

1.Introducere

1.1 Context

Industria jocurilor video este una relativ tânără, dar evoluția pe care a cunoscut-o este una impresionantă. Plecând de la proiecte de cercetare realizate în anii '50, a ajuns în atenția publicului în anii '70 și '80, și o dată cu evoluția tehnologiei a ajuns ecosistemul masiv pe care îl putem observa în ziua de azi.

În fiecare an aceasta atinge câte un nou apogeu, mulțumită nenumăratelor corporații, studio-uri de dezvoltare și dezvoltatori independenți care publică neconținut mii de jocuri pe un număr în continuă creștere de platforme (calculatoare personale, console de jocuri, telefoane mobile, etc.). Această industrie ar putea fi comparată cu cea cinematografică; produsele finale sunt rezultatul unor perioade în general lungi de dezvoltare, în care se investesc bugete foarte variate (plecând de la proiecte fără buget până la proiecte precum Battlefield 4, al cărui buget a fost de o sută de milioane de dolari) și pot fi împărțite în multiple categorii, precum acțiune, aventură, simulatoare și horror.

Motivul pentru care această piață are atât de mult succes este faptul că jocurile oferă o formă de divertisment asemănătoare celor găsite în cărți și filme, dar abordarea este una diferită: pentru desfășurarea poveștii sau furnizarea divertismentului, este necesară intervenția consumatorului prin interacțiunea cu jocul. Astfel, de exemplu, firul narativ al unui roman va fi mereu același, static - nu va suferi modificări indiferent de numărul de întreruperi, viteza și durata sesiunilor de citit, pe când desfășurarea unui joc, în general, nu poate avea loc fără implicarea utilizatorului.

1.2 Motivație

Toate jocurile care intră în categoria horror împărtășesc o trăsătură comună. Ele țintesc să sperie consumatorul. Motivul pentru care există această categorie în mare parