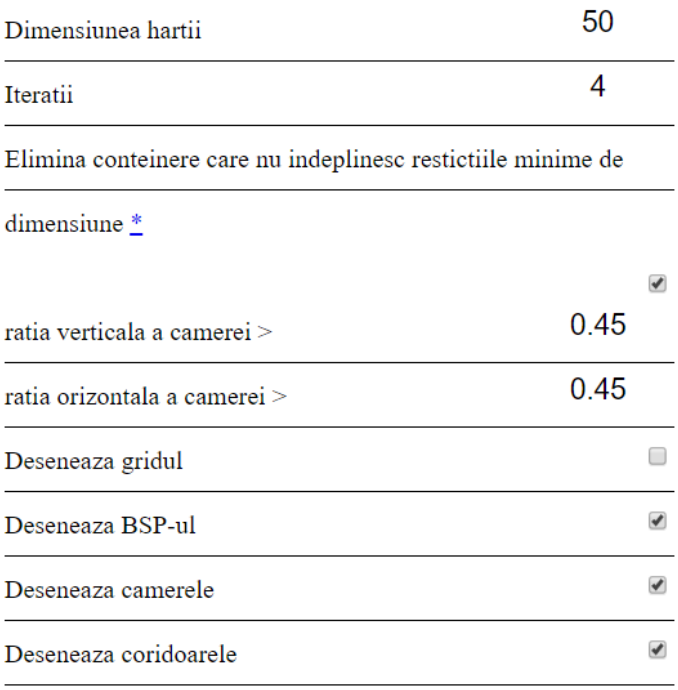




Majoritatea problemelor generării procedurale provin din o problemă mult mai mare și generală a acetui domeniu și anume o lipsă de control a dezvoltatorului asupra procesului de generare.

După cum se poate observa și în aplicatiile detaliate în această lucrare dezvoltatorul pur și simplu introduce valori în casuțe ale programului și obține o hartă sau un dungeon, dar acesta nu poate să modifice lucrurile generate îndeajuns de mult încat să se potivească perfect cu povestea pe care el vrea să o transmită jucătorilor săi.





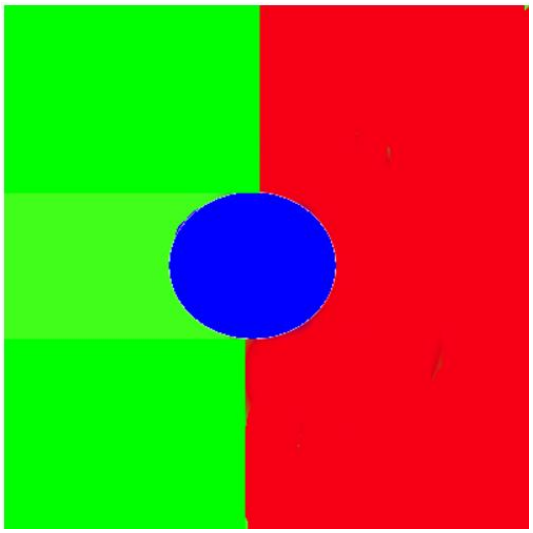
Problema controlului asupra datelor generate este probabil cel mai mare impediment al acestui domeniu. Deși există moduri de a remedia aceste probleme ele nu sunt neapărat satisfăcătoare.

Spre exemplu folosirea de L-systems oferă posibiliatatea de a controla mai mult conținutul, dar acestea nu pot fi folosite petru toate situațiile unde am dorii sa generăm conținut rapid, și de asemenea prezintă impedimentul necesităţii dezvoltatorului să învețe felul în care se scrie un L-system.

O altă soluție care există pentru a remedia problema controlului este combinarea elementelor realizate manual cu elemente generate procedural. Această abordare oferă o soluţie fezabilă dar nu este aplicabilă în toate tipurile de jocuri și poate duce la o experienţă repetitivă, astfel jocul pierzând avantajele pe care le oferă generarea procedurală, și anume un joc care este mereu "nou".

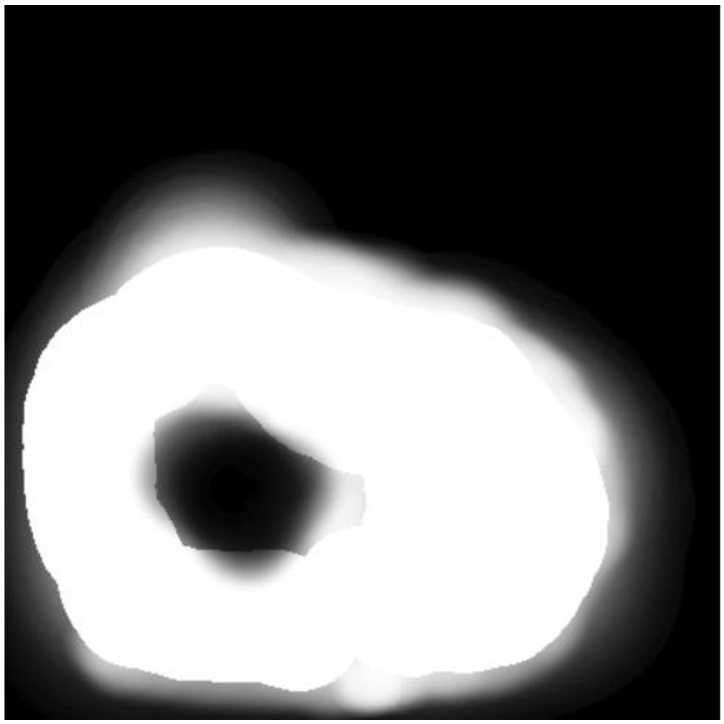
Abordarea care pare să fie cel mai bun compromis pană în momentul de față poate fi observată în generarea procedurală a unui oraş prezentată în lucrarea Procedural Modeling of Cities - Computer Graphics ETHZ - ETH Zürich unde autorii realizează un oraş prin intermediul unor "hărţi".

Aceasta abordare constă în realizarea unor hărţi de populaţie și harţi de model de drum. Mai precis există o hartă care denotă felul în care vor fi generate drumurile (drepte, au curbe, sunt circulare),

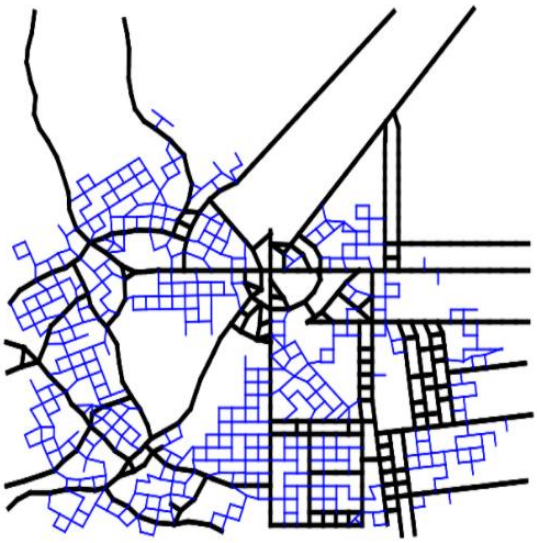


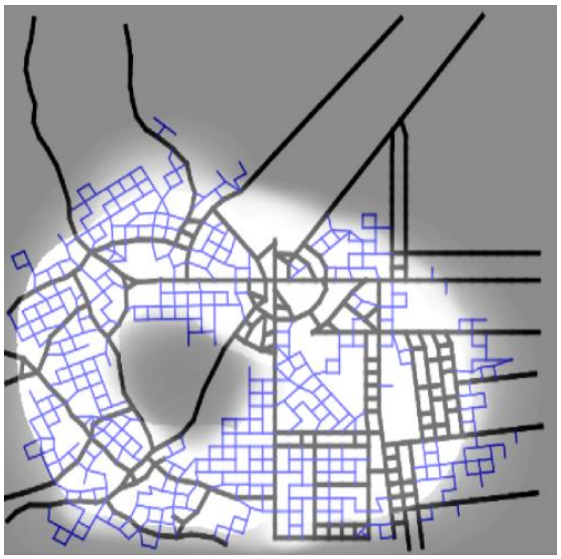
Roșu -drum drept Verde – cu curbe Albastru – circular

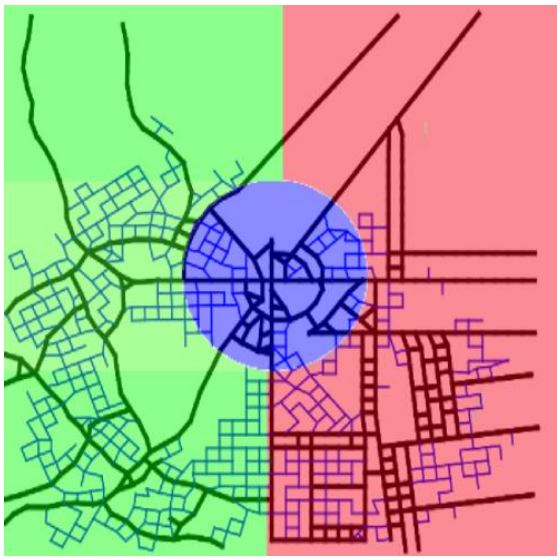
o hartă care denotă locurile unde se află persoane, cu alte cuvinte o hartă a densitații populației



iar prin intermediul acestor două hărti sa fie generată o hartă a orașului



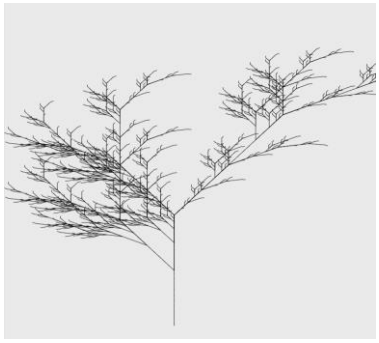


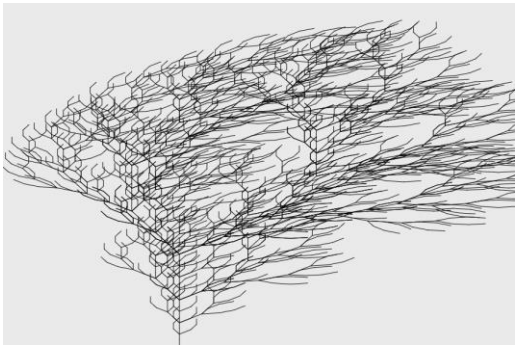


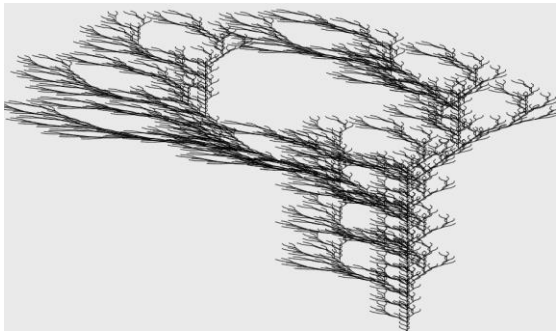
[9]

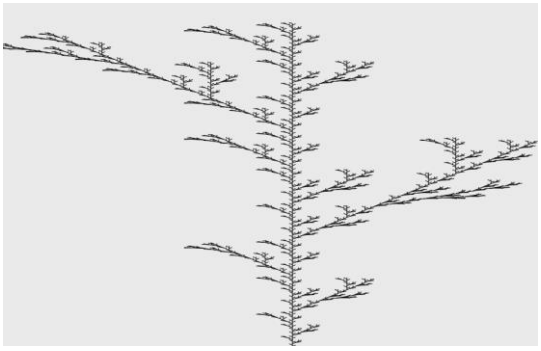
Această abordare permite mult mai mult control, dar într-un final nu este nici ea o soluție datorită lipsei posibilității de a crea zone speciale pentru povestea unui joc.

Cu toate aceste limitări generea procedurală de date prezintă avantaje substanţiale prin uşurința de a obține resurse pentru un joc cum ar fi plantele pentru a decora universul jocului









sau opțiunea de a genera uşor un oraş generic prin care jucătorul poate să treacă lucruri care ar duce la o reducere drastică a costurilor de producție ale jocurilor.

În concluzie generarea procedurală de date reprezintă o perspectivă promițătoare pentru reducerea timpului de dezvoltare a jocurilor căt si costurile de realizare a acestora.

5. BIBLIOGRAFIE

[1] Fuchs, Henry; Kedem, Zvi. M; Naylor, Bruce F. (1980). "On Visible Surface Generation by A Priori Tree1 Structures"

[2] Schumacker, Robert A.; Brand, Brigitta; Gilliland, Maurice G.; Sharp, Werner H (1969). Study for Applying Computer-Generated Images to Visual Simulation (Report). U.S. Air Force Human Resources Laboratory

[3] Chin, N., and Feiner, S., Near Real-Time Shadow Generation Using BSP Trees , *Computer Graphics (SIGGRAPH '89 Proceedings)*, pp. 99-106, July 1989.

[4] Grzegorz Rozenberg and Arto Salomaa. The mathematical theory of L systems (Academic Press, New York, 1980)

[5] Procedural Modeling of Cities - Computer Graphics ETHZ - ETH Zürich, 2001

[6] Smelik et al. 2009

[7] Alexander 1977; strogatz 1994; Eldelstein-Keshet 2005

[8] Hartsook , A Zook, S Das and M Riedl "Toward supporting sories with procedurally generated game worlds"

[9] https://josauder.github.io/procedural_city_generation/#procedural-city-generation-în-python-documentation

[10] http://www.roguebasin.com/index.php?title=Basic_BSP_Dungeon_generation

[11] Chin, N., and Feiner, S., Near Real-Time Shadow Generation Using BSP Trees,*Computer Graphics*

[12] http://www-cs-students.stanford.edu/~amitp/game-programming/polygon-map-generation/

[13] K. Hartsook, A. Zook, S. Das, and M. Riedl, “Toward supporting stories with procedurally generated game worlds,” în *Proc. IEEE Conf. Comput. intell. Games*, Sep. 2011, pp. 297–304.