UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA

FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

LUCRARE DE LICENŢĂ

GENERAREA UNUI ORAȘ ÎN MOD PROCEDURAL

Conducător ştiinţific:
Dr. LAZĂR Ioan, Lector Universitar

Absolvent:
CHIBICI Tiberiu Alexandru

2015

Cuprins

1. Introducere

Cu fiecare generație nouă de jocuri video, ne apropiem tot mai mult de obiectivul final al jocurilor: acela de a ne permite să păşim în alte lumi extraordinare, de a trăi alte vieți, de a experimenta aventuri pe care niciodată nu le-am putea experimenta în viața reală... și fantezia să fie atât de convingătoare încât să ne fie greu să o deosebim de o experiență reală. Pentru realizarea, chiar și în parte, a acestui obiectiv, lumile virtuale simulate în aceste jocuri devin tot mai complexe. Datorită complexității ridicate, cantitatea de muncă necesară pentru construirea unui joc a crescut foarte mult. Pentru a putea face față acestei complexități ridicate, multe companii au investit în dezvoltarea unor motoare de jocuri (*game engines*) și în unelte îmbunătățite. Cu toate acestea, cantitatea de muncă necesară rămâne foarte ridicată, și vedem cum bugetul alocat pentru jocuri ajunge la sume imense de sute de milioane de dolari [1].

O mare parte din munca necesară pentru dezvoltarea unui constă în realizarea modelelor 3D: tot ce ține de mediul virtual în care are loc acțiunea, personajele jocului, unelte, arme, vehicule etc. În general, toate aceste componente sunt construite manual, de către modelatori. Generarea procedurală este un mod de a obține modelele 3D folosind algoritmi și tehnici de programare, în locul modelatorilor.

În această lucrare de licență voi prezenta în mod detaliat modul prin care se poate genera un oraș în mod procedural, și voi prezenta un sistem de generare procedurală a orașelor dezvoltat în motorul grafic Unity3D, în limbajul de programare C#.

Ca și conținut, în capitolul 2 voi explica câteva din noțiunile folosite în acest domeniu și care vor fi folosite mai departe. În capitolul 3 voi trece în revistă diversele abordări prin care se poate realiza generarea procedurală, și ca un caz particular, generarea unui oraș. În capitolul 4 voi prezenta sistemul care generează un oraș în mod procedural pe care l-am dezvoltat. În capitolul 5 voi prezenta câteva exemple de aplicații și jocuri comerciale ce folosesc generarea procedurală.

Fig. 1: Oraș generat în sistemul dezvoltat

2. Noțiuni introductive

Pentru început, voi prezenta câteva noțiuni introductive legate de grafica pe calculator, de jocuri și de tema abordată, de care ne vom folosi mai departe pentru generarea procedurală.

a. Modele 3D

Un model 3D este o reprezentare a unui obiect tridimensional în memoria calculatorului [2]. Modelele stau la baza tuturor aplicațiilor grafice (inclusiv jocurilor).

Există mai multe metode de a reprezenta modelele 3D pe calculator, dintre care cea mai utilizată în aplicații grafice și jocuri este *reprezentarea poligonală*. În această reprezentare, modelul este format din multe poligoane primitive (de obicei triunghiuri sau patrulatere); pentru fiecare poligon, sunt specificate coordonatele vârfurilor. Pentru că un vârf poate apărea în mai multe poligoane, specificarea vârfurilor și specificarea poligoanelor este de obicei separată în două tabele.

Să luăm ca exemplu un cub centrat în origine, cu dimensiunile 2x2x2: în memorie, vor fi stocate următoarele vârfuri:

Indice	Coordonate vârf
0	-1, 1, 1
1	1, 1, 1
2	1, 1, -1
3	-1, 1, -1
4	-1, -1, 1
5	1, -1, 1
6	1, -1, -1
7	-1, -1, -1

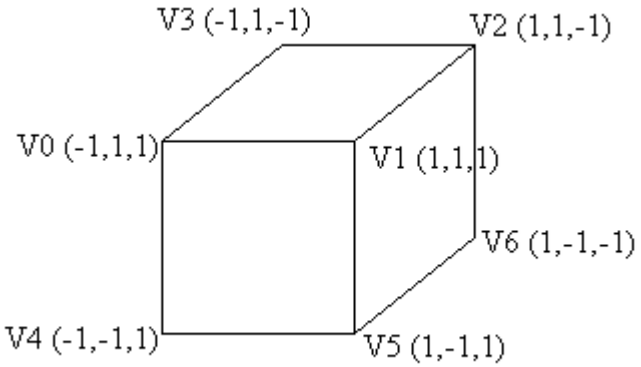


Fig. 2: Cub centrat în origine cu dimensiunile 2x2x2

În al doilea tabel sunt specificate lista poligoanelor

Indice	Indice vârfuri	Descriere
0	0, 4, 5, 1	În față
1	0, 3, 7, 4	Lateral stânga
2	1, 5, 6, 2	Lateral dreapta
3	2, 6, 7, 3	Spate
4	0, 1, 2, 3	Deasupra
5	4, 7, 6, 5	Dedesubt

b. Nivelele de detaliu (*level of detail – LOD*)

Datorită limitărilor hardware, numărul de poligoane ce poate fi desenat la un moment dat pe ecran este limitat. Cu toate că plăcile video moderne devin tot mai performante, această limitare încă este o problemă mare. În același timp, cu cât crește numărul de poligoane cu care este reprezentat un obiect, acesta va arăta mai bine, mai realistic, și este de dorit ca să maximizăm

numărul de poligoane afișat pe ecran, fără a afecta foarte mult performanța. Din acest motiv, s-au dezvoltat diferite tehnici prin care se poate reduce numărul de poligoane afișat.

Una din aceste tehnici este tehnica nivelelor de detaliu [3]: având în vedere că la distanțe mari, detaliile modelelor nu vor fi vizibile, putem construi mai multe versiuni ale modelelor 3D, având un număr diferit de poligoane. Atunci când obiectul este la o distanță mare de cameră, pe ecran este desenat modelul cu mai puține poligoane; atunci când obiectul se află la o distanță mică de cameră, pe ecran este desenat modelul cu mai multe poligoane.

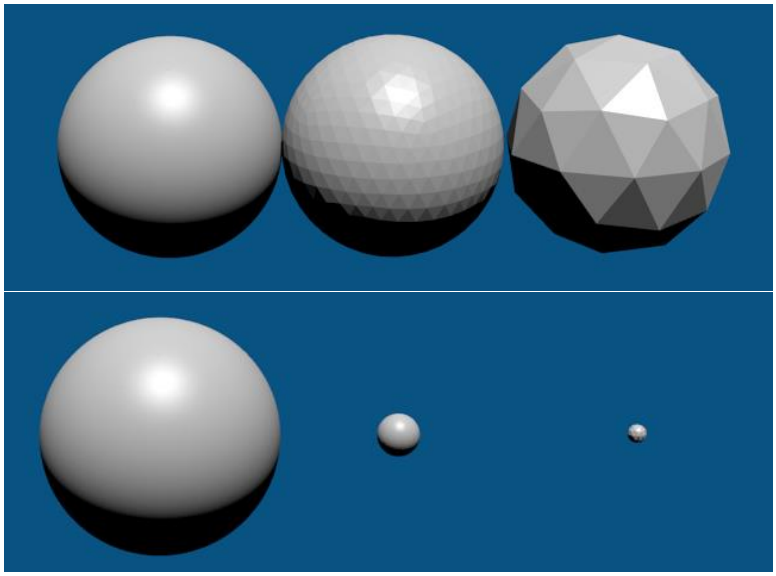


Fig. 3: Sus, o sferă la diferite nivele de detaliu. Jos, sfera la distanțe diferite

c. Harta înălțimilor (*heightmap*)

Pentru reprezentarea unui mediu virtual de dimensiuni mari, putem optimiza memoria utilizată folosind o hartă a înălțimilor, în locul reprezentării geometrice. Harta înălțimilor constă într-un tablou bidimensional cu valori reale, unde indicele liniei și al coloanei reprezintă coordonatele *x* și *z* ale punctului, iar valoarea reală reprezintă coordonata *y* [4].

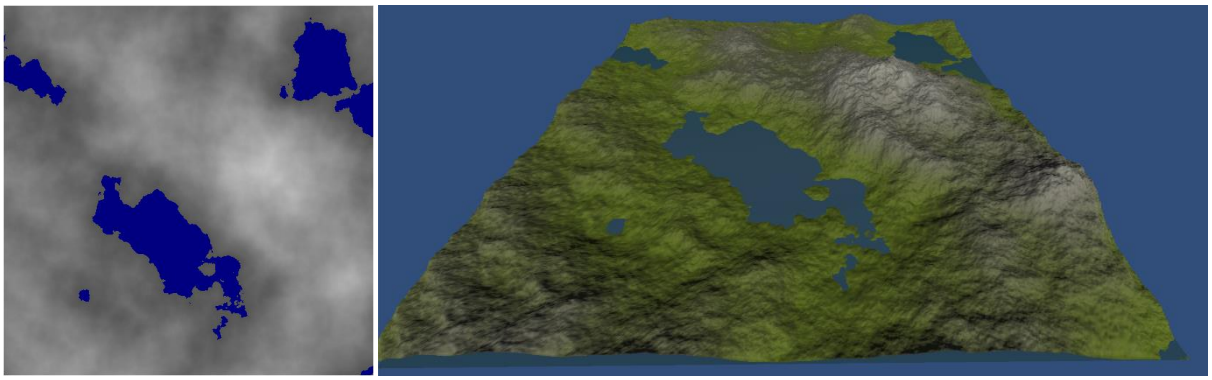


Fig. 4: În stânga, harta înălțimilor (albastru înseamnă apă). În dreapta, terenul corespunzător.

Un mod convenient de a reține tabloul este folosind imagini în tonuri de gri; alb înseamnă înălțimea maximă, negru înseamnă înălțimea minimă.

În aplicațiile grafice, harta înălțimilor trebuie convertită într-un model 3D. După cum am menționat anterior, indicele liniei și al coloanei reprezintă coordonatele *x* și *z* ale punctului, iar valoarea reală reprezintă coordonata *y*, și astfel construim tabelul de vârfuri. Pentru tabelul poligoanelor, vom împărți grila în triunghiuri, după cum este ilustrat în *Fig. 5*.

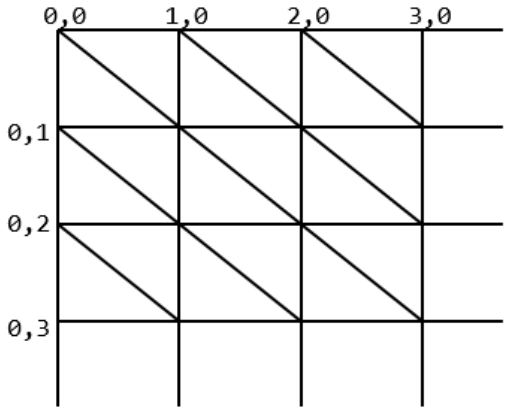


Fig. 5: Împărțirea hărții în poligoane

Există și moduri de a optimiza acest proces de conversie: dacă distanța de la cameră la o regiune a hărții este suficient de mare, putem să calculăm o medie între mai multe celule pentru a desena mai puține poligoane. Atunci când camera se apropie de o zonă în care am calculat folosind media între mai multe celule, putem să recalculăm folosind toate celulele.

d. Funcții de zgomot

Atunci când pornim radioul pe o frecvență pe care nu transmite nicio stație, ceea ce auzim este zgomot [5]. În general, zgomotul este aspectul mai puțin dorit: e mai greu să auzim pe cineva într-o cameră zgomotoasă, decât într-o cameră liniștită. Cu toate acestea, zgomotul face parte din natură, și pentru a genera procedural lumi virtuale care să aibă o structură realistică ne vom folosi de acesta. Vom folosi zgomotul pentru a genera în mod procedural relieful.

La bază, generarea zgomotului implică generarea de numere aleatorii. Diferențele între funcțiile de zgomot constau în modul în care sunt utilizate aceste numere aleatorii.

Zgomotul alb

Cel mai simplu tip de zgomot este *zgomotul alb*. Modul de generare al zgomotului alb este: pentru fiecare punct în care dorim să generăm zgomotul, alegem un număr aleatoriu. În *Fig. 6* se poate observa o imagine care a fost generată folosind zgomot alb, după următorul algoritm: pentru fiecare pixel din imagine, alegem la întâmplare un ton de gri.

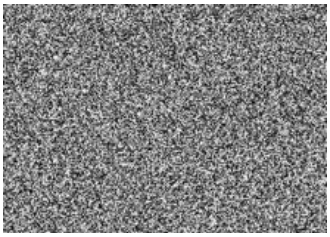


Fig. 6: Zgomot alb

Dacă folosim imaginea generată cu zgomot alb ca și o hartă a înălțimilor, obținem relieful ilustrat în *Fig. 7*. Scopul nostru este de a genera un relief apropiat de realitate; zgomotul alb nu este o funcție potrivită pentru a genera relieful.

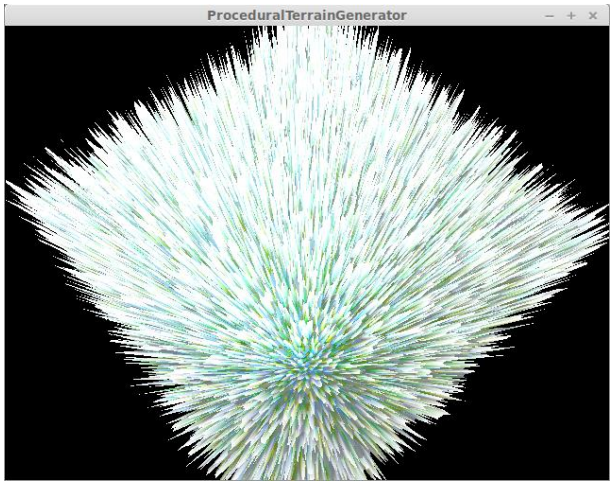


Fig. 7: Relief generat folosind zgomot alb

Zgomotul Perlin

Nemulțumit cu lipsa de naturalețe în zgomotul generat până atunci pe calculator, *Ken Perlin* [6], a dezvoltat o funcție de zgomot ce îi poartă numele, pentru a fi folosită în filmul *Tron* (apărut în 1982).

Pentru generarea hărții înălțimilor avem nevoie de două dimensiuni, așadar mai departe, voi explica modul în care funcționează algoritmul pentru două dimensiuni.

Algoritmul primește ca date de intrare coordonatele *x* și *y* ale punctului pentru care vrem să calculăm funcția de zgomot. Întreg spațiul este împărțit într-o grilă, și pentru început calculăm coordonatele *x0, y0, x1, y1* celulei în care se află vectorul dat ca parametru. În general, celulele au dimensiunea 1x1, deci calculul se reduce la obținerea părții întregi din coordonatele punctului (x, y).

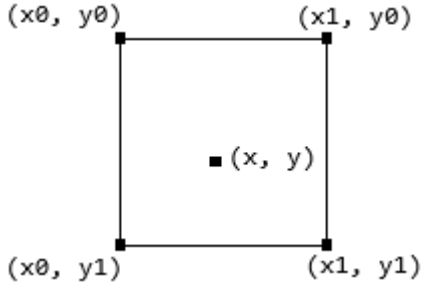


Fig. 8: Calculăm coordonatele x0, y0, x1, y1

Pentru fiecare din cele 4 puncte care delimitează celula, avem generați câte un vector aleatoriu, numit *vector gradient*. În versiunea îmbunătățită [7], vectorii nu sunt aleatorii, ci sunt aleși la întâmplare din cei 12 vectori ce pornesc din centrul unui cub, și sunt perpendiculari pe muchii.

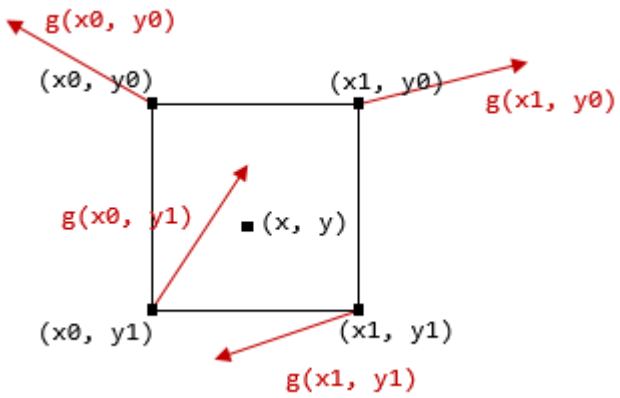


Fig. 9: Vectori gradienți

La următorul pas, trebuie să calculăm vectorii ce pornesc din punctele ce delimitează celula și se opresc în punctul în care calculăm funcția de zgomot, numiți *vectori distanță*.

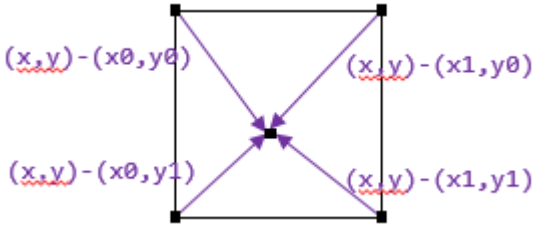


Fig. 10:Vectorii distanță

Mai departe, calculăm produsul scalar între vectorii gradient și vectorii distanță corespunzători fiecărui punct ce delimitează celula, pe care îi vom nota *g00, g01, g10, g11*. În final, vom interpola rezultatele produselor scalare folosind interpolarea liniară, pentru a obține valoarea finală *z* a funcției de zgomot în acel punct.

$$u = \frac{x - x_0}{x_1 - x_0}; v = \frac{y - y_0}{y_1 - y_0}$$

$$z_1 = lerp(g_{00}, g_{10}, u);$$

$$z_2 = lerp(g_{01}, g_{11}, u)$$

$$z = lerp(z_1, z_2, v)$$

Putem îmbunătăți rezultatul obținut dacă folosim o funcție de *easing*. În versiunea îmbunătățită a algoritmului, Perlin recomandă folosirea funcției de *easing*:

$$f(t) = 6t^5 - 15t^4 + 10t^3$$

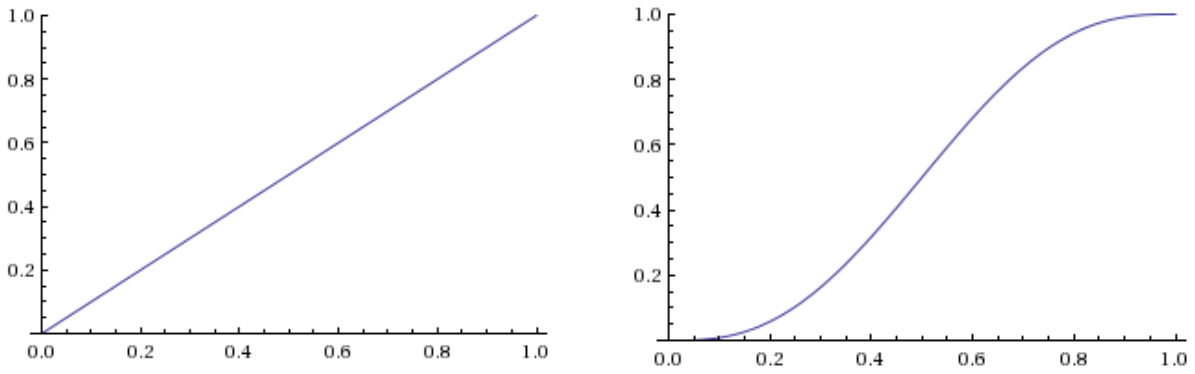


Fig. 12: În stânga: fără funcție de easing, în dreapta: funcția propusă de Perlin

Modul în care calculăm interpolarea se schimbă astfel:

$$z_1 = lerp(g_{00}, g_{10}, 6u^5 - 15u^4 + 10u^3);$$

$$z_2 = lerp(g_{01}, g_{11}, 6u^5 - 15u^4 + 10u^3)$$

$$z = lerp(z_1, z_2, 6v^5 - 15v^4 + 10v^3)$$

O explicație clară și o implementare ușor de înțeles a algoritmului poate fi găsită în articolul [8]. După cum se observă și în *Fig. 11*, această funcție de zgomot este mult mai potrivită pentru generarea reliefului.

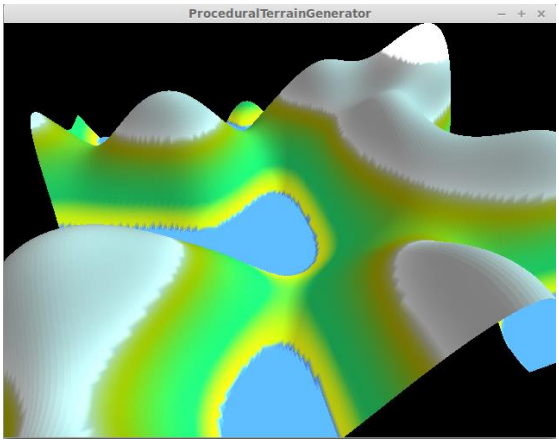


Fig. 11: Relief generat folosind zgomotul Perlin

e. Sisteme Lindenmayer

Un sistem Lindenmayer (sau *L-System*) [9] [10] este definit ca un tuplu $G = (V, \omega, P)$, unde: V este o mulțime de variabile, ω este un șir de simboluri ce definește starea inițială, și P este o mulțime de reguli de producție, ce înlocuiesc variabilele cu alte șiruri de simboluri. Regulile sunt aplicate asupra șirului inițial ω de la dreapta la stânga, obținându-se un alt șir. Procesul se aplică în mod repetat de un număr stabilit de ori.

Să luăm ca exemplu următorul sistem:

Exemplul 1

- *Variabilele*: $V = \{A, B\}$
- *Starea inițială:* $\omega = (A)$
- *Reguli:* $P = \{(A \rightarrow AB), (B \rightarrow A)\}$

După aplicarea regulilor de n ori, obținem:

$n = 0$: A
$n = 1$: $A\ B$
$n = 2$: $A\ B\ A$
$n = 3$: $A\ B\ A\ A\ B$

Ceea ce putem face cu șirul rezultat este să interpretăm fiecare simbol ca o acțiune, ceea ce vom demonstra cu următorul exemplu:

Exemplul 2: Triunghiul Sierpinski

Triunghiul Sierpinski poate fi desenat folosind un sistem Lindenmayer, folosind sistemul:

- *Variabilele*: $V = \{A, B\}$
- *Constante:* $+, -$
- *Starea inițială:* $\omega = (A)$
- *Reguli:* $P = \{(A \to B - A - B), (B \to A + B + A)\}$

Pentru a realiza desenul, vom interpreta șirul rezultat astfel: A sau B înseamnă „mergi înainte", $+$ înseamnă „întoarce la dreapta cu $60°$", $-$ înseamnă „întoarce la stânga cu $60°$".

Iată rezultatele obținute după n iterații:

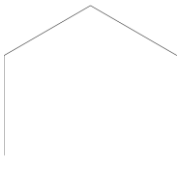


Fig. 13: n=1 iterații

Fig. 14: n=2 iterații

Fig. 15: n=4 iterații

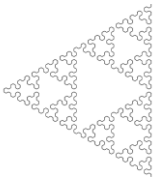


Fig. 16: n=6 iterații

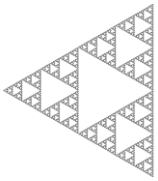


Fig. 17: n=8 iterații

Alte exemple

După cum am observat, sistemele Lindenmayer, cu toate că par un concept foarte simplu, pot fi folosite pentru a genera structuri foarte complexe. Cu regulile potrivite, pot fi generați astfel arbori ce arată natural:

Fig. 18: Arbori generați cu sisteme Lindenmayer

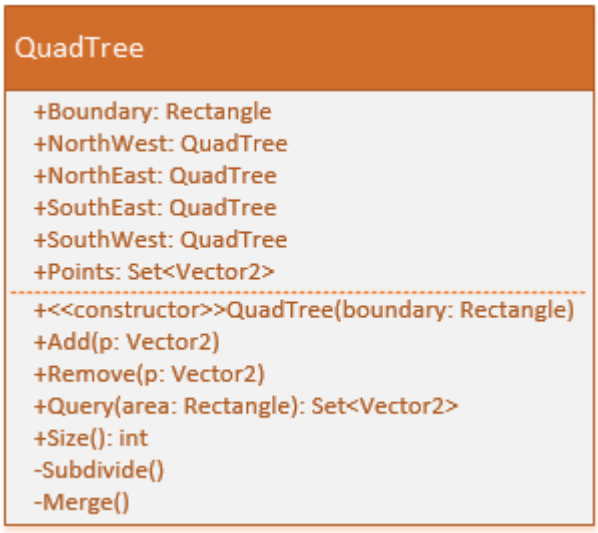
f. Arbori cuaternari (*Quad trees*)

Un arbore cuaternar [11] este o structură de date arborescentă în care un nod are exact 4 fii. Această structură de date este folosită în general pentru partiționarea unei secțiuni din plan,

partiționare realizată prin subdivizarea regiunii în mod recursiv în 4 cadrane. Regiunea este în general rectangulară, dar aceasta poate avea orice formă.

