UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

**FACULTATEA DE INFORMATICĂ**

LUCRARE DE LICENȚĂ

# Generare Procedurală Folosind Modele Markov

propusă de

**Alexandru Loghin**

**Sesiunea:** iulie, 2019

Coordonator științific

**Conf. Dr. Anca Vitcu**

UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

**FACULTATEA DE INFORMATICĂ**

# Generare Procedurală Folosind Modele Markov

**Alexandru Loghin**

**Sesiunea:** iulie, 2019

Coordonator științific

**Conf. Dr. Anca Vitcu**

Avizat,

Îndrumător lucrare de licență,

Conf. Dr. Anca Vitcu.

Data: ..... Semnătura: .....

## Declarație privind originalitatea conținutului lucrării de licență

Subsemnatul **Loghin Alexandru** domiciliat în **România, jud. Iași, mun. Iași, calea Buzăului, nr. 25, bl. A, et. 5, ap. 45**, născut la data de **01 ianuarie 2018**, identificat prin CNP **1234567891234**, absolvent al Facultății de informatică, **Facultatea de informatică** specializarea **informatică**, promoția 2019, declar pe propria răspundere cunoscând consecințele falsului în declarații în sensul art. 326 din Noul Cod Penal și dispozițiile Legii Educației Naționale nr. 1/2011 art. 143 al. 4 și 5 referitoare la plagiat, că lucrarea de licență cu titlul **Generare Procedurală Folosind Modele Markov** elaborată sub îndrumarea domnului **Conf. Dr. Anca Vitcu**, pe care urmează să o susțin în fața comisiei este originală, îmi aparține și îmi asum conținutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licență să fie verificată prin orice modalitate legală pentru confirmarea originalității, consimțind inclusiv la introducerea conținutului ei într-o bază de date în acest scop.

Am luat la cunoștință despre faptul că este interzisă comercializarea de lucrări științifice în vederea facilitării falsificării de către cumpărător a calității de autor al unei lucrări de licență, de diplomă sau de disertație și în acest sens, declar pe proprie răspundere că lucrarea de față nu a fost copiată ci reprezintă rodul cercetării pe care am întreprins-o.

Data: ..... Semnătura: .....

## Declarație de consimțământ

Prin prezenta declar că sunt de acord ca lucrarea de licență cu titlul **Generare Procedurală**
**Folosind Modele Markov**, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test, etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de informatică.

De asemenea, sunt de acord ca Facultatea de informatică de la Universitatea ”Alexandru-Ioan Cuza” din Iași, să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Absolvent **Alexandru Loghin**

Data: .....  Semnătura: .....

# Cuprins

# Motivație

Datorită pasiunii mele pentru *gamedev* și algoritmică am decis să realizez o modalitate de a genera procedural un mediu cât mai diversificat. Folosirea modelelor Markov a venit din necesitatea de a scăpa cumva de metoda clasică prin care se generează un șir de obiecte într-un mod aleator iar apoi se încearcă validarea acesteia după anumite constrângeri. Folosind acest model stocastic putem controla apariția unei anumite subsecvențe prin stabilirea unei anumite probabilități de a se produce așadar eliminând cu totul pasul de validare.

Scopul principal al proiectului a fost dezvoltarea unei librări ce ușurează procesul de generare. De asemenea a fost nevoie și de un sistem ce facilitează plasarea obiectelor în mediul 3D , modelul Markov fiind folosit doar ca și generator. Odată funcționale mi-am îndreptat atenția asupra interfeței grafice și a interacțiunii dintre jucător și joc.

Datorită motorului grafic folosit și a instrumentelor de care dispune, timpul necesar pentru crearea jocului a fost îmbunătățit substanțial. De asemena cu prilejul acestei aplicații am avut posibilitatea de a aprofunda și testa diferite tehnici de randare și post-procesare. Câteva dintre aceste elemente includ SSR [1], lumină volumetrică, reflexii în timp real, corectare de culoare și multe altele.

Odată cu finalizarea acestei aplicații , din cauza naturii sale *open source* , se va putea folosi librăria creată în cadrul acesteia pentru orice tip de proiect fără constrăngeri, fiind ușor adaptabilă pentru orice sarcină ce necesită generare procedurală, de la muzică până la construcția unui mediu virtual, librăria find ușor de folosit și axata pe eficiență.

[1] Screen Space Reflection

# Introducere

Având în vedere necesitatea creării unor medii cât mai versatile ce respectă anumite condiții sau restricții fizice s-au dezvoltat anumite tehnici pentru a facilita construcția automată și randomizată a oricărui element ce intră în componența unui joc , de la modele tridimensionale până la coloana sonoră.

Am decis așadar să încerc o abordare nouă , folosindu-mă de un model Markov ce a fost antrenat să respecte anumite reguli pentru a genera spațiul 3D. Scopul acestei lucrări este de a aduce o nouă perspectivă asupra generării procedurale și de a demonstra validitatea acestei abordări.

Tema lucrării s-a ivit din cauza necesității de adaptare rapidă a condițiilor folosite pentru generarea obiectelor din componența unui joc. Unul din avantajele acestei abordări este flexibilitatea oferită de modelul Markov fiind capabil să adapteze constrăngerile folosite la generare cu o simplă reantrenare a modelului.

Aplicația prezentă este construită folosind un motor grafic peste care am dezvoltat o librărie capabilă să reprezinte un model Markov și să îl antreneze. Folosindu-mă de această librărie , pot genera o secvență de obiecte ce încearcă să respecte fidel constrângerile folosite la antrenare.

Procesul de generare procedurală este compus din două etape. Generarea obiectelor și plasarea lor convenabil în spațiul virtual. Am încercat în special să decuplez cele două etape pentru o mai bună modularizare și o coeziune ridicată. Cele două sisteme lucrează independent unul față de celălalt , relația dintre cele două find una de agregare.

Cele două sisteme sunt puse în funcțiune pentru a genera în timp real mediul pentru jucător, mediu ce este împărțit în trei zone de interes , urbana , rurală și deșertică. De asemenea pentru a adauga complexitate se alternează între trei tipuri de platforme ce simulează un drum drept, un drum cu viraj la stănga sau un drum cu viraj la dreapta.

# Capitolul 1

# Aspecte teoretice

## 1.1 Preambul

În domeniul probabilităților un model Markov este folosit pentru a modela un sistem ce se schimbă într-un mod aleator. De obicei acesta ține cont doar de starea curentă și nu depinde de evenimentele aleatoare , acest lucru numindu-se și prorietatea Markov. Ulterior s-au dezvoltat modele ce au un așa numit ordin , extinzând astfel numărul de stări de care se ține cont în cadrul modelului.

De obicei există o separație clară între tipurile de modele Markov, criteriul folosit este disponibilitatea de a observa sau nu stările modelului. Așadar rezultă două mari categorii, sisteme cu stări complet observabile , din care fac parte lanțurile Markov și sisteme cu stări parțial observabile, cel mai cunoscut sistem fiind modelul Markov cu stări invizibile sau HMM [1]. Pe langă aceste modele prezentate, s-au dezvoltat și anumite derivate fiecare cu avantajele sale demonstrând astfel flexibilitatea și capacitatea de exprimare a acestor sisteme.

În mare parte această teză se axează în jurul parții discrete a acestor modele, numărul de stări/observații find finit numărabile și ușor de caracterizat dar există și o ramură ce se ocupă cu partea continuă a acestor modele, fiecare abordare având avantajele și dezavantajele ei.

Legat de partea practică, de-a lungul timpului aceste modele au fost folosite în foarte multe domenii de la bioinformatică până la lingvistică computațională. O aplicație foarte importantă a acestor modele este recunoașterea vocală, modelele Markov fiind standardul folosit în industrie pentru asistenții personali ca *Siri* și *Alexa*.

[1] Hidden Markov Model

## 1.2 Modele Markov cu stări invizibile

Acest model statistic este caracterizat în principal de faptul că stările interne sunt ascunse de un privitor exterior. Tot ce se poate observa în cadrul acestui model este emisia unor etichete sau obiecte $\{v_1, v_2, v_3, \ldots, v_n\}$ dintr-o mulțime notată pe scurt $V$. Acest lucru complică în general structura modelului și algortimii ce folosesc acest sistem.

Un avantaj direct este creșterea capacității de expresivitate, putând modela mai fidel datele, datorită eliminării necesității de a cunoaște toate stările evenimentului.

Pentru a caracteriza concret și complet acest model avem nevoie de un *5-uplu* $HMM = (\mathbf{S}, \mathbf{V}, \mathbf{A}, \mathbf{B}, \pi)$ ce desemnează astfel :

- $\mathbf{S} = \{S_1, S_2, \ldots, S_N\}$ mulțimea ce desemnează numărul de stări ascunse din model, avand un numar de $N$ elemente. Starea relativa la timpul $t$ se va nota ca si $q_t$.
- $\mathbf{V} = \{V_1, V_2, \ldots, V_M\}$ , cele $M$ etichete observabile.
- $\mathbf{A} = \{a_{ij}\}$ unde $a_{i,j} = P(q_{t+1} = S_j | q_t = S_i), 1 \leq i, j \leq N$ , reprezentând distribuția de probabilitea asociată tranzițiilor între stări.
- $\mathbf{B} = \{b_i(k)\}$ unde $b_i(k) = P(V_k \; la \; timpul \; t \; | q_t = S_i), 1 \leq i \leq N, 1 \leq k \leq M$, ce caracterizează distribuția de probabilitate asociată emiterii unui element din $V$ în starea $i$.
- $\pi = \{\pi_i\}$ unde $\pi_i = P(q_1 = S_i), 1 \leq i \leq N$, folosit inițial pentru a stabili prima stare din model și anume $q_1$.

În general modul de funcționare a acestui model poate fi descris foarte ușor cu o bucată de pseudocod:

```
1  t ← 1;
2  stare ← alege o stare S_i cu probabilitate π_i;
3  repeta la infinit
4          stare ← alege o noua stare S_j de tranzitie de la starea S_i cu probabilitate a_ij ∈ A;
5          afiseaza observatia V_k cu probabilitatea b_i(k) ∈ B;
6          t ← t + 1;
```

Uneori printre alte publicații de specialitate modelul poate fi descris prin specificarea parametrilor $N, M$ și cele trei distribuții de probabilitate notate astfel $\lambda = \{\mathbf{A}, \mathbf{B}, \pi\}$.

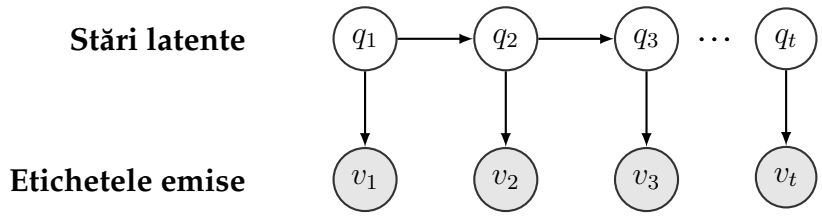


Figura 1.1: Structura modelului Markov cu stari invizibile

În cadrul aplicației mulțimea de etichete $V$ au fost reprezentă de elementele de tip *GameObject* ce urmează a fi instanțiate. Din motive de performanță și flexibilitate aplticația folosește în total un număr de sașe modele Markov cu stări invizibile a căror parametri au fost estimați să respecte anumite restricții.

Se poate face o distincție între cei șase generatori după scopul lor în cadrul jocului. Trei dintre ei sunt folosiți pentru a genera terenul pe care jucătorul navighează. Această decizie a fost facută din necesitatea de a putea controla numărul de platforme pentru fiecare din cei trei generatori, ce în fond reprezintă trei zone distincte, urbană, rurală și deșertică enumerate conform ordinii de apariție în joc.

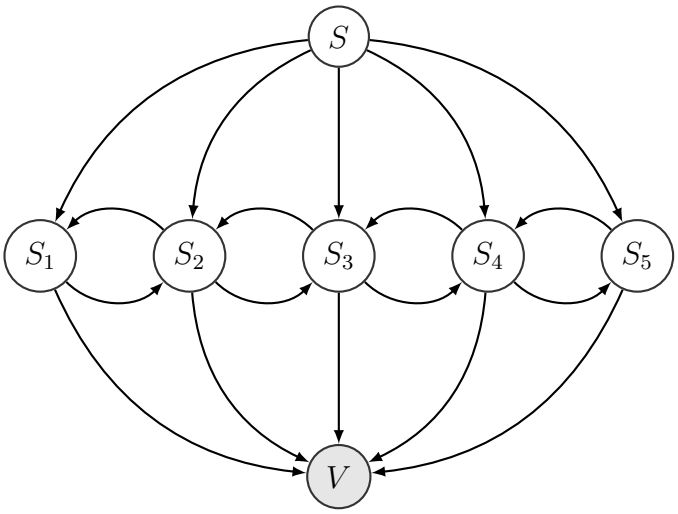


Figura 1.2: Structura unui generator din cele trei zone

Se observă din figură, că avem un nod de start $S$ ce ne conduce la una dintre cele cinci stări, iar fiecare stare are acces la $V$ mulțimea etichetelor în cazul nostru, cele trei tipuri de platformă *Left*,*Right* si *Straight*. Din model s-au omis probabilitățile de pe arce tocmai pentru a nu încărca desenul cât și pentru faptul că ele vor fi inferate ulterior de un algoritm de estimare a parametrilor, inițial probabilitățile fiind necunoscute.

Restul de generatori sunt folosiți pentru obiectele de urmează a popula platformele. Din cauza complexității și a numărului mare de stări și etichete nu se poate realiza o reprezentare grafică. Cel mai mare dintre generatori are douăzeci de stări și zece obiecte ce au rol de etichete pentru emisie.

O observație importantă legată de mulțimea $V$ este că aceasta conține un element mai special și anume un *GameObject* denumit sugestiv epsilon. Acest obiect semnifică elementul vid, prin care se poate insera un spațiu între etichete pentru un aspect mai plăcut la instanțierea pe platforme. Această decizie s-a luat din necesitatea de a separa unele obiecte de celelalte pentru a putea reproduce mai fidel lumea reală.

## 1.3 Estimare de parametri

Această dilema este una dintre cele trei mari aspecte a modelelor Markov cu stări invizibile, celelalte două fiind determinarea probabilității de emisie a unei secvențe și determinarea celei mai bune secvențe de stări ce descriu seria de emisii observate. Problema estimării de parametri este cea mai grea dintre cele trei, nici acum neexistând un algoritm simplu ce rezolvă această problemă. Majoritatea algoritmilor folosesc o metodă iterativă pentru a incerca o estimare a parametrilor ce descriu unu HMM și anume $\lambda = \{\mathbf{A}, \mathbf{B}, \pi\}$.

Din cauză naturii iterative în general acești algoritmi se bazează pe convergența parametrilor pentru terminare. Un lucru bun este că s-a demonstrat apropierea de un anumit punct a acestor algoritmi ceea ce inseamnă ca se termină întotdeauna, cu toate acestea abordarea iterativă suferă de o convergență foarte înceată și de pericolul continuu de a rămâne blocat într-un optim local.

De aceea s-au încercat multe artificii pentru a obține un $log - likelihood$ cât mai bun. Unul dintre ele, folosit și de această aplicație, este inițializarea random a parametrilor $\lambda = \{\mathbf{A}, \mathbf{B}, \pi\}$ și incercarea antrenării până la convergență de mai multe ori pastrându-se cel mai bun rezultat. De asemenea numărule de stări poate influența drastic comportamentul modelului, fiin un factor decisiv pentru $underfit$ și $overfit$.

Din păcate determinarea numărului de stări latente optime pentrun un model dat este o problemă grea ce nu are o soluție fixă. În general abordarea acestei probleme este prin $trial\ and\ error$. Există și anumite criterii bazate pe $log - likelihood$ cum ar fi $BIC$

[2] sau $AIC$ [3].

În cadrul acestei proiect a fost folosită metoda $trial\ and\ error$ în defavoarea criteriilor prezentate mai sus doarece, modelele sunt relativ mici și nu necesită cei mai optimi parametri. În general s-a urmărit ca numărul de stări să fie mai mare decăt numărul de etichete, ceea ce previne de fapt inabilitatea rețelei de a învăța restricțiile la antrenare. Acest lucru se mai numește și $underfit$.

## 1.4 Algoritmi ce tratează problema estimării de parametri

Cel mai cunoscut algoritm pentru inferare de parametri, fiind dat un model Markov cu stări invizibile este așa numitul algoritm $Baum-Welch$ sau algoritmul $Forward-Backward$. Așa cum implică și numele acest algoritm are la bază două etape. Inițial parametri modelului $\lambda = \{\mathbf{A}, \mathbf{B}, \pi\}$ sunt inițializați random și mai apoi actualizați după fiecare iterație până la convergență după cum urmează:

- **Forward** este etapa în care se calculează de obicei $P(O|\lambda)$, unde $O$ este secvența de etichete observate, folosind tehnica programării dinamice pentru stocarea anterioară a rezultatelor. Acest calcul este necesar doarece va fi folosit în urmatoarea etapă pentru reajustarea parametrilor modelului.

- **Backward** este etapa de calcul a probabilității $P(O_{t+1}, \ldots, O_T|S_i, \lambda)$, unde $T$ este lungimea de etichete observate, ceea ce semnifică șansa de a observa secevența $O_{t+1}, \ldots, O_T$ aflându-ne în starea $i$ și la momentul $t$ din timp.

În ciuda popularității acestui algoritm am decis să folosesc altă modalitate de estimare a parametrilor din motive de viteză și simplitate atăt la nivel de implementare căt și la nivel teoretic. Publicația ce descria acest algoritm prezenta grafice convingătoare în privința vitezei atunci căND lungimea secvenței observate $T$ depășește numărul de etichete al modelului.

Un alt aspect alt acestei abordări este că algoritmul actualiează doar doar parametri $\mathbf{A}, \mathbf{B}$ ai modelului ignorând complet parametrul $\pi$ ceea ce duce la o amplificare a vitezei cu consecințe minime, acesta find folosit doar la alegerea stării inițiale $q_1$.

[2] Bayesian Information Criterion

[3] Akaike Information Criterion

De asemenea ca și $Baum - Welch$, fiind dată o secvență $O$ de etichete observate de lungime $T$, antrenare nu se poate face pe o subsecvență $O_1, \ldots, O_m$ iar mai apoi pe o alta subsecventa $O_{m+1}, \ldots, O_T$, cu $1 < m < k$, deoarece modelul ar avea parametri adaptați doar după ultima subsecvență pe care a fost antrenat. Un alt aspect important este normalizarea valorilor ce se face la fiecare iterație după actualizarea parametrilor. Acest lucru este necesar doarece fiecare tranziție și emisie depinde în fond de o varaibilă aleatoare a carei sumă de probabilități trebuie să fie exact egala cu unu.

Algoritmul pe care am decis să îl folosesc are la bază o matrice $C$ de dimensiune $M \times M$ construită cu scopul de a caracteriza numărul de apariții a unei anumite perechi $O_t O_{t+1}$ în secvența de etichete observate $O$.

Această soluție se bazează foarte mult pe calculul matriceal de accea în cadrul aplicației am fost nevoit să apelez la o librărie ce este folosită pentru calcul numeric. Deoarece acest algoritm se bazează foarte mult pe matrici, o incercare de implementare folosind unități de procesare grafice ar putea îmbunătăți substanțial viteza de convergență a algoritmului datorită puterii mai mari de calcul.

Parametri $\mathbf{A}, \mathbf{B}, \pi$ sunt initializați random și normalizați pe linii, iar apoi este calculat $\bar{C} = B\bar{A}B^T$, unde $\bar{A} = \pi_k \cdot a_{kl}, 1 \leq k \leq N, 1 \leq l \leq N$. Următorul pas este calcularea matricii $R$ prin împărțirea element cu element a matricii $C$ la matricea $\bar{C}$.

După calculul acestor matrici intermediare urmează actualizare paremetrilor după urmatoarele ecuații preluate din publicație, $\bar{A}' = \bar{A} \odot B^T RB$ si $B' = B \odot (RB\bar{A}^T + R^T B\bar{A})$, unde $\odot$ reprezintă înmulțirea element cu element a două matrici. După acest calcul se face actualizarea astfel $A = norm(\bar{A}'), B = norm(B')$, unde $norm$ este o funcție ce realizează normalizarea pe toată matricea $A$ iar pentru $B$ doar pe coloane.

Acești pași se realizează pănă un anumit criteriu de convergență este satisfăcut. De obicei aceste criterii includ ca norma matricilor $A, B$ să fie sub un anumit epsilon împreună cu funcția de $log-likelihood$. În implementarea acestui algoritm am folosit ca și criterii de convergență, cum sugera și publicația de unde am preluat ideea, calcularea unei norme de tip $L^2$ aplicata matricilor $A, B, R$, ce ar trebui sa fie sub un $\epsilon = 10^{-6}$ și un număr maxim de iterații pentru care ar trebui să fie îndeplinită. Bineînțeles aceste condiții sunt doar cele standard, ele putând fi ușor schimbate în funcție de necesitate.

## 1.5 Limitări ale modelului

În secțiunile anterioare am văzut avantajele modelului Markov cu stări invizibile dar acesta suferă de o mare problemă, blocarea într-un optim local. Din cauza acestei probeleme estimarea de parametri este o sarcină destul de dificilă și delicată. De asemenea modelul depinde foarte mult de proprietatea Markov ce specifică clar că o stare viitoare poate să depindă doar de starea curentă, limitând astfel capacitatea de a modela date mult mai complex. Se poate extinde modelul la ultimele $n-1$ stări dar complexitatea de antrenare și durata crește considerabil.

Cea mai simplă soluție pentru a preveni blocarea este inițializarea random a parametrilor modelului și reantrenarea lui de mai multe ori, pastrandu-se cel mai bun rezultat. Acestă soluție este fezabilă deoare prin randomizare se pot obține parametri din vecinătatea soluției optime, scăzând posibilitatea ca algoritmul să ramană blocat într-un optim local, atunci când există o multitudine de aceste puncte. Deși această soluție rezolva problema în unele situații reantrenarea de foarte multe ori este costisitoare, de accea există posibilitatea de a salva parametri estimați, putând fi ulterior utilizați de model prin o simplă citire.

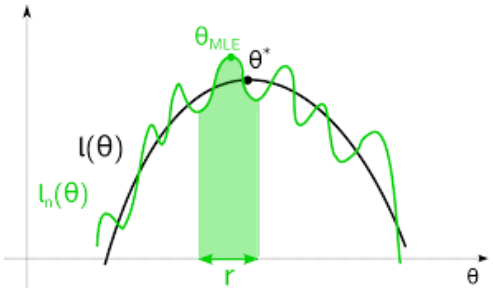


Figura 1.3: Grafic al functie $log-likelihood$ cu foarte multe puncte de optim local

Din cauza acestor limitări algoritmii de estimare a parametrilor precum $Baum-Welch$ nu garantează găsirea optimului global. Deseori un optim local este suficient pentru probleme ce nu necesită soluția optimă dar trebuie luat in considerare acest aspect atunci când se utiliează un astfel de algoritm.

O altă problemă a acestei abordări este că in general este asumată independența etichetelor una față de celălaltă limitând astfel flexibilitatea modelului și capacitatea de modelare a datelor.

## 1.6 Complexitate și metode de optimizare

În acestă secțiune se vor prezenta problem legate de complexitatea algoritmului de antrenare cât și a algoritmului de eșantionare dintr-o distribuție de probabilitate discretă. Conform publicației algoritmul de estimare a parametrilor prezentat anterior ce foloseste de o matrice de apariție $C$ are complexitatea timp de ordinul $O(IM^2N+T)$

, unde $I$ reprezintă numărul de iterații până la convergență.

Comparativ cu algoritmul $Baum-Welch$ ce are ca si complexitate timp $O(IN^2T)$ iar în practică secvența de etichete pe care e antrenat algoritmul are o lungime considerabil mai mare decât cardinalul mulțimii $V$, adică $M$, de unde se observă că abordarea cu matricea de apariție aduce un surplus de rapiditate de procesare ce este important în domeniul jocurilor unde viteza este foarte importantă pentru a nu îngreuna procesul de randare și mulți alți factori de care este influențat un joc complex.

Legat de procesul de eșantionare, acesta este realizat prin calcularea distribuției $CDF$[4] pentru parametri $A, B$ ce mai apoi sunt sortate în ordine crescătoare pe linii pentru a putea realiza o căutare binară a stării către care se va face tranziția fiind dat un număr generat pseudorandom.Din punct de vedere al complexității sunt necesari pași adiționali cum ar fi sortarea și stocarea valorilor cumulative dar asta se face în etapa de inițializare având efect minimal asupra jocului. Ce se poate spune totuși de această abordare este că timpul de cautare a stării se transforma din $O(N)$, unde $N$ repezintă numărul de stări, în $O(log(N))$. Această optimizare este importantă deoarece operația de eșantionare are loc de zeci de ori pentru o platformă fiind practic cea mai folosită operație din cadrul librăriei.

Deși în practică metoda de $sampling$ în timp de $O(log(N))$ s-a dovedit a fi suficientă, există un algoritm și mai eficient decât cel prezentat ce reușeste sa determine starea sau emisia find dat un număr generat random în timp de $O(1)$, având aceiași complexitate la inițializare cât și memorie ca și algoritmul de căutare binară.

Acest algoritm se numește *metoda lui Alias*, și reușeste să obțina cea mai bună viteză posibilă pentru eșantionarea dintr-o distribuție de probabilitate discretă. În general ideea din spatele algorimlui este de a crea o altă distribuție pe baza celei de la intrare ce permite eșantionare în timp constant.

[4]Cumulative Distribution Function

# Capitolul 2

# Tehnologii

În cadrul acestui proiect s-au folosit următoarele tehnologii :

- **Unity 2019.1b** : motorul grafic folosit pentru construirea și randarea jocului.
- **Blender** : program *open source* folosit pentru modelarea 3D și contrucția obiectelor ce au fost ulterior importate în motorul grafic.
- **Math.NET** : librărie *open source* folosită pentru realizarea algoritmului de învățare automată deoarece acesta se bazează foarte mult pe lucru cu matrici.
- **C#** : limbajul nativ folosit de Unity pe lângă JavaScript , decizia fiind luată pur din motive de viteză.

Din multitudinea de motoare grafice disponibile am ales **Unity** doarece e o tehnologie cu care am mai lucrat, fiind conștient de capabilitățile acestuia. Multitudinea de unelte îl fac foarte ușor de recomandat și împreună cu magazinul de *assets* m-au ajutat foarte mult în procesul de construcție al acestui proiect. De asemenea **Unity** dispune de o documentație foarte bogată și bine pusă la punct , calități cheie pentru aprofundarea acestui motor grafic.

Câteva exemple de unelte *in-house* de care dispune motorul grafic ar fi , motorul de simularea a fizicii , motorul audio , sistemul de prefabricate , sistemul de iluminare și randare etc. Fără folosirea unei tehnologii ca **Unity** , acest proiect nu ar fi fost posibil într-un timp atăt de scurt , timpul de dezvoltare putând fi chiar triplat. Singurul dezavantaj imediat al acestui motor grafic este natura sa *closed source*.

Pentru modelele 3D folosite în cadrul jocului a fost necesară folosirea unei editor și anume **Blender**. Deși mare parte din obiectele importate în joc sunt preluate de pe

magazinul integrat în **Unity** , există anumite obiecte ce au fost modelate de către mine. Cu acest prilej am aprofundat anumite tehnici ce implică modelarea 3D , unul dintre cele mai importante find *UV unwrapping*.

Am ales **Blender** datorită politcii *open source* și a documentației vaste disponibile, fiind foarte ușor de lucrat în acesta, neputând aduce aplicația la un nivel dorit de finisaj fară acestă unealtă.

Din punct de vedere al librăriilor externe am folosit **Math.NET** , importată în **Unity** folosind **NuGet**. Necesitatea acestei librări este datorat algoritmului de invățare automată ce se bazează foarte mult pe lucrul și manipularea matricilor, de la adunarea cu un scalar până la înmulțirea/împărțirea element cu element. Detaliile legate de acest algoritm împreuna cu avantajele și dezavantajele acestuia vor fi discutate în secțiunea teoretică.

Datorită motorului grafic folosit și a limitărilor acestuia întregul proiect a fost construit folosind **C#**, alternativa sa fiind **JavaScript**. Din cauza experienței cu **C#** și a performanței ridicate am luat decizia de a folosi acest limbaj.

## 2.1 Structura proiectului

Aplicația creată este împărțită în doua etape , generare și instanțiere. Din cauza unor motive de performanță fiecare zonă, deșertică, rurală și urbană are propriul său generator. De asemenea fiecare tip de platformă folosește propriul său model Markov însumând astlfel un total de șase generatori, ce se ocupă doar cu producerea unei secvențe specifice de obiecte. Pentru partea de instanțiere am fost nevoit să construiesc un sistem ce facilitează instanțierea obiectelor generate, plasândule pe platformă în sensul acelor de ceasornic.

Aplicația fiind facută în **Unity** fișierele și directoarele au o structură arborescentă după cum urmează:

```
Assets
    Generators
    HMM
    Sound
    Prefabs
    Scripts
    Textures
    VolumetricLights
Packages
    PostProcessing
```

Directorul de **Assets** conține toate elementele necesare aplicației, de la librăria folosită pentru generarea obiectelor până la modulele de sunet și interfață iar directorul **Packages** conține cele mai importante uneltele folosite pentru realizarea și finisajul jocului importate direct de către **Unity**.

**Generators** aici se află cei șase generatori folosiți pentru construirea mediului.

**HMM** este directorul ce conține scripturile necesare pentru construirea modelului Markov și a antrenării lui.

**Sound** conține toate efectele de sunet și muzica din joc.

**Prefabs** este directorul cu toate prefabricatele din joc, incluzând modelele pentru platforme, obiecte, jucător cât și inamici.

**Scripts** aici sunt stocate toate fișierele sursă ce modelează comportamentul jucatorului, al meniurilor și a sistemelui de generare/instanțiere, sumându-se în total la optsprezece fișiere.

**Textures** directorul ce conține toate fișierele de tip *Albedomap* , *Heightmap* și *Occlusionmap* folosite în cadrul materialelor ce au fost ulterior importate pe obiecte.

**VolumetricLights** directorul ce conține fișierele preluate de pe *Github* pentru lumină volumetrică.

**PostProcessing** modulul folosit de *Unity* pentru a putea integra efecte precum corectare de culoare, *bloom*, *motion blur* și multe altele.

## 2.2 Componente

Aplicația prezentă este împărțită pe diferite module ce controlează sunetul, generarea, *UI-ul* , instanțierea și multe altele. În **Unity** acest lucru se realizează în mare parte prin scripturi atașate de obiectele din scena curentă.

Am optat să construiesc jocul în jurul unei singure scene, ceea ce necesită un numar mai mare de controalere și scripturi. O consecință directă a folosirii unei singure scene este integrarea meniului principal în același mediu ca și jocul în sine. De aici derivă necesitatea de două scripturi ce controlează tranziția de la meniu la joc intitulate sugestiv *MenuController* și *GameController*.

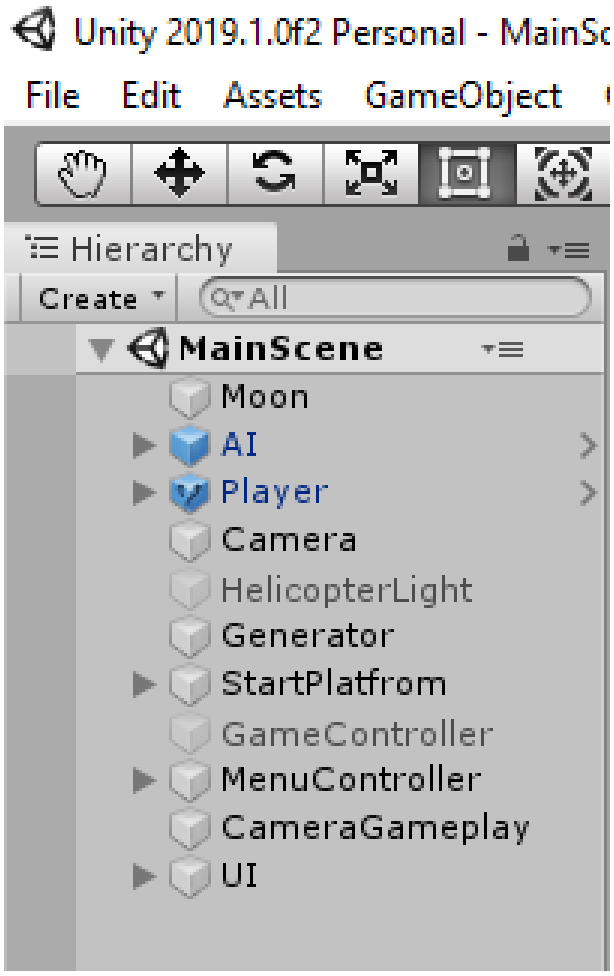


Figura 2.1: Structura arborescentă a obiectelor din scenă

De asemenea există un obiect *Generator* ce conține toți generatori necesari pentru construirea scenei. Unul pentru fiecare tip de zonă de interes și tip de platformă. Legătura între cei șase generatori este facută de cele două scripturi atașate de obiectul *Camera* denumite *PlatformBuilder* și *PlatformController*. Primul se ocupă cu instanțiera și plasarea obiectelor pe platformă în funcție de parametri de instanțiere setați de un script atașat de un *Spawn Point*. Cel de-al doilea se ocupă cu randomizarea și controlarea numărului de platforme pentru fiecare zonă rurală , urbană și deșertică.

### 2.2.1 MenuController

Acest obiect se ocupă de toate aspectele ce țin de meniul principal. Deoarece există o singură scenă scriptul este destul de complex , rolul său fiind activarea tuturor componentelor ce țin de jocul principal, sunet, inamici, jucător, cameră, efecte și multe altele.

În continuare se vor prezenta cele mai importante secvențe de cod ce se găsesc în

fișierul **MenuController**.

Exemplu 2.1: Funcțiile din MenuController

```
public void OnGameExit(){
        Application.Quit();
}
public static void OnGameReset(){
        retry = true;
}
public void OnCredits(){
        this.creditsStart = true;
        this.credits.SetActive(true);
}
public void OnGameStart(){
        gameStart = true;
        gameController.SetActive(true);
        player.constraints = RigidbodyConstraints.None;
        AI.constraints = RigidbodyConstraints.None;
        player.velocity = Vector3.forward * 3;
        AI.velocity = Vector3.forward;
        bars.SetActive(true);
}
```

Cele patru funcții controlează tranzițiile aplicației în funcție de interacțiunea jucatorului cu meniul principal. De remarcat în funcția *OnGameStart* linia ce activează *GameController* odată ce se pornește jocul și anume *gameController.SetActive(true)*.

### 2.2.2 GameController

Obiectul ce se ocupă de funcționalitatea principală din joc. În mare parte meniul de pauză și meniul de final al jocului sunt controlate de acest script.

Legat de parte de cod , fișierul sursă atașat de obiect arată astfel:

Exemplu 2.2: Funcțiile din GameController

```
public void SetPausedState(){
        gamePaused = gamePaused == true ? false : true;
}
public void SetGameOverState(){
        player.GetComponent<CarController>().enabled = false;
        cameraScript.enabled = false;
        enemyScript.enabled = false;
        gameOver = true;
        gameOverScreen.SetActive(true);
}
public void TogglePause(){
        ToggleSound();
        ToggleRender();
        TogglePauseMenu();
        SetPausedState();
}
```

Meniul din joc odată activat oprește randarea jocului, lucru ce complică apelu de funcții dependente de timp.

### 2.2.3 PlatformBuilder

Scriptul ce realizează construirea în timp real a platformelor și le instanțiază. Datorită volumului mare de cod mai jos este prezentată doar funcția ce adaugă obiectele pe platformă. Înainte de instanțiere funcția are grija să preia datele de *spawn* , conținute de scriptul atașat *SpawnSettings* pentru fiecare loc valid de instanțiere.

Exemplu 2.3: Funcțiile din PlatformBuilder

```
1  public GameObject BuildPlatform(GameObject state){
2      GameObject nextPlatform = Instantiate(state, new Vector3(currentPlatform.transform.position.x + xOffset, −15, currentPlatform.transform.position.z + zOffset),
           Quaternion.Euler(0, 0, 0));
3      HMM propGenerator = ChoosePropGenerator(nextPlatform.name);
4      Transform spawnPoints = nextPlatform.transform.Find("SpawnPoints");
5      foreach (Transform spawn in spawnPoints){
6          SpawnSettings spawnSettings = spawn.GetComponent<SpawnSettings>();
7          if (spawnSettings.isStackable == false && spawnSettings.maxObjectStack > 1)
8              throw new System.Exception("Spawn is not stackable!");
9          for (int i = 0; i < spawnSettings.maxObjectStack; ++i){
10             InstantiateProp(propGenerator, spawn, spawnSettings, i);
11         }
12     }
13     nextPlatform.transform.rotation = Quaternion.Euler(0, rotation, 0);
14     return nextPlatform;
15 }
```

Scriptul *SpawnSettings* conține doi parametri *isStackable* și *maxObjectStack* folosite pentru a specifica dacă locul de *spawn* poate să conțină mai multe obiecte și în ce cantitate.

### 2.2.4 PlatformController

Din cauza necesității de a controla durata unei anumite zone evitând astfel inconsistența, *PlatformController* alege un număr de platforme pentru fiecare secțiune tinând cont de o limită superioara și inferioară.

Exemplu 2.4: Funcțiile din PlatformController

```
1  public GameObject GetNextPlatform(){
2      GameObject output;
3      currentEmissionCount++;
4      if (currentEmissionCount > maxEmissionCount){
5          output = transitionPlatforms[currentGeneratorIndex];
6          currentEmissionCount = 0;
7          currentGeneratorIndex = (currentGeneratorIndex + 1) % generators.Length;
8          maxEmissionCount = Random.Range(lowerBound, upperBound);
9      }
10     else{
11         output = generators[currentGeneratorIndex].NextEmission();
12     }
13     return output;
14 }
```

# Capitolul 3

# Arhitectura aplicației

În mare parte aplicația este construită în jurul unei singure scene folosind $Unity$, arhitectura aplicației depinzând în mare parte de uneltele disponibile în motorul grafic și modul în care acesta funcționează Am încercat să împart cât de mult se poate să împart aplicația în module cu o coeziune cât mai ridicată. De acceea sistemul de instanțiere, generare și amplasare a obiectelor este construit cu principalul scop de a fi modular. Chiar și librăria ce modelează prototipul Markov cu stări invizibile este decuplat de algoritmul de antrenare în sine, putând fi schimbat oricând fără repercursiuni.

Aplicația folosește și multe funcții de $callback$ pentru a determina când anumite elemente din cadrul jocului ar trebui activate. Spre exemplu cea mai folosită funcție este cea de $OnTriggerEnter$, ce determină când un $RigidBody$[1] se intersectează cu un *BoxCollider*[2].

Exemplu 3.1: Exemplu de utilizare a functiei OnTriggerEnter

```
1 void OnTriggerEnter(Collider collidingObject){
2         if (!this.triggered){
3             this.callbackObject.GetComponent<PlatformBuilder>().InstantiatePlatform();
4             this.triggered = true;
5         }
6 }
```

De asemena $Unity$ dispune de o unealtă foarte utilă numită $Inspector$ ce permite asignarea de referințe a obiectelor din scenă în scripturi, lucru foarte util și eficient atunci când este necesară legarea anumitor module. De exemplu $callbackObject$ reprezintă obiectul din scenă de care este atașat codul sursă ce se ocupă de construirea platformelor, numit sugestiv $PlatformBuilder$, din care se apelează funcția $InstantiatePlatform$

[1]Componenta ce determina daca obiectul este supus motorului de fizica

[2]Componenta ce determina zona de coliziune a unui obiect

destinată acestui scop.

## 3.1 Detalii legate de implementarea librăriei

Cum menționam și anterior pentru implementare am avut nevoie de o librărie ce se ocupă cu calcul numeric. Am folosit $Math.NET$ pentru calculul matriceal și a normelor $L^2$, împreună cu structurile speciale $Matrix$ și $Vector$ din cadrul librarie pentru a manipula parametri $\lambda = \{\mathbf{A}, \mathbf{B}, \pi\}$ ai modelului.

În cadrul librăriei s-au folosit o multitudine de funcții auxiliare ce ajută la normalizare și structurarea mai bună a codului. Mai jost sunt prezentate cele mai importante funcții auxiliare din cadrul acesteia.

Exemplu 3.2: Funcție auxiliara ce realizează normalizarea

```
1  private Matrix<double> Normalize(Matrix<double> target, Vector<double> divisors, string mode){
2          Matrix<double> output = Matrix<double>.Build.DenseOfMatrix(target);
3          for (int i = 0; i < target.RowCount; ++i){
4              for (int j = 0; j < target.ColumnCount; ++j){
5                  if (mode == "row")
6                      output[i, j] /= divisors[i];
7                  if (mode == "column")
8                      output[i, j] /= divisors[j];
9              }
10         }
11         return output;
12 }
```

Exemplu 3.3: Funcție auxiliara ce construiește matricea de frecvență $C$

```
1  private Matrix<double> GetOccurenceMatrix(List<int> observations){
2          Matrix<double> occurences = Matrix<double>.Build.Dense(this.emissionCount, this.emissionCount);
3          for (int i = 0; i < observations.Count - 1; ++i){
4              occurences[observations[i], observations[i + 1]] += 1;
5          }
6          return occurences;
7  }
```

Exemplu 3.4: Funcție auxiliara ce contruieste o matrice cu intrări random

```
1  private void SetRandomMatrix(double[,] matrix){
2          double[] divisors = new double[matrix.GetLength(0)];
3          double rowMax = 0.0f;
4          for (int i = 0; i < matrix.GetLength(0); ++i){
5              rowMax = 0;
6              for (int j = 0; j < matrix.GetLength(1); ++j){
7                  matrix[i, j] = Random.Range(0.0f, 100.0f);
8                  rowMax += matrix[i, j];
9              }
10             divisors[i] = rowMax;
11         }
12         Normalize(matrix, divisors);
13 }
```

Legat de aplicarea capitoluilui teoretic, antrenarea este realizată de o singură funcție căreia îi sunt pasați parametri modelului, împreună cu secvența de etichete observate. Funcția aplică ecuațiile discutate în cadrul capitolului de aspecte teoretice, bineînțeles adaptate la limbajul de programare folosit.

Exemplu 3.5: Implementare algoritmului de antrenare a modelului

```
1  public double Train(List<int> observations, double[,] transitionProbabilities, double[,] emissionProbabilities, List<double> pi){
2          this.SetEmissionMatrix(emissionProbabilities);
3          this.SetTransitionMatrix(transitionProbabilities);
4          this.SetDrawProbabilities(pi);
5          Matrix<double> previousJoinDistribution = Matrix<double>.Build.Dense(this.emissionCount, this.emissionCount);
6          Matrix<double> previousEmission = Matrix<double>.Build.Dense(this.stateCount, this.emissionCount);
7          Matrix<double> previousTransition = Matrix<double>.Build.Dense(this.stateCount, this.stateCount);
8          Matrix<double> jointDistribution;
9          Matrix<double> emissionDelta;
10         Matrix<double> transitionDelta;
11         Matrix<double> occurenceMatrix = this.GetOccurenceMatrix(observations);
12         this.SetMatrixBar();
13         this.emissionMatrix = this.emissionMatrix.Transpose();
14         previousEmission = previousEmission.Transpose();
15         double likelihood = 0.0f;
16         for (int i = 0; i < this.maxIterations; ++i){
17             jointDistribution = occurenceMatrix.PointwiseDivide(this.emissionMatrix.Multiply(this.transitionMatrix).Multiply(this.emissionMatrix.Transpose()));
18             transitionDelta = this.transitionMatrix.PointwiseMultiply(this.emissionMatrix.Transpose().Multiply(jointDistribution).Multiply(this.emissionMatrix));
19             emissionDelta = this.emissionMatrix.PointwiseMultiply(jointDistribution.Multiply(this.emissionMatrix).Multiply(this.transitionMatrix.Transpose()).Add(
                       jointDistribution.Transpose().Multiply(this.emissionMatrix).Multiply(this.transitionMatrix)));
20             this.transitionMatrix = transitionDelta.Divide(transitionDelta.RowSums().Sum());
21             this.emissionMatrix = Normalize(emissionDelta, emissionDelta.ColumnSums(), "column");
22             if (this.transitionMatrix.Subtract(previousTransition).L2Norm() < this.epsilon && this.emissionMatrix.Subtract(previousEmission).L2Norm() < this.epsilon &&
                       jointDistribution.Subtract(previousJoinDistribution).L2Norm() < this.epsilon)
23                 break;
24             previousEmission = emissionMatrix;
25             previousTransition = transitionMatrix;
26             previousJoinDistribution = jointDistribution;
27         }
28         likelihood = GetLikelihood(occurenceMatrix, this.transitionMatrix, this.emissionMatrix.Transpose(), this.emissionMatrix.Multiply(this.transitionMatrix).Multiply(this.
                   emissionMatrix.Transpose()));
29         this.transitionMatrix = Normalize(this.transitionMatrix, this.transitionMatrix.RowSums(), "row");
30         this.emissionMatrix = this.emissionMatrix.Transpose();
31         return likelihood;
32 }
```

Funcția $Train$ descrisă mai sus este folosită în clasa mare denumită sugestiv $HMM$ pentru a antrena modelul. Modul cum funcționează $Unity$ toate clasele ce sunt derivate din $MonoBehaviour$ pot fi atașate pe un obiect din scenă. De accea există un obiect ascuns în scenă ce are atașat toți generatorii, putând schimba foarte ușor toți parametri modelului chiar din $Inspector$. Din cauza acestui lucru librăria este foarte ușor de preluat și utilizat, aceasta conținând și un modul de salvare/citire a parametrilor modelului.

În continuare se va prezenta o diagramă $UML$ ce descrie librăria și relația dintre componentele ei. Se poate observa că modelul Markov cu stări invizibile poate lucra complet independent de algoritmul de învățare automată.

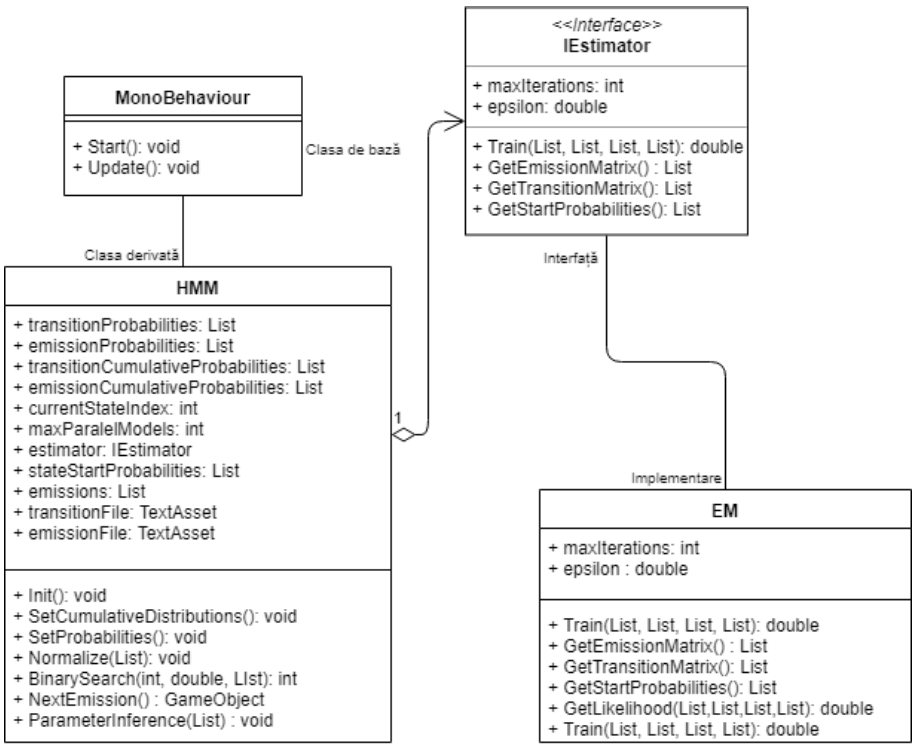


Figura 3.1: Diagrama UML ce modelează libraria de HMM

*IEstimator* este interfața ce specifică ce proprietăți și metode algortimul de estimare are nevoie să implementeze. Se observă din schemă implementare modelului $Strategy$, ce elimină asocierea dintre clasa *HMM* și algoritmul de antrenare. Din această cauză se poate face foarte ușor schimbarea algoritmului de antrenare în timpul rulării programului în funcție de necesitate.

Exemplu 3.6: Folosirea funcție $Train$ în cadrul modelului

```
1  public void ParameterInference(List<int> observations)
2  {
3          double[,] bestEmission = this.emissionProbabilities;
4          double[,] bestTransition = this.transitionProbabilities;
5          double[] startProbabilites = this.stateStartProbabilities.ToArray();
6          double maxLoglikelihood = double.MinValue;
7          for (int i = 0; i < this.maxParalelModels; ++i){
8              double likelihood = estimator.Train(observations, this.transitionProbabilities, this.emissionProbabilities, this.stateStartProbabilities);
9              if (maxLoglikelihood < likelihood){
10                 maxLoglikelihood = likelihood;
11                 bestEmission = estimator.GetEmissionMatrix();
12                 bestTransition = estimator.GetTransitionMatrix();
13                 startProbabilites = estimator.GetStartProbabilities();
14             }
15             this.SetRandomProbabilities();
16         }
17         this.emissionProbabilities = bestEmission;
18         this.transitionProbabilities = bestTransition;
19         this.stateStartProbabilities = new List<double>(startProbabilites);
20         this.currentStateIndex = this.SetInitialState();
21         this.SetCumulativeDistributions();
22     }
23 }
```

## 3.2 Detalii legate de implementarea sistemului de instanțiere

Așa cum am mai menționat modelul Markov cu stări invizibile este folosit ca și generator iar pentru instanțiere a fost necesară implementarea unui sistem ce folosește puncte de *spawn* amplasate pe platforme. Odată cu aceste puncte există și un script ce descrie dacă punctul de instanțiere poate să conțină mai multe obiecte sau nu, amplasarea lor facandu-se unu după altul. Se poate specifica și rotația obiectului, dar in general se urmarește ca obiectele de pe platformă să fie orientate înspre jucător, în sensul acelor de ceasornic.

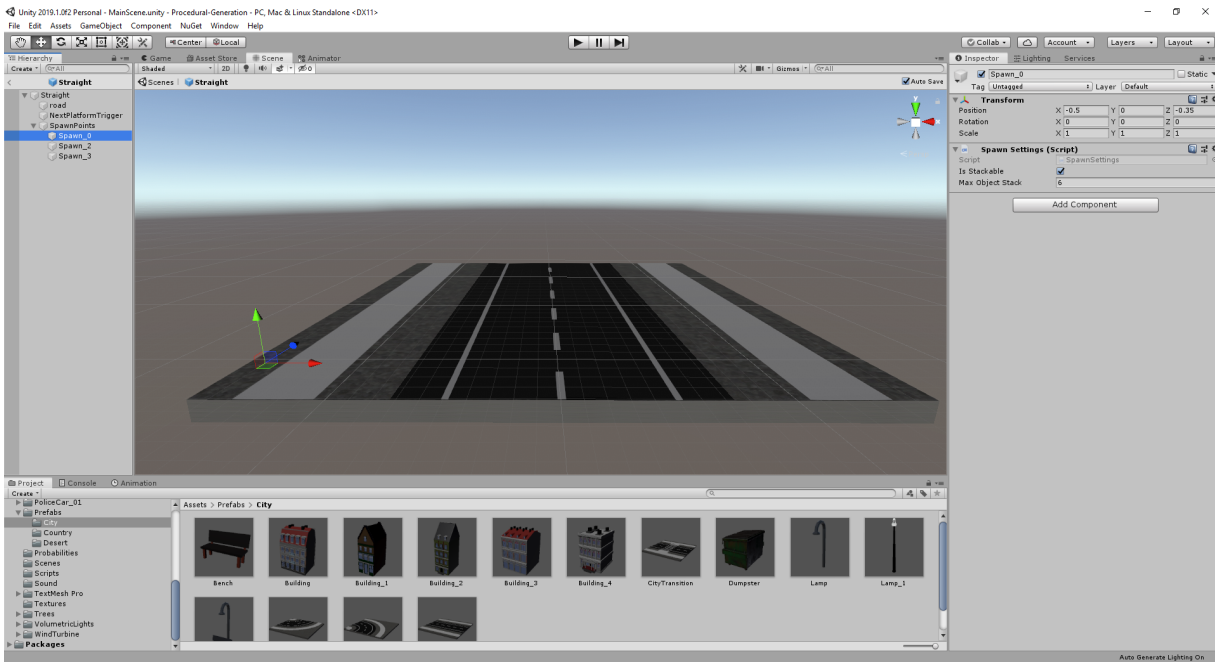

Figura 3.2: Sistemul de instanțiere

De asemenea punctele de *spawn* sunt plasate exact pe marginea platformei iar algoritmul de instanțiere ia în calcul acest lucru. Cu toate acestea ele pot fi mutate spre interiorul platformei pentru a da senzația de realism. O limitare a acestui sistem este sensibilitatea la numărul de obiecte pe un loc de instanțiere, deoarece acesta nu ține cont de lungimea platformei.

## 3.3 Utilizarea librăriei în joc

În dezvoltarea acestei librării obiectivele principale au fost modularitatea și usurința de utilizare. Acest lucru a fost realizat foarte simplu datorită modului în care *Unity*

permite atasarea de fisiere sursă pe obiecte din scenă, având la dispoziție toti parametri disponibili din acea clasă.

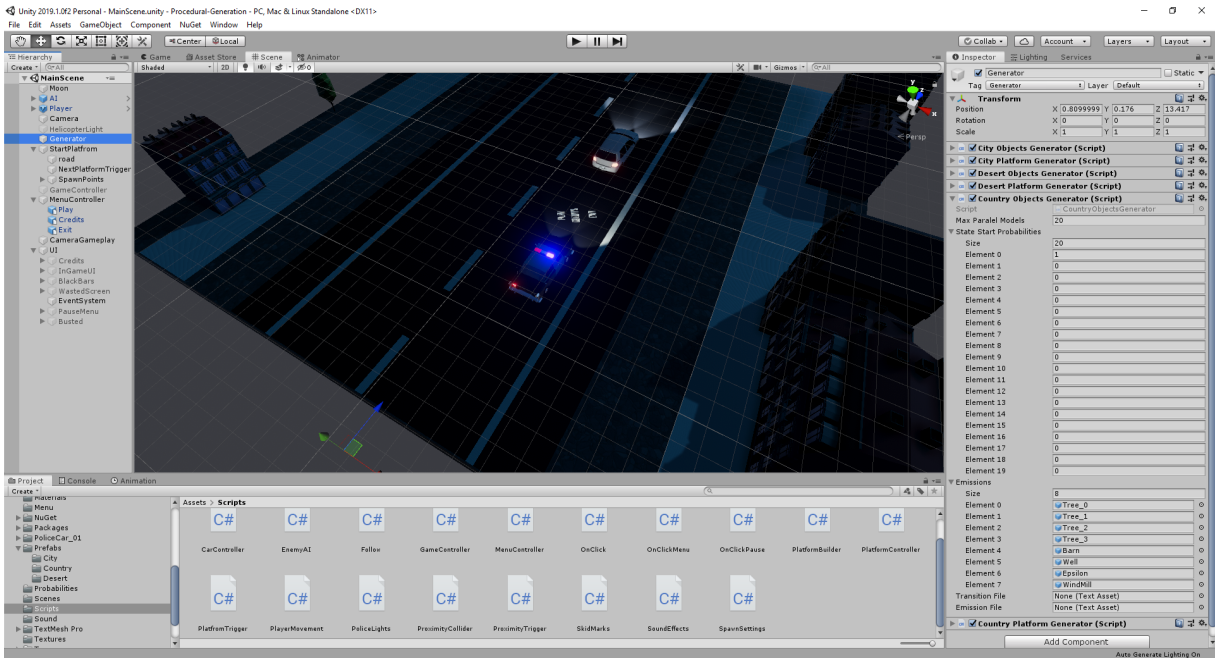

Figura 3.3: Generatorii folosiți în cadrul jocului

În general pentru a putea folosi librăria, se definește o clasă nouă ce moștenește din clasa $HMM$, în cadrul careia se apelează funcția $Init()$ în funcția $Start()$.

Exemplu 3.7: Exemplu de folosire a librăriei

```
1 public class DesertObjectsGenerator : HMM{
2     void Start(){
3         this.Init();
4     }
5 }
```

Este necesară această procedură din cauza modului cum functionează $Unity$, deoarece scripturile nu pot fi atașate pe obiecte în cadrul scenei dacă nu sunt derivate din $MonoBehaviour$ iar $C\#$ nu permite moștenire multiplă.

## 3.4 Detalii legate de aspectul aplicatiei

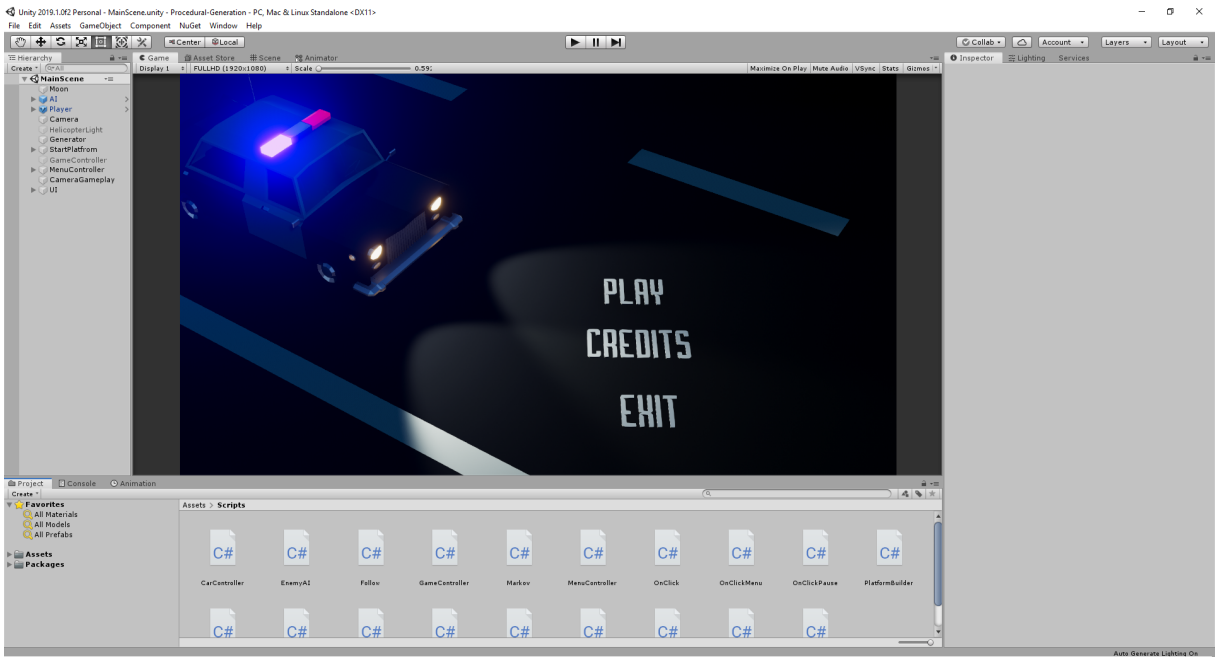


Figura 3.4: Motorul Grafic Unity

## 3.5 Structura întregii aplicații

# Concluzii

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Nunc mattis enim ut tellus elementum sagittis vitae et. Placerat in egestas erat imperdiet sed euismod. Urna id volutpat lacus laoreet non curabitur gravida. Blandit turpis cursus in hac habitasse platea. Eget nunc lobortis mattis aliquam faucibus. Est pellentesque elit ullamcorper dignissim cras tincidunt lobortis feugiat. Viverra maecenas accumsan lacus vel facilisis volutpat est. Non odio euismod lacinia at quis risus sed vulputate odio. Consequat ac felis donec et odio pellentesque diam volutpat commodo. Etiam sit amet nisl purus in. Tortor condimentum lacinia quis vel eros donec. Phasellus egestas tellus rutrum tellus pellentesque eu tincidunt. Aliquam id diam maecenas ultricies mi eget mauris pharetra. Enim eu turpis egestas pretium.

# Bibliografie

## 3.6 Cărți și Articole


- Madhusudana Shashanka, *A Fast Algorithm For Discrete HMM Training Using Observed Transitions*,
  https://pdfs.semanticscholar.org/9c19/af77335ee1e28875b84eeb0b8c71d
  pdf

- Gleidson Mendes Costa, Tiago Bonini Borchartt,
  *Procedural terrain generator for platform games using Markov chain* , 2018, http://
  www.sbgames.org/sbgames2018/files/papers/ComputacaoShort/188123.
  pdf

- Fanny Yang,Sivaraman Balakrishnan, Martin J. Wainwright,
  *Statistical and Computational Guarantees for the Baum-Welch Algorithm*, 2015,
  https://arxiv.org/pdf/1512.08269.pdf

- Sam Snodgrass, Santiago Ontañón,
  *Experiments in Map Generation using Markov Chains*,2014,
  http://fdg2014.org/papers/fdg2014_paper_29.pdf

- Liviu Ciortuz, *Machine Learning Course*,
  https://profs.info.uaic.ro/~ciortuz/SLIDES/hmm.pdf

- *Unity User Manual*,
  https://docs.unity3d.com/Manual/index.html


## 3.7 Obiecte preluate și folosite în cadrul aplicației


- Unity Asset Store, *Low Poly European City Pack*,

https://assetstore.unity.com/packages/3d/environments/urban/low-poly-european-city-pack-71042

- Unity Asset Store, *Low Poly Desert Pack*, https://assetstore.unity.com/packages/3d/environments/free-low-poly-desert-pack-106709
- Unity Asset Store, *Low Poly Police Car 01*, https://assetstore.unity.com/packages/3d/vehicles/land/low-poly-police-car-01-142826
- Unity Asset Store, *Low Poly Destructible 2 Cars No.8*, https://assetstore.unity.com/packages/3d/vehicles/land/low-poly-destructible-2-cars-no-8-45368
- Unity Asset Store, *PBR Dirty Dumpster*, https://assetstore.unity.com/packages/3d/props/exterior/pbr-dirty-dumpster-59840
- Unity Asset Store, *Unity Particle Pack 5.x*, https://assetstore.unity.com/packages/essentials/asset-packs/unity-particle-pack-5-x-73777
- Unity Asset Store, *Post Processing Stack*, https://assetstore.unity.com/packages/essentials/post-processing-stack-83912
- Joao Paulo, *Low Poly Farm*, https://3dshed.wordpress.com/2018/08/29/low-poly-farm-animation/
- Joao Paulo, *Wind Turbine*, https://3dshed.wordpress.com/2018/02/12/wind-turbine-001-animated/
- Joao Paulo, *Trees Pack*, https://3dshed.wordpress.com/2018/04/07/trees-pack-001/
- Joao Paulo, *Desert Texture*, https://3dtextures.me/2018/08/13/cracked-mud-001/

- Joao Paulo, *Asphalt Texture*,
  `https://3dtextures.me/2018/05/07/asphalt-005/`
- Joao Paulo, *Stone Tile Texture*,
  `https://3dtextures.me/2018/09/27/stone-tiles-003/`
- Joao Paulo, *Grass Texture*,
  `https://3dtextures.me/2018/01/05/grass-001-2/`
- SlightlyMad, *Volumetric Lights*,
  `https://github.com/SlightlyMad/VolumetricLights`
- DavidMenke, *Main Menu Sound*,
  `https://freesound.org/people/davidmenke/sounds/319750/`
- Kenney.nl, *Buttons Effects*,
  `https://opengameart.org/content/51-ui-sound-effects-buttons-switches-and-clicks`
- *Borg Font*,
  `https://www.dafont.com/borg.font`