Avantajul principal al folosirii unui arbore cuaternar pentru stocarea punctelor in plan este optimizarea căutărilor: în loc să verificăm pentru fiecare punct dacă aparține zonei în care căutăm, vom verifica doar punctele ce se află în regiunile din arbore care se intersectează cu regiunea căutată.

TAD Arbore cuaternar



Boundary este regiunea acoperită de nod, *Points* este o mulțime ce conține punctele din acest nod. Clasa *QuadTree* mai conține 4 membrii corespunzători celor 4 cadrane în care poate fi subdivizat nodul.

Operația de adăugare

La adăugare, verificăm întâi dacă nodul curent este deja subdivizat. În cazul în care acesta este subdivizat, vom adăuga nodul în unul din cele 4 cadrane. Dacă nodul nu este încă subdivizat, îl adăugăm în lista de puncte. Dacă lista de puncte depășește capacitatea prestabilită, vom subdiviza nodul.

Iată algoritmul scris în pseudocod:

```
Subalgoritm Adaugă(p: Vector2, qtree: QuadTree) este:
        { Adaugă punctul p în arborele cuaternar qtree }
        { Precondiții: p este în interiorul regiunii qtree.Boundary        }
        { Postcondiții: p ∈ qtree }

        Dacă qtree.NorthWest = null și size(qtree.Points) ≥ QTREE_CAPACITY atunci:
                Subdivide(qtree)
        SfDacă

        Dacă qtree.NorthWest = null atunci:
                Adaugă(p, qtree.Points)
        Altfel
                Dacă @ p în interiorul qtree.NorthWest.Boundary atunci:
                        Adaugă(p, qtree.NorthWest)
                Altfel dacă @ p în interiorul qtree.NorthEast.Boundary atunci:
                        Adaugă(p, qtree.NorthEast)
                Altfel dacă @ p în interiorul qtree.SouthEast.Boundary atunci:
                        Adaugă(p, qtree.SouthEast)
                Altfel
                        Adaugă(p, qtree.SouthWest)
                SfDacă
        SfDacă
SfSubalgoritm
```

```
Subalgoritm Subdivide(qtree: QuadTree) este:
        { Subdivide arborele cuaternar qtree în cadrane }
        midX ← (qtree.Boundary.Left + qtree.Boundary.Right) / 2
        midY ← (qtree.Boundary.Top + qtree.Boundary.Bottom) / 2
        qtree.NorthWest ← CreeazăQtree(qtree.Boundary.Left, qtree.Boundary.Top, midx, midy)
        qtree.NorthEast ← CreeazăQtree(midx, qtree.Boundary.Top, qtree.Boundary.Right, midy)
        qtree.SouthEast ← CreeazăQtree(midx, midy, qtree.Boundary.Right, qtree.Boundary.Bottom)
        qtree.SouthWest ← CreeazăQtree(qtree.Boundary.Left, midy, midx, Boundary.Bottom)

        Pentru fiecare punct din qtree.Points:
                Adaugă(punct, qtree)
        SfPentru
        Distruge(qtree.Points)
SfSubalgoritm
```

Operația de căutare

Prin operația de căutare selectăm din toate punctele existente într-un arbore cuaternar pe acelea care se află în regiunea dată ca parametru. Iată o implementare în pseudocod:

```
Subalgoritm Caută(region: Rectangle, qtree: QuadTree, result: Set<Vector2>) este:
        { Selectează punctele aflate în regiunea region. }
        { Postcondiții: result = region ∩ qtree }

        Dacă region ∩ qtree.Boundary ≠ null atunci:
                Dacă qtree.NorthWest = null atunci
                        region = qtree.Points ∩ region
                Altfel
                        Caută (region, qtree.NorthWest, resultNW)
                        Caută (region, qtree.NorthEast, resultNE)
                        Caută (region, qtree.SouthEast, resultSE)
                        Caută (region, qtree.SouthWest, resultSW)
                        region = resultNW ∪ resultNE ∪ resultSE ∪ resultSW
                SfDacă
        SfDacă
SfSubalgoritm
```

3. Generarea procedurală a orașelor

Generarea procedurală este un mod de a obține modele 3D folosind algoritmi și tehnici de programare. Ideea de bază de la care pornește aceasta este observarea și stabilirea anumitor caracteristici din mediul ce dorește a fi generat (de exemplu, din lumea reală), și apoi dezvoltarea unor algoritmi ce pot produce rezultate care oarecum să semene.

La început, această tehnică a fost folosită în jocuri datorită constrângerilor de memorie existente: nu exista suficient spațiu de memorie pentru a include medii virtuale create de dezvoltatorii jocurilor. Ca soluție, aceștia au recurs la generarea procedurală; mediile virtuale puteau fi generate pe loc, în momentul în care utilizatorii foloseau jocul.

Astăzi, deși algoritmii de generare procedurală au fost îmbunătățiți de-a lungul timpului, sunt puține jocuri în care aceștia sunt folosiți, și în majoritatea cazurilor acestea sunt jocuri dezvoltate independent (*indie)*, unde bugetul este mult mai limitat și numărul de dezvoltatori este redus.

De ce generarea procedurală?
Există mai multe tehnici prin care se pot obține modele 3D. Cea mai folosită tehnică este *modelarea manuală*, folosind programe de modelare (cum ar fi *3D Studio Max* sau *Blender*). Ca avantaj, modelatorii pot avea un control mult mai fin la detalii, dar dezavantajul principal constă în cantitatea de muncă necesară. Modelarea unui singur personaj poate dura, în funcție de cât de detaliat trebuie să fie și cât de priceput este modelatorul, de la câteva ore la câteva săptămâni. Ținând cont că într-un joc pot exista zeci de personaje, pe lângă întreg mediul virtual care trebuie modelat, cantitatea de muncă este imensă.

O altă tehnică apărută mai recent este *scanarea obiectelor reale*, tehnică ce se realizează folosind aparatură și programe speciale. Ca avantaj, folosind această tehnică se pot obține modele 3D mult mai repede, în câteva minute, față de modelarea manuală. Recent au apărut și soluții software care încearcă să reconstruiască modele 3D folosind câteva fotografii la unghiuri diferite. Dezavantajul principal este prețul acestei tehnologii; chiar dacă există și soluții foarte ieftine, calitatea modelelor produse de acestea lasă mult de dorit. Un scanner de calitate costă câteva mii de dolari. Un alt dezavantaj este că aceste scannere pot scana doar obiecte relativ mici, și nu pot fi folosite pentru a modela medii virtuale mari.

Generarea procedurală este o altă tehnică de a obține modele 3D. Avantajul principal constă în infinitatea de rezultate diferite ce se pot obține: folosind această tehnică, se pot genera o infinitate de lumi, care pot fi infinit de mari. Astfel, jucătorii vor petrece mai mult timp jucând, pentru că fiecare joc nou este diferit și unic (calitate ce se numește *replay-value*). De asemenea, generarea procedurală rezolvă și o parte din problemele existente la celelalte tehnici: costul și timpul de dezvoltare sunt mult reduse, și pot fi generate medii oricât de mari. Un alt avantaj important este posibilitatea refolosirii sistemelor generatoare: un sistem generator poate fi refolosit în mai multe jocuri. Ca dezavantaj, unii algoritmi nu sunt ușor de implementat, și efortul care ar fi depus pentru modelare este redirecționat către dezvoltarea algoritmilor. De

asemenea, unii proiectanți preferă alte tehnici pentru că nu au destul control asupra mediilor create.

Generarea orașelor

Într-un număr semnificativ de jocuri acțiunea se petrece în orașe. Ținând cont de dimensiunea unui astfel de mediu virtual, cât și de complexitatea acestuia, modelarea manuală poate dura foarte mult timp. În acest caz, generarea procedurală ar putea economisi mult timp și efort. În continuare, vom studia cum putem genera în mod procedural un oraș.

Generarea constă în parcurgerea mai multor pași: pentru început vom genera relieful pe care să așezăm orașul, apoi vom genera rețeaua stradală, vom aloca spații pentru clădiri, vom genera clădirile, și în final vom umple spațiile rămase libere cu vegetație.

a. Generarea reliefului

Prima etapă în generarea orașului constă în generarea reliefului. Cea mai folosită tehnică de generare folosește funcția de zgomot Perlin, care a fost descrisă în capitolul 1. În plus față de funcția de zgomot Perlin vom folosi o tehnică specifică fractalilor, și anume *octave* [12], pe care le vom ilustra folosind funcția sinus.

Octave

Pentru funcția sinusoidală, definim ca *lungime de undă* distanța dintre două valuri, și ca *amplitudine* înălțimea unui val. *Frecvența* este numărul de valuri pe o unitate de timp (axa *Ox*), adică $\frac{1}{\text{lungimea de undă}}$.

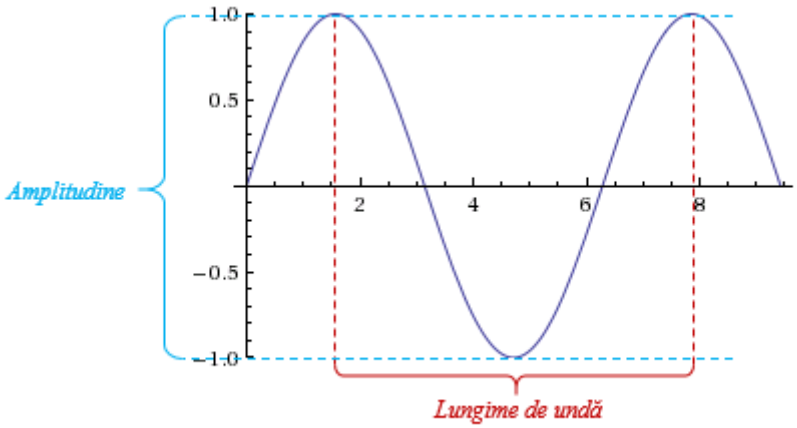


Fig. 19: Lungimea de undă și amplitudinea funcției sin(x)

Putem mări sau micșora amplitudinea funcției dacă înmulțim rezultatul obținut cu o constantă. Putem altera și frecvența funcției, dacă înmulțim argumentul *x* cu o constantă:

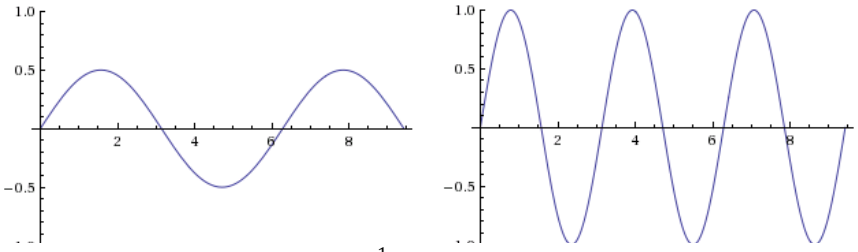
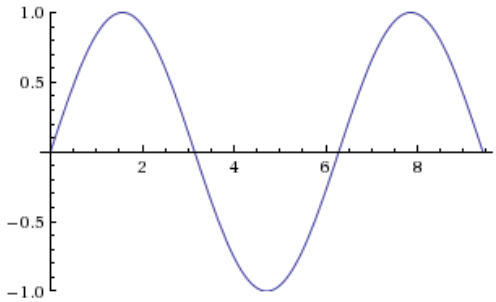
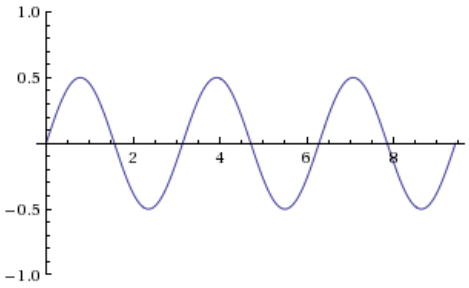


Fig. 20: Stânga: $\frac{1}{2}sin(x)$*. Dreapta:* $sin(2x)$

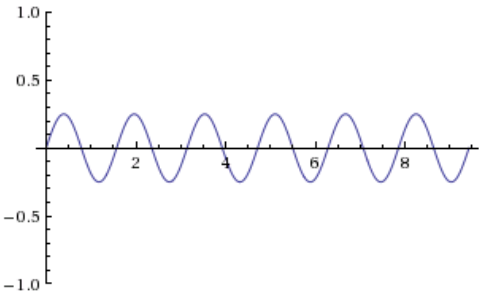
Vom începe cu frecvența și amplitudinea 1, și vom genera *n* funcții. La fiecare pas, dublăm frecvența, și înmulțim amplitudinea cu o constantă numită *persistență*. Fiecare astfel de funcție generată se numește o *octavă*. În final, adunăm funcțiile obținute.



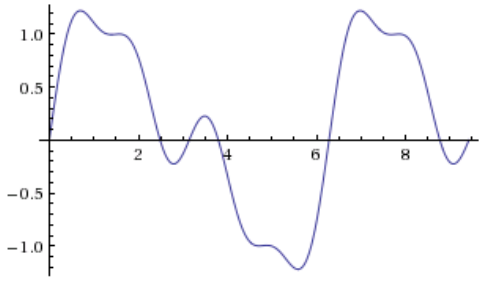
Frecvența=1, Amplitudinea=1



Frecvența=2, Amplitudinea=0.5



Frecvența=4, Amplitudinea=0.25



Suma funcțiilor

Fig. 21: Funcția sinus în 3 octave

Dacă aplicăm același algoritm pentru funcția de zgomot Perlin, obținem:

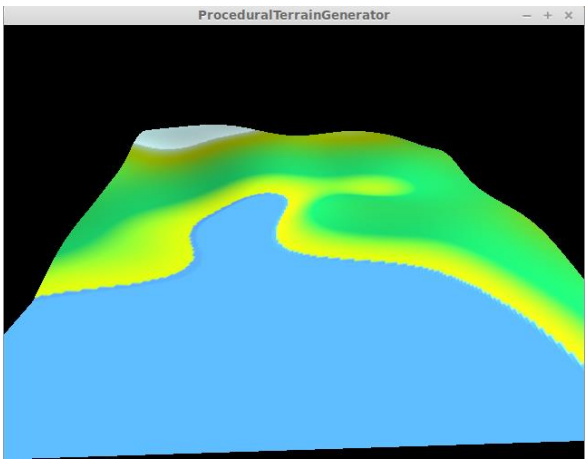


Fig. 22: Zgomot Perlin, 2 octave

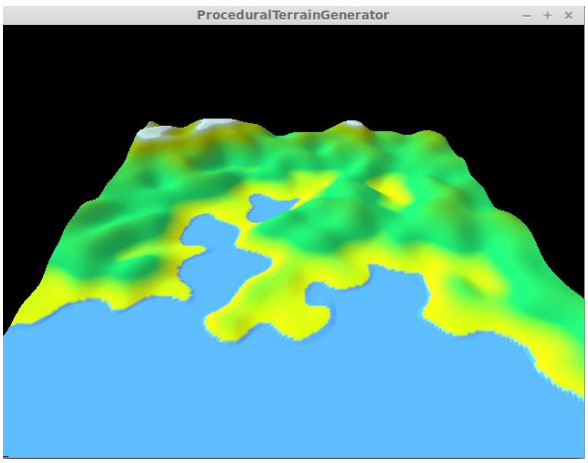


Fig. 23: Zgomot Perlin, 4 octave

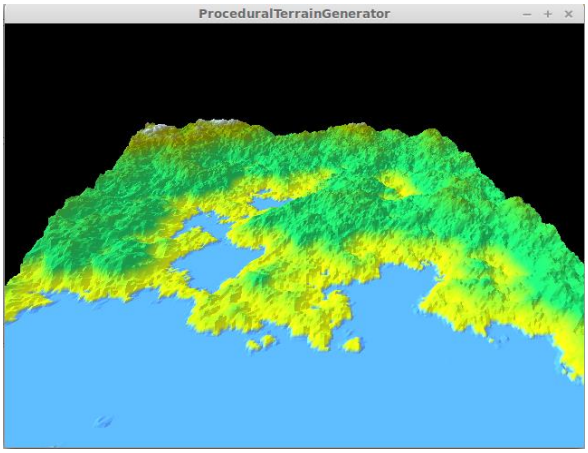


Fig. 24: Zgomot Perlin, 8 octave

Eroziunea

Eroziunea este un proces care se întâmplă în lumea reală și are un efect foarte vizibil asupra reliefului, după cum se poate observa și în *Fig. 25.* Dacă reușim să reproducem acest efect și în relieful generat, putem obține un relief mult mai realistic.

Fig. 25: Efectul eroziunii observat în lumea reală

În articolul „*The Synthesis and Rendering of Eroded Fractal Terrains*” [13] sunt descrise 2 metode de eroziune pe care le putem reproduce: **eroziunea hidraulică**, ceea ce înseamnă simularea efectului ploilor care transportă sediment din zonele mai înalte către zonele mai joase, și **eroziunea termică**, prin care se simulează căderi de pământ, cauzate de casarea produsă de diferențe de temperatură. Aceste metode, în combinație cu zgomotul Perlin pot produce reliefuri foarte realistice. Ca avantaj, *eroziunea hidraulică* poate să ne ofere și albii pentru râuri.

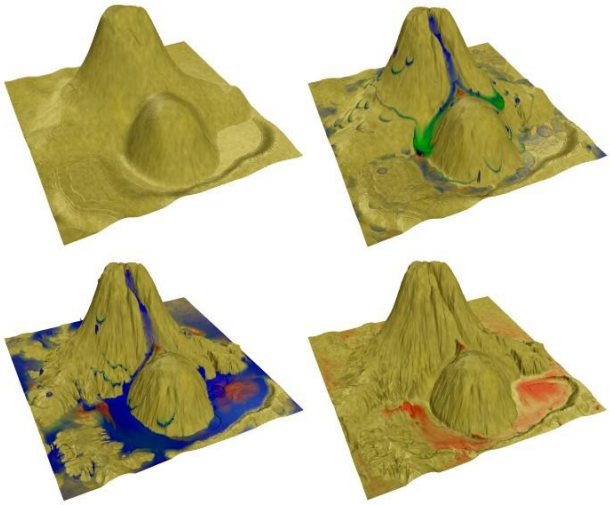


Fig. 26:Efectul eroziunii hidraulice

Plasarea apei
Există mai multe abordări pentru plasarea apei.

Cea mai simplă metodă este să alegem o valoare prag care să fie nivelul apei. Orice punct din harta înălțimilor care va fi sub acest prag, va fi sub nivelul apei. Această abordare este foarte bună pentru generarea mărilor și oceanelor.

Ca o extindere a primei metode, putem să alegem anumite văi de pe hartă ca să le transformăm în lacuri, și pentru acea regiune vom calcula un alt nivel al apei.

O abordare mai complexă, și care dă rezultate mult mai realistice, este să folosim simularea pe care am făcut-o pentru eroziunea hidraulică pentru a depista zonele în care putem plasa râuri și lacuri.

b. Generarea rețelei stradale

După ce avem generat un relief pe care să construim orașul, următorul pas este generarea rețelei stradale.

Aceasta, în general, se reprezintă în memorie ca un graf, unde fiecare nod are asociat un punct pe harta înălțimilor. În practică sunt folosite cel puțin 2 grafuri pentru a reprezenta aceeași rețea stradală:

- *Graful articulațiilor:* fiecare segment de drum drept este reprezentat ca un arc în acest graf. Nodurile reprezintă intersecțiile și punctele de articulație.
- *Graful intersecțiilor*: acesta conține doar intersecțiile și legăturile dintre acestea, și este folosit pentru a micșora numărul de noduri, și a optimiza algoritmii de căutare a drumurilor.

Pentru a genera o rețea stradală există mai multe abordări posibile. În articolul scris de Kelly și McCabe [14], aceștia analizează câteva metode care pot fi folosite pentru acest scop.

Rețea grilă
Ideea de bază este că orașul este sub forma unei grile, în care celulele sunt spații în care pot fi plasate clădiri, și muchiile sunt străzi. Ca îmbunătățire, unele celule pot fi unite, sau unele muchii pot fi ușor deformate. Totuși, acestea arată departe față de un oraș real, lipsește diversitatea existentă în orașele reale, orașele generate astfel arată foarte repetitiv.

Sisteme Lindenmayer
O abordare este descrisă de Parish și Müller în [15], și folosește **sisteme Lindenmayer**: pentru fiecare segment de drum deja generat, încerc întâi să determin succesorul ideal, conform sistemului Lindenmayer. La următorul pas încerc să determin în ce măsură soluția propusă mă ajută să ating scopurile globale, și apoi verific dacă soluția propusă respectă constrângerile locale. Dacă soluția respectă aceste condiții, ea e acceptată.

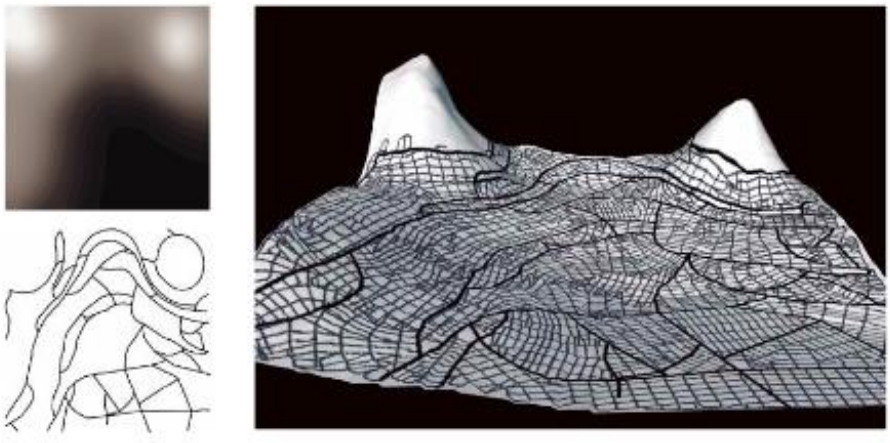


Fig. 27: Rețea stradală generată folosind un sistem Lindenmayer

Scopurile globale propuse de autori sunt: toate zonele mai dens populate să fie conectate (harta zonelor populate este generată in prealabil), apoi rețeaua străzilor să respecte anumite forme prestabilite (cum ar fi Manhattan, sau radial). Constrângerile locale sunt: diferența de nivel (un drum nu poate avea o înclinație prea mare), obstacole naturale (cum ar fi apa).

În Fig. 27 putem observa un exemplu de rețea stradală generată folosind sistem propus. În imaginea din stânga sus este harta înălțimilor folosită pentru generarea străzilor, și in stânga jos este o hartă cu străzile principale ce leagă zonele mai populate.

Agenți

O abordare a problemei generării rețelei de drumuri care folosește agenți este cea propusă de Watson în [16]. Aceasta folosește o mulțime de agenți care pot modela anumite entități, cum ar fi dezvoltatori, autorități ce planifică, și constructori de drumuri. Sistemul prezentat nu doar generează orașe, ci este capabil să simuleze și dezvoltarea lor în timp.

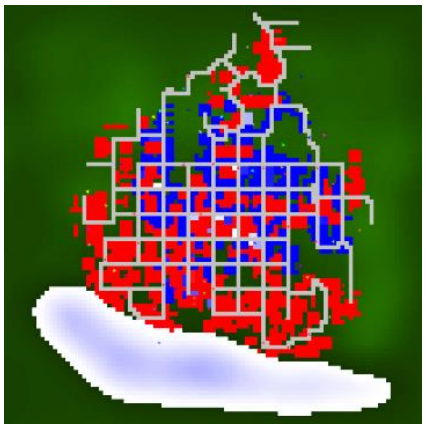


Fig. 28: Rețea a drumurilor generară folosind agenți

Segmentele de drum sunt create de două tipuri de agenți: *extenderi* care parcurg harta în apropierea zonelor dezvoltate, pentru a căuta zone ce nu sunt deservite de drumuri. *Conectorii* parcurg drumurile deja generate, măsurând distanțele dintre anumite puncte. Dacă distanța e prea mare, aceștia propun un segment de drum mai scurt între aceste puncte.

Un avantaj al acestei abordări este că diferiți agenți pot urmări diferite tipuri de aranjamente ale drumurilor (radial, organic, Manhattan), și astfel se pot genera orașe mult mai apropiate de realitate. Dezavantajul, însă, constă în complexitatea ridicată a algoritmului.

Generare bazată pe șabloane

O altă abordare a problemei poate fi generarea bazată pe șabloane, prezentată în [17]. Algoritmul folosește o combinație de șabloane (de exemplu, Manhattan, radial etc) și diagrame Voronoi pentru a genera procedural drumurile. Algoritmul de generare a unei diagrame

Voronoi primește ca date de intrare o mulțime de puncte numite *semințe*, și acesta împarte planul în poligoane astfel încât fiecare punct dintr-un poligon să aibă cea mai apropiată sămânță în același poligon. Algoritmul descris în articol alege puncte aleatorii ca semințe, având o densitate proporțională cu densitatea populației în acea zonă. Apoi, algoritmul împarte harta în poligoane folosind diagrame Voronoi, și astfel se obține rețeaua stradală.

Fig. 29: Rețele stradale generate folosind șabloane

În *Fig. 29* se pot observa modele de orașe generate folosind această metodă. În ordinea acelor de ceasornic, acestea sunt: diagrame Voronoi, șablon radial, șabloane combinate, șablonul Manhattan.

Avantajul acestei abordări este simplitatea algoritmului, dar ca dezavantaj, orașele generate nu sunt realistice.

c. Alocarea spațiilor pentru clădiri

Metoda împărțirii în poligoane

O metodă pentru a aloca spații pentru clădiri este descrisă în articolul lui Parish și Müller [15]. După ce am generat rețeaua stradală, folosim graful drumurilor pentru a determina poligoanele formate de acestea. Mai departe, „micșorăm” poligoanele pentru a lăsa spațiu pentru drumuri. Următorul pas este ca să „tăiem” în mod recursiv fiecare poligon obținut, adăugând muchii care să fie aproximativ paralele cu cele existente, până dimensiunea poligoanelor este acceptabilă. În final, eliminăm din toate poligoanele pe cele care nu sunt adiacente cel puțin unui drum, sau sunt prea mici.

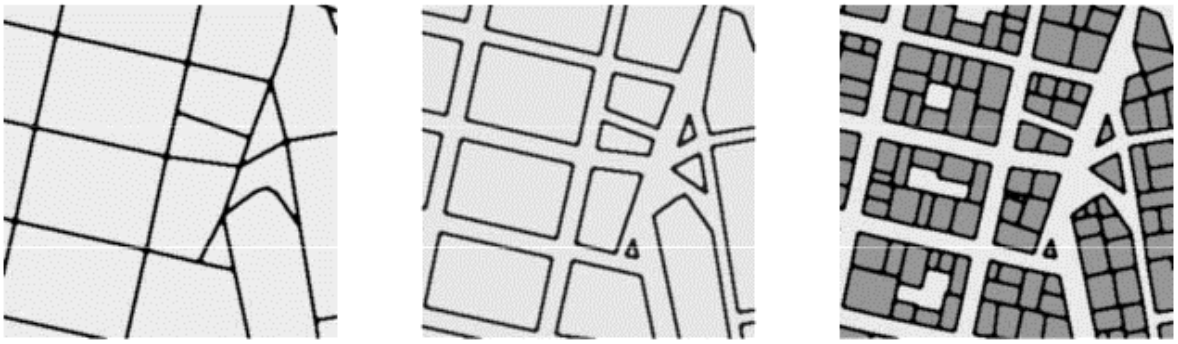


Fig. 30: Metoda împărțirii în poligoane. La primul pas, extragem poligoanele din graful rețelei stradale, apoi reducem dimensiunea acestora pentru a lăsa spațiu pentru drumuri, și în final tăiem poligoanele.

Dificultatea acestei metode constă în efectuarea primilor doi pași: determinarea poligoanelor, și micșorarea acestora. Un algoritm pentru determinarea poligoanelor într-un graf este prezentat în articolul [18] de X.Y.Jiang, iar un algoritm pentru micșorarea poligonului este descris în articolul [19] de Ron Wein.

Metoda grilei în jurul drumului

O altă metodă de alocare a spațiilor ce este folosită în jocul *Cities: Skyline* este construirea unei grile în jurul drumului. Algoritmul funcționează astfel: pentru fiecare segment de drum, încercăm să adăugăm câte o celulă; dacă celula pe care încercăm să o adăugăm intersectează un alt segment de drum sau o altă celulă, atunci nu o mai adăugăm.

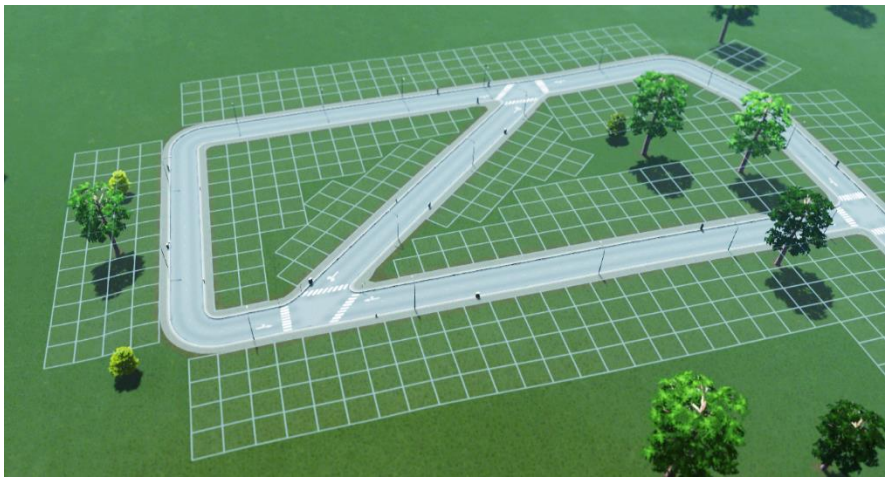


Fig. 31: Alocarea spațiilor pentru clădiri în Cities: Skyline. O clădire poate ocupa mai multe celule adiacente.

d. Generarea geometriei clădirilor

Ținând cont de multitudinea de stiluri arhitecturale existente, implementarea unui sistem care să genereze procedural clădirile este o provocare. Pentru generarea clădirilor există mai multe abordări posibile.

Primitive geometrice

O abordare simplă este abordată în [20]. Algoritmul de generare împarte clădirea în mai multe grupuri de „etaje”, și le generează pe rând de sus în jos. Pentru fiecare astfel de grup, se generează câteva poligoane primitive care sunt apoi reunite. La următorul grup de etaje, se generează un alt poligon în aceeași manieră, și rezultatul este reunit cu poligonul obținut la grupul precedent. La sfârșit, pentru fiecare grup de etaje se extrudează poligonul obținut pentru a obține o formă 3D. Acest proces este exemplificat în *Fig. 32*.

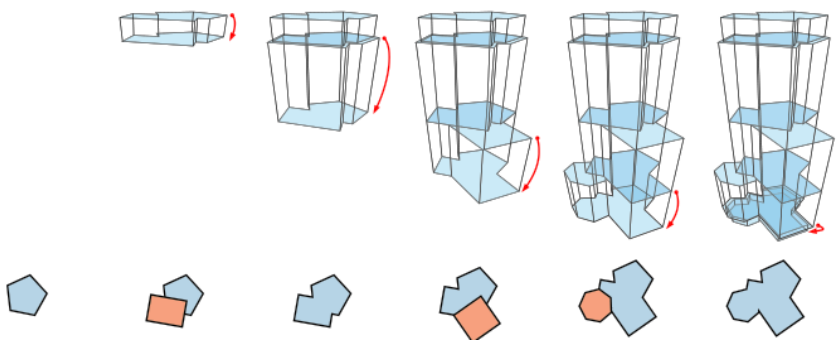


Fig. 32: Exemplu de clădire generată prin metoda primitivelor geometrice.

Sisteme Lindenmayer

Această metodă este descrisă mai detaliat în [15]. Ideea de bază este construirea unui sistem Lindenmayer care să aibă ca stare inițială un paralelipiped cu forma de bază al clădirii, și să adauge detalii iterativ. Avantajul acestei metode este că permite implementarea unui sistem *LOD (level of detail)*. Când clădirea este foarte depărtată de cameră, aceasta poate fi desenată cu o versiune simplificată a modelului 3D, ceea ce nu afectează vizibil imaginea, și îmbunătățește performanța. Cu cât obiectul este aproape, cu atât crește numărul de iterații executat al sistemului, și clădirea va avea mai mult detaliu.

Forme CGA

Formele CGA, inspirate din sistemele Lindenmayer, folosesc anumite gramatici pentru modelarea procedurală a formelor tridimensionale. Această metodă este descrisă detaliat în [21], și dintre metodele prezentate produce rezultatele cele mai bune.

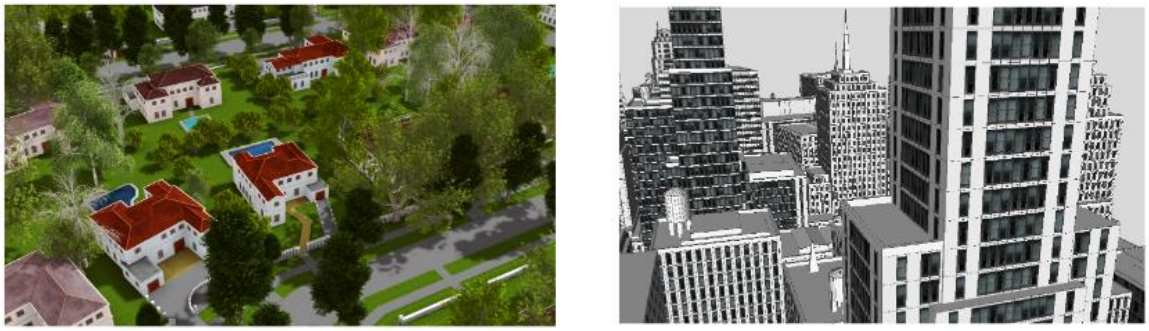


Fig. 33: Clădiri generate prin forme CGA

e. Finisarea

După ce am terminat generarea orașului, putem să adăugăm diverse detalii ce pot îmbunătăți aspectul orașelor.

Spre exemplu, **vegetația** dă un plus de viață orașelor; un mod de a adăuga vegetația este să generăm o hartă a densității copacilor, hartă ce ne va spune care este probabilitatea să generăm un copac într-un anumit punct. Mai departe, alegem puncte aleatoriu în care să plasăm vegetație, și vom plasa vegetație numai dacă în acel punct nu există nici un obstacol (obstacole pot fi masele de apă, drumurile, clădirile), și distanța de cel mai apropiat copac nu este mai mică decât o valoare prag.

Un alt exemplu de detaliu este adăugarea unor **clădiri speciale**, cum ar fi stadioane, monumente, spitale, parcuri etc; acestea pot face orașele să fie mult mai credibile.

4. Aplicație practică folosind generarea procedurală

În capitolul 3 am descris tehnicile ce pot fi folosite pentru generarea în mod procedural a unui oraș. În continuare, voi descrie modul în care am realizat un sistem generator de orașe.

Am ales să realizez acest sistem folosind motorul grafic *Unity3D*. Acest motor permite scrierea programelor în limbajul *C#* sau *JavaScript*, și oferă toate funcționalitățile necesare pentru a crea jocuri și aplicații grafice.

a. Arhitectura sistemului

Sistemul este organizat în mai multe straturi, conform *Fig. 34*. La bază stă modelul aplicației, care conține definirea entităților sistemului. Următorul strat, *Utils*, conține diverse proceduri și algoritmi folosite în întreaga aplicație.

Stratul *Business* conține logica aplicației, care constă în câteva clase ce se ocupă de partea administrativă (managementul resurselor) și generarea propriu-zisă a orașului.

Stratul *Unity Scripts* conține logica care este strâns legată de motorul *Unity3D*.

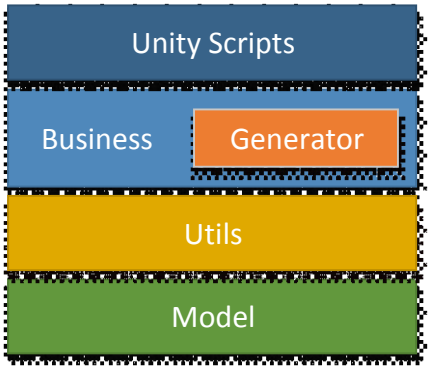
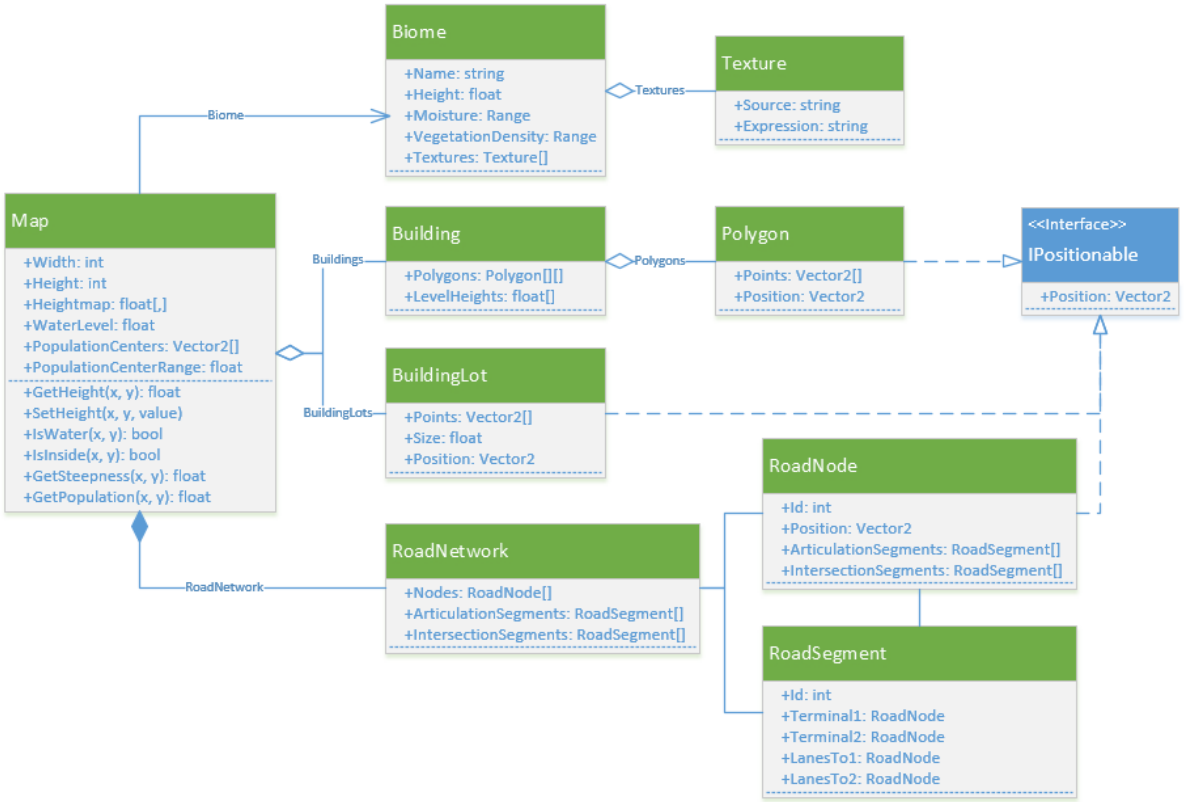


Fig. 34: Arhitectura stratificată

b. Modelul conceptual



Map este clasa principală, care conține tot ce ține de un oraș: harta înălțimilor, rețeaua drumurilor, geometria clădirilor.

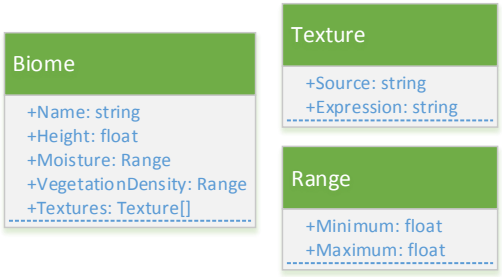
c. Climate

În articolul prezentat de Amit Patel [22], acesta se folosește de înălțimea și umiditatea reliefului pentru a asocia anumite climate fiecărei zone. Conform fiecărui climat, putem asocia un anumit tip de vegetație acelei zone, și aspectul zonelor va fi diferit.

În aplicația mea, am ales să implementez un sistem care funcționează în mod invers: acesta alege la început un climat, și relieful generat se va supune anumitor restricții specifice acelui climat. Spre exemplu, într-o zonă montană relieful va fi mult mai abrupt față de o zonă de câmpie.

Un climat este reprezentat în aplicație de clasa *Biome,* definită în modelul aplicației, și aceasta are ca atribute:

- *Name* reprezintă denumirea climatului;
- *Height* reprezintă înălțimea maximă a reliefului;
- *Moisture* reprezintă umiditatea, adică un interval ce reprezintă limitele procentajului din relief care să fie acoperit de apă;
- *VegetationDensity* este un interval ce specifică limitele de densitate permisă;
- *Textures* conține o listă de texturi ce vor fi aplicate terenului. Fiecare textură este specificată prin numele unui fișier de tip imagine ce conține textura, și o expresie matematică prin care se specifică regula după care va fi aplicată acea textură.



Climatele sunt definite și încărcate din fișiere XML. Clasa responsabilă pentru administrarea climatelor este *BiomeManager* din modulului *Business*.

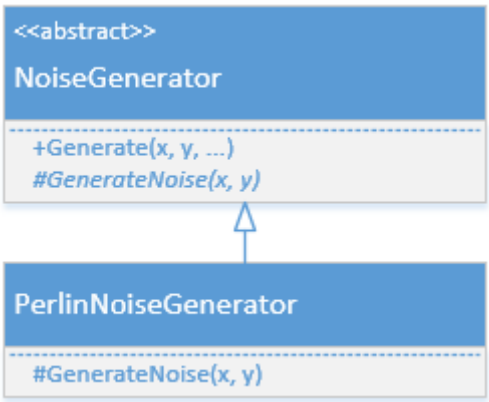
Iată un exemplu de climat stocat în formatul XML:

```
<?xml version="1.0" encoding="utf-8" ?>
<biome>
       <name>Mountain</name>
       <height>400</height>
       <moisture min=".1" max=".3"/>
       <vegetationDensity min=".5" max=".9" />
  <textures>
    <!-- Aplicată constant -->
    <texture src="mountain" expr="0.2" />
    <!-- Aplicată în zonele mai joase, unde relieful nu este abrupt -->
    <texture src="grass" expr="min(1 / (10 * steepness), (1 - ((height - waterLevel) /
(maxHeight - waterLevel)))^14)" />
    <!-- Aplicată pe vârfurile munților -->
    <texture src="snow" expr="2 * ((height - waterLevel) / (maxHeight - waterLevel))^2"
/>
  </textures>
</biome>
```

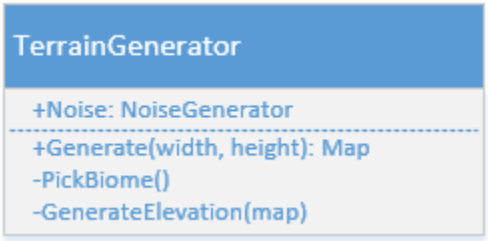
d. Generarea reliefului

La pasul precedent am ales un climat din cele existente; următorul pas este generarea în mod procedural a reliefului. Acesta se realizează folosind funcții de zgomot, după cum este descris în capitolul 3.

Clasa abstractă *NoiseGenerator*, construită după șablonul *template*, cuprinde toată logica necesară pentru filtrarea zgomotului și aplicarea octavelor, dar este exclusă implementarea propriu-zisă a funcției de zgomot. Clasa *PerlinNoiseGenerator* implementează funcția de zgomot dezvoltată de *Perlin* (funcție descrisă în capitolul 2).



Generarea reliefului este realizată în clasa *TerrainGenerator*: funcția *Generate* primește ca date de intrare dimensiunile hărții, și construiește un obiect de tipul *Map*, generează relieful și alege nivelul apei.



Texturarea reliefului

Implicit, relieful seamănă mai degrabă cu o hârtie mototolită decât cu un peisaj din lumea reală. Problema este că lipsesc complet texturile: relieful are forma unui relief, dar nu și aspectul unuia. Având în vedere faptul că nu putem aplica o singură textură pentru întreaga hartă, problema crește în complexitate: munții sunt făcuți din piatră, pământul este acoperit cu iarbă, vârful munților este acoperit cu zăpadă.

Soluția [23] este să folosim un sistem de reguli pentru fiecare tip de textură, reguli ce vor indica în ce măsură să fie aplicată textura pe diferite zone ale hărții.

Iată câteva exemple de astfel de reguli:

- Zăpada: poate apărea doar pe vârful munților, la înălțimi foarte mari, și e mare probabilitatea să fie mai proeminentă pe o anumită parte a munților
- Nisipul: poate apărea la nivele foarte apropiate de nivelul apei
- Iarba: crește pe teren relativ neted, nu va crește în pante mari

Pe baza regulilor se poate dezvolta o formulă care să returneze valori în intervalul $[0, 1]$, unde 0 înseamnă că textura nu va fi aplicată deloc în acea zonă, și 1 înseamnă că aceasta va predomina.

În sistemul de generare a orașelor prezentat, texturile și regulile specifice acestora pot fi specificate în fișierele de configurare XML ale climatelor. Evaluarea regulilor se face în clasa *TerrainGeneratorScript*, aflată în modulul *UnityScripts*.

e. Generarea hărții populației

Următorul pas pentru generarea orașului este realizarea unei hărți a populației, hartă ce va fi folosită mai departe pentru generarea drumurilor și a clădirilor.

Aici am avut de ales între două variante:

1. Să generez o hartă a populației similară cu harta înălțimilor, folosind zgomotul Perlin. Această metodă funcționează bine, însă necesită mai multe resurse computaționale și de memorie.
2. Să aleg aleatoriu câteva puncte care să fie centrele populate ale orașului. Această metodă nu oferă rezultate la fel de bune, dar necesită mult mai puține resurse.

Dintre cele două metode, am ales să implementez a doua metodă. În clasa *PopulationCentersGenerator* sunt alese centrele populate ale orașului; întâi este ales numărul de astfel de puncte, și următoarea regulă dă rezultate suficient de bune în practică:

$$nmax = 16 * \sqrt{mp}$$

$$n = random(\frac{nmax}{2}, nmax)$$

Unde *mp* este (impropriu spus) numărul de megapixeli al hărții, și se calculează:

$$mp = \frac{mapWidth * mapHeight}{1024^2}$$

Apoi, generez *n* puncte, și le aleg astfel încât acestea să nu fie prea aproape de marginea hărții (prea aproape însemnând la o distanță mai mică decât $\frac{1}{8}$ din dimensiunea hărții de marginile acesteia), și să nu fie în apă.

O altă variabilă aleasă de generator este răspândirea influenței unui centru de populație, și experimentând cu diferite dimensiuni ale hărții am ajuns la formula:

$$PopulationCenterRange = 31 * mp + 496.66$$

În *Fig. 35* se pot observa centrele populate și zonele influențate de acestea colorate cu mov.

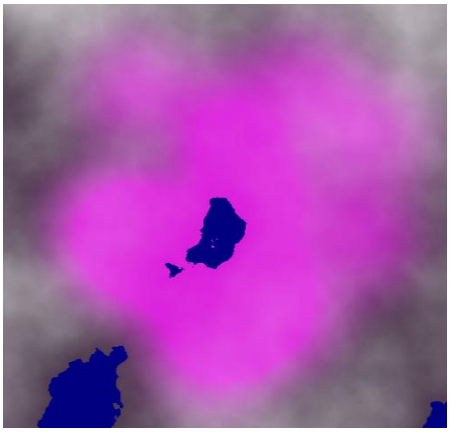
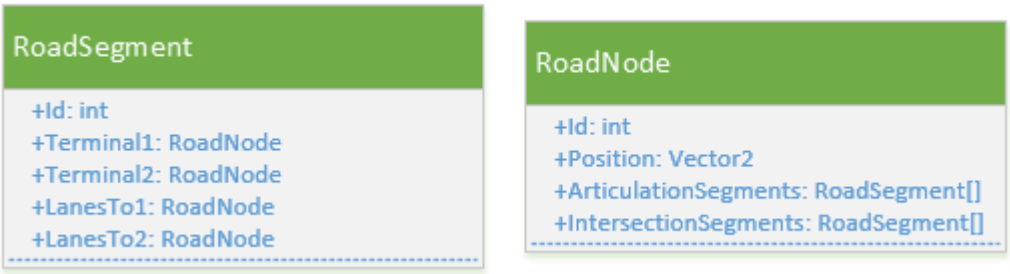


Fig. 35: Influența centrelor de populație

f. Generarea rețelei stradale

Rețeaua stradală este reprezentată prin două grafuri, un graf al articulațiilor și un graf al intersecțiilor. Nodurile sunt reprezentate de clasa *RoadNode* și au atașate informații despre poziția nodului pe hartă și despre segmentele de drum adiacente; segmentele de drum sunt reprezentate de clasa *RoadSegment*, și acestea au atașate informații despre nodurile adiacente și despre numărul de benzi.



Pentru generarea rețelei stradale am folosit o variantă a algoritmului descris de Parish și Müller [15] (prezentată în capitolul 3), variantă elaborată în articolul [24].

Iată algoritmul descris în pseudocod:

```
Algoritm GenerezăRețeaStradală este:
        @ Construiește coada cu priorități queue
        @ Generează segmentul rădăcină
        i ← 0
        CâtTimp i < nrMaxIterații și !queue.empty() execută:
                segment ← queue.pop()
                Dacă VerificăConstrângeriLocale(segment) atunci:
                        @ adaug segmentul în rețeaua de drumuri
                        PentruFiecare segmentNou din ScopuriGlobale(segment):
                                segmentNou.Time ← segmentNou.Time  + segment.Time + 1
                                queue.add(segmentNou)
                        SfPentru
                SfDacă
        SfCâtTimp
SfAlgoritm
```

Algoritmul menține o coadă cu priorități de unde extrage segmentul propus cu *timpul* minim. *Timpul* este o proprietate atașată segmentelor propuse care indică atât momentul cât și prioritatea pe care o are un segment de drum propus. Acesta este util pentru ca algoritmul să se extindă pe toată harta, și să nu rămână blocat într-o zonă.

Constrângerile verificate de funcția *VerificăConstrângeriLocale* sunt următoarele:

1. Segmentul generat nu depășește un obstacol: obstacole pot fi marginile hărții sau masele de apă;
2. Înclinarea drumului nu este prea mare: o înclinare acceptabilă este sub 30 grade.
3. În cazul în care un nod adiacent formează o intersecție, numărul de drumuri care se intersectează nu depășește o anumită limită.
4. Dacă segmentul intersectează un alt segment, unghiul dintre cele două segmente nu este prea mic.

Funcția verifică și cazurile speciale prezentate în articolul lui Parish și Müller [15], și poate propune modificarea segmentului generat:

1. Segmentul generat intersectează un alt segment; segmentul generat este tăiat în punctul de intersecție și se păstrează doar prima bucată.
2. Segmentul generat se termină foarte aproape de o intersecție; segmentul va fi modificat astfel încât să ajungă până în intersecție.
3. Segmentul generat se termină foarte aproape de un alt segment de drum; acesta va fi prelungit astfel încât să se intersecteze cu segmentul existent.

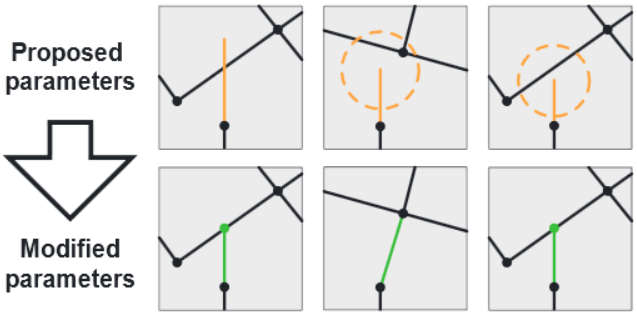


Fig. 36: Reguli pentru modificarea segmentelor generate, prezentate în articolul lui Parish și Muller [15]

Funcția *ScopuriGlobale* propune o listă de segmente ce pornesc din capătul segmentului curent (dat ca parametru). Vor fi propuse segmente conform următoarele reguli:

Dacă segmentul precedent este un segment de autostradă:

1. Aleg o direcție apropiată de direcția segmentului precedent; mă deplasez fie în acea direcție, fie drept înainte, astfel încât să mă deplasez spre zona mai populată.
2. Dacă densitatea populației este peste o anumită valoare limită, există o probabilitate ca autostrada să se bifurce, fie la dreapta, fie la stânga cu un alt segment de autostradă.
3. Dacă densitatea populației este peste o anumită limită, există o probabilitate ca autostrada să se bifurce în segmente de drum obișnuit.

Dacă segmentul precedent este un segment de drum obișnuit:

1. Pot să merg înainte cu o anumită probabilitate ce depinde de densitatea populației.
2. Drumul se poate bifurca la stânga sau la dreapta în funcție de densitatea populației.

Generarea geometriei străzilor

Clasa *RoadMeshGenerator* din modulul *UnityScripts* este responsabilă pentru generarea geometriei drumurilor. La generarea geometriei avem două cazuri care vor fi tratate separat: generarea intersecțiilor și generarea segmentelor de drum.

Pentru generarea geometriei intersecțiilor, trebuie ca pentru fiecare intersecție să calculăm coordonatele punctelor de colț: colțurile străzilor, respectiv colțurile trotuarelor; ordonăm toate segmentele adiacente unui nod în ordine trigonometrică, și iterăm fiecare pereche de segmente alăturate. Dacă luăm în considerare paralelogramul format din marginea străzilor și cele două segmente (care pot fi considerate axele drumurilor), putem obține poziția colțului străzii din suma a doi vectori, laturile paralelogramului. Cunoaștem lățimea segmentelor de drum (înălțimile paralelogramului), și putem determina ușor unghiul dintre acestea (folosind produsul scalar). Folosind cele două, putem calcula lungimea laturilor paralelogramului. Înmulțind vectorii de direcție ai segmentelor de drum cu lungimile, obținem vectorii respectivi laturilor paralelogramului.

Algoritmul este exemplificat în *Fig. 38*; în acest caz, coordonatele vectorului *streetCornerAB* se pot calcula adunând vectorii:

$$streetCornerAB = N + \overrightarrow{N\,F} + \overrightarrow{N\;streetCornerAB}$$

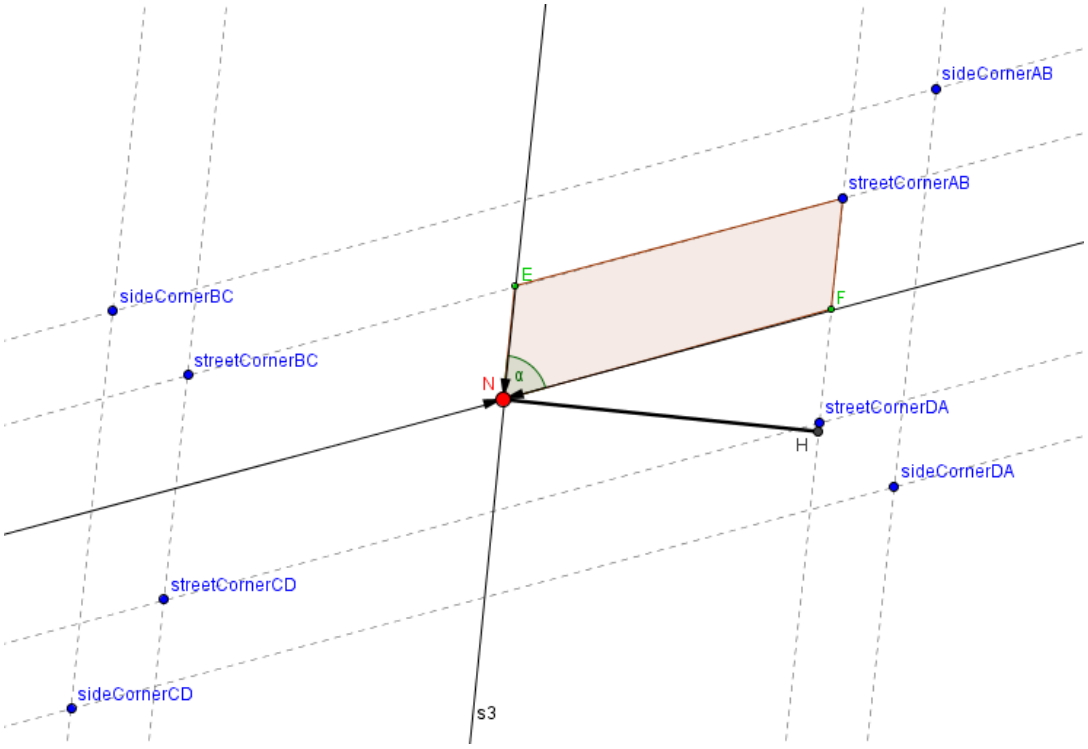


Fig. 38: Paralelogramul obținut și înălțimea din N

La următorul pas după obținerea coordonatelor colțurilor, trebuie să trasăm perpendicularele din fiecare colț de trotuar pe segmentele de drum alăturate. Pentru fiecare segment vom obține două drepte perpendiculare: dintre acestea, o eliminăm pe cea mai apropiată de nod, și vom

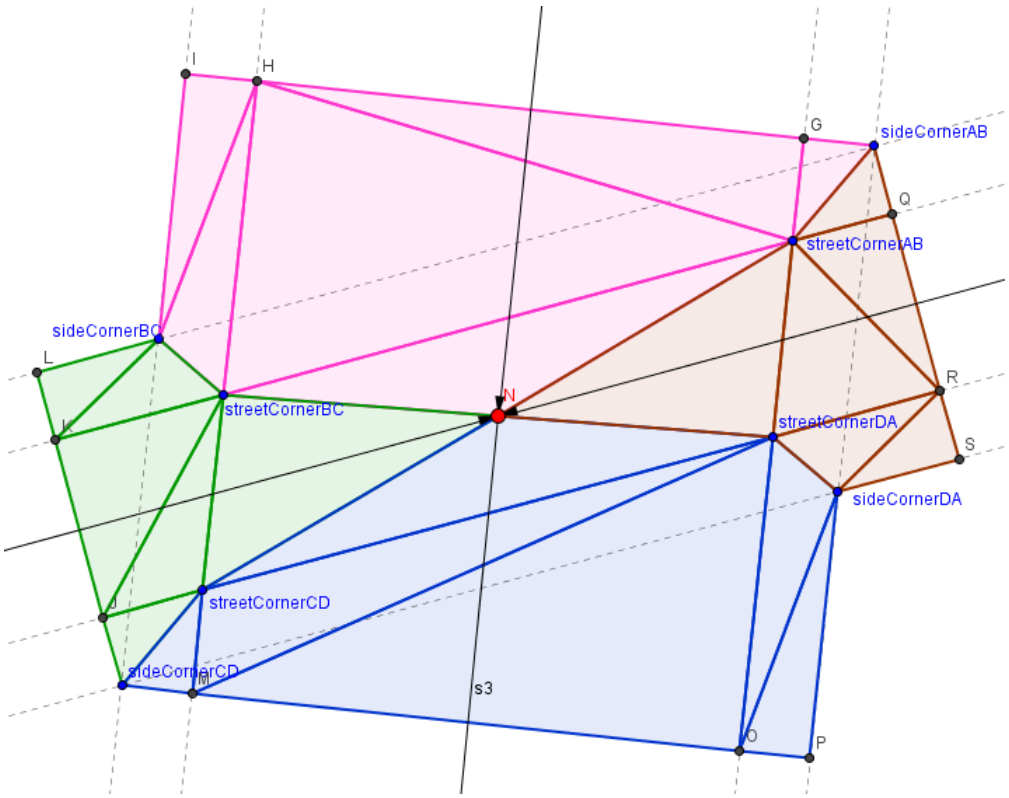


Fig. 37: Triunghiurile obținute după trasarea perpendicularelor din colțurile trotuarelor.

rămâne astfel cu câte o dreaptă perpendiculară pe fiecare segment. Calculăm punctele de intersecție între dreaptă și marginile trotuarelor, respectiv între dreaptă și marginile străzii.

Folosind aceste puncte putem obține geometria intersecției (triunghiurile din care sunt formate modelele 3D), după cum este ilustrat și în *Fig. 37*: găsim o regulă pe care o putem aplica pentru fiecare segment de drum (de exemplu: porțiunea de trotuar dintre dreapta perpendiculară și colțuri, porțiunea de drum dintre perpendiculară și colțuri etc).

După ce am generat intersecțiile corespunzătoare ambelor capete unui segment, putem genera și acel segment de drum. Pentru generarea segmentului ne vom folosi de punctele de intersecție cu perpendicularele de la calculul intersecțiilor.

g. Generarea clădirilor

Clădirile sunt reprezentate de clasa *Building*, și au ca proprietăți o listă cu poligoane corespunzătoare grupurilor de nivele, și o listă cu înălțimile corespunzătoare fiecărui grup de etaje. Clasa responsabilă pentru generarea clădirilor este *BuildingGenerator*.

Alocarea spațiilor pentru clădiri

Pentru alocarea spațiilor pentru clădiri am folosit o metodă inspirată din cea folosită în jocul *Cities: Skyline*. Diferența este că algoritmul nu generează o grilă, ci generează pătrate de diferite dimensiuni în care se vor construi clădirile: pentru fiecare segment de drum, vom plasa pătrate de diferite dimensiuni pe cele două părți ale drumului. Atunci când plasăm pătrate, verificăm că acestea nu intersectează alte pătrate sau segmente de drum. În *Fig. 39* este ilustrată această metodă de alocare a spațiilor.

Fig. 39: Spațiile generate pentru clădiri

Generarea clădirilor

Clădirile au fost generate după metoda reunirii poligoanelor, metodă descrisă mai pe larg în capitolul 3: împărțim clădirea în grupuri de etaje, și pentru fiecare grup de etaje (începând de la grupul cel mai de sus) generăm aleatoriu câteva poligoane primitive. După generarea primitivelor, facem reuniunea acestora cu toate poligoanele generate pentru nivelele superioare. De asemenea, pentru fiecare grup de etaje vom alege aleatoriu o înălțime, înălțime ce depinde și de populația zonei (în zonele mai dens populate, clădirile vor fi mai înalte).

Următorul pas după ce am generat clădirile este să generăm geometria acestora. Clasa responsabilă pentru generarea geometriei clădirilor este *BuildingMeshGenerator* și face parte din modulul *UnityScripts*. Pentru generarea geometriei, vom aplica o tehnică numită *extrudare* fiecărui poligon generat anterior: prin această tehnică construim prisme drepte ce au ca bază poligoanele generate. Punând cap la cap prismele obținute din poligoanele fiecărei clădiri vom obține geometria completă a acestora.

Fig. 40: Clădiri obținute prin metoda reunirii poligoanelor

Ceea ce a mai rămas de făcut pentru clădiri este aplicarea unor texturi. După finalizarea acestui pas, iată orașul obținut:

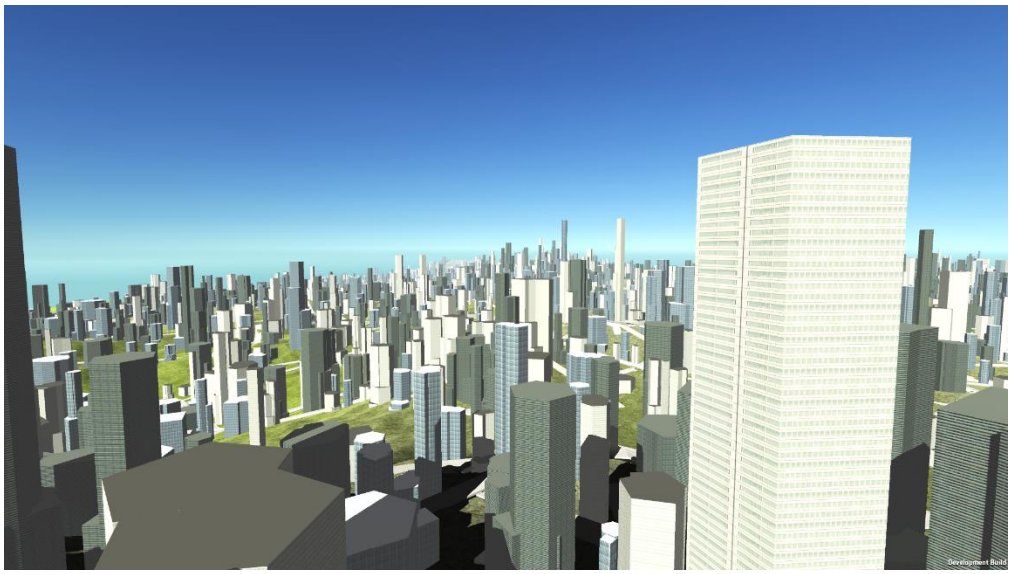


Fig. 41: Clădirile generate după aplicarea texturilor

5. Exemple de aplicații comerciale

a. În industria filmelor

Industria filmelor este un mare consumator al imaginilor generate pe calculator. Folosind generarea procedurală, costurile și timpul necesar pentru a crea peisaje și orașe apropiate de lumea reală pot fi mult reduse. Există și aplicații comerciale care sunt capabile să genereze procedural peisaje naturale și orașe. Aceste aplicații pot fi folosite atât în industria filmelor, cât și în dezvoltarea jocurilor.

Terragen

Terragen este o aplicație comercială care realizează peisaje într-un mod procedural. Filme ce au fost realizate folosind această aplicație includ *Ender's Game (2013), Tron Legacy (2010)* și *Man of Steel (2013).*

Fig. 42: Imagini generate cu Terragen

CityEngine

CityEngine este o aplicație comercială folosită pentru generarea procedurală a orașelor. Aceasta e capabilă să genereze atât rețeaua stradală cât și clădiri în multe stiluri arhitecturale, inclusiv orașe arhaice, cât și orașe imaginare futuriste.

Fig. 43: Orașe generate cu CityEngine

SpeedTree

SpeedTree este o unealtă folosită atât în filme cât și în dezvoltarea jocurilor ce generează vegetație în mod procedural. Aplicația are un număr foarte mare de parametrii ce pot fi modificați, și generarea se produce pas cu pas, asistată de utilizator. Astfel, aplicația poate fi folosită pentru a produce o varietate imensă de vegetație.

Fig. 44: Plante generate folosind SpeedTree

b. În jocuri

În jocuri, generarea procedurală poate fi folosită atât în momentul dezvoltării jocului (unde pot fi folosite aplicații precum cele prezentate mai sus), cât și la timpul rulării jocului. La începuturile industriei, multe jocuri alegeau generarea procedurală din cauza limitelor impuse de hardware.

Rogue (1980)

Unul din primele jocuri care au folosit generarea procedurală este *Rogue*. Acesta este un joc de tip RPG (role-playing game) în care lumea virtuală constă în catacombe ce sunt generate procedural. Jocul a fost foarte popular la apariție, stârnind apariția unui gen întreg de jocuri, *jocurile-ca-rogue (rogue-like).*

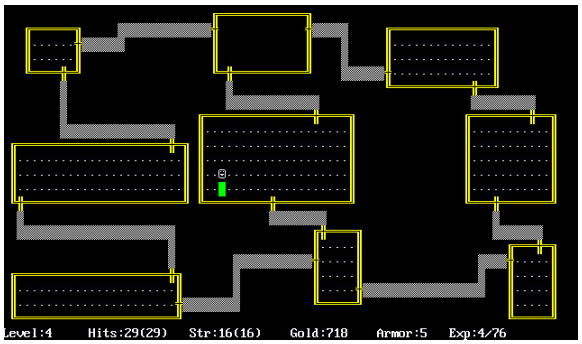


Fig. 46:Rogue

Fig. 45: Diablo

Diablo (1996)

Diablo este tot un joc de tip RPG în care lumea era generată în mod procedural. Față de *Rogue*, unde grafica constă din caractere ASCII, *Diablo* folosește modul de vizualizare izometric. Jocul a fost foarte apreciat pentru *replay-value*, și încă rămâne unul din cele mai apreciate jocuri.

Spore (2008)

Fig. 47: Spore

Spore este un joc care acoperă mai multe genuri: la început, jucătorul poate controla o ființă ce seamănă cu o celulă, ce evoluează în timp. Jucătorul poate alege modul în care să evolueze și să crească ființa, precum și felul în care aceasta arată. În al doilea stadiu al jocului, are loc trecerea din ființă acvatică în ființă terestră. Întreaga lume terestră este generată procedural, cât și celelalte creaturi pe care utilizatorul le va întâlni. Utilizatorul poate alege să se „împrietenească” cu alte creaturi, sau să fie dușmani. La al treilea stagiu, ființa e suficient de evoluată ca să formeze o societate tribală, și jocul devine un joc de strategie. În următorul stadiu, tehnologia evoluează și ființa devine specia dominantă pe planetă, și jocul devine un joc de strategie modern. Ultimul stadiu constă în explorarea spațiului.

Jocul a fost apreciat foarte mult pentru originalitatea sa, cât și pentru generarea procedurală. Totuși a fost criticat pentru ușurătatea cu care a abordat anumite elemente de joc.

Fig. 48: Minecraft

Minecraft (2011)

În *Minecraft*, lumea virtuală este formată din cubulețe mici numite *voxeli*. Aceste cubulețe pot fi de mai multe tipuri, cum ar fi *pământ*, *nisip*, *piatră* etc, fiecare având anumite proprietăți. Întreaga lume este formată din astfel de cubulețe, și este generată în mod procedural. Jocul permite jucătorilor să meargă prin lume oricât de departe, lumea generată este teoretic infinită.

Există mai multe moduri de joc, cum ar fi *supraviețuire*, unde jucătorii trebuie să supraviețuiască într-un mediu ostil, folosindu-se de resursele naturale. În modul *creativ*, jucătorii pot să construiască orice, având la dispoziție orice tip de cubulețe.

6. Concluzii

Având în vedere complexitatea mereu în creștere a jocurilor, generarea procedurală oferă o alternativă mult mai puțin costisitoare construirii elementelor de conținut pentru jocuri, față de metoda tradițională de modelare manuală. Tehnicile dezvoltate pentru generarea procedurală sunt capabile să producă rezultate foarte realistice. Sisteme comerciale precum *SpeedTree* sau *Terragen* sunt folosite foarte mult pentru dezvoltarea jocurilor, și în viitor vom vedea tot mai multe jocuri ce folosesc generarea procedurală.

În opinia mea, modul clasic de modelare manuală a obiectelor va fi folosit tot mai puțin în viitor, și vom vedea tot mai multe sisteme procedurale. Atunci când nu vom mai putea deosebi lumea reală de cea virtuală, majoritatea elementelor de conținut în lumile virtuale vor fi generate în mod procedural. Spre exemplu, nu va mai fi considerat acceptabil ca să vedem în jocuri doi copaci identici; singurul mod în care putem crea o infinitate de copaci care să fie unici este prin generarea procedurală.

Bibliografie

[1] The Economist, „Why video games are so expensive to develop," September 2014. [Interactiv]. Available: http://www.economist.com/blogs/economist-explains/2014/09/economist-explains-15.

[2] Wikipedia, „3D Modeling," [Interactiv]. Available: https://en.wikipedia.org/wiki/3D_modeling.

[3] J. H. Clark, „Hierarchical Geometric Models for Visible Surface Algorithms," *Communications of the ACM,* vol. 19, nr. 10, pp. 547-554, October 1976.

[4] Wikipedia, „Heightmap," [Interactiv]. Available: https://en.wikipedia.org/wiki/Heightmap.

[5] Wikipedia, „Noise," [Interactiv]. Available: https://en.wikipedia.org/wiki/Noise.

[6] K. Perlin, „Making noise," 1982. [Interactiv]. Available: www.noisemachine.com/talk1/.

[7] K. Perlin, „Improving Noise," *ACM Transactions on Graphics,* vol. 21, nr. 3, pp. 681-682, 2002.

[8] A. Biagioli, „Understanding Perlin Noise," 09 August 2014. [Interactiv]. Available: https://flafla2.github.io/2014/08/09/perlinnoise.html.

[9] P. Prusinkiewicz, A. Lindenmayer și J. Hanan, „Developmental Models of Herbaceous Plants for Computer Imagery Purposes," *Computer Graphics ,* vol. 22, nr. 4, pp. 141-150, 1988.

[10] Wikipedia, „L-system," [Interactiv]. Available: https://en.wikipedia.org/wiki/L-system.

[11] R. Finkel și J. Bentley, „Quad Trees: A Data Structure for Retrieval on Composite Keys," *Acta Informatica,* vol. 4, nr. 1, p. 1–9, 1974.

[12] H. Elias, „Perlin Noise," [Interactiv]. Available: http://freespace.virgin.net/hugo.elias/models/m_perlin.htm.

[13] F. K. Musgrave, C. E. Kolb și R. S. Mace, „The Synthesis and Rendering of Eroded Fractal Terrains," *ACM SIGGRAPH Computer Graphics,* vol. 23, nr. 3, pp. 41-50, 1989.

[14] G. Kelly and H. McCabe, "A Survey of Procedural Techniques for City Generation," *ITB Journal,* no. 14, 2006.

[15] Y. I. H. Parish și P. Müller, „Procedural Modeling of Cities,” în *SIGGRAPH '01 Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, New York, 2001.

[16] T. Lechner, B. Watson, U. Wilensky and M. Felsen, "Procedural City Modeling," 2003.

[17] J. Sun, X. Yu, G. Baciu și M. Green, „Template-based generation of road networks for virtual city modeling,” în *VRST '02 Proceedings of the ACM symposium on Virtual reality software and technology*, New York, 2002.

[18] X. Jiang și H. Bunke, „An optimal algorithm for extracting the regions of a plane graph,” *Pattern Recognition Letters*, vol. 14, nr. 7, pp. 553-558, 1993.

[19] R. Wein, „Exact and approximate construction of offset polygons,” *Computer-Aided Design*, vol. 39 , nr. 6, pp. 518-527, 2007.

[20] S. Greuter, J. Parker, N. Stewart și G. Leach, „Real-time procedural generation of `pseudo infinite' cities,” în *GRAPHITE '03 Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia* , New York, 2003.

[21] P. Müller, P. Wonka, S. Haegler, A. Ulmer și L. V. Gool, „Procedural modeling of buildings,” în *SIGGRAPH '06 ACM SIGGRAPH 2006 Papers* , New York, 2006 .

[22] A. Patel, „Polygonal Map Generation for Games,” 4 September 2010. [Interactiv]. Available: http://www-cs-students.stanford.edu/~amitp/game-programming/polygon-map-generation/.

[23] A. Aitchison, „Procedural Terrain Splatmapping,” 14 November 2013. [Interactiv]. Available: https://alastaira.wordpress.com/2013/11/14/procedural-terrain-splatmapping/.

[24] tmwhere.com, „Procedural City Generation,” [Interactiv]. Available: http://www.tmwhere.com/city_generation.html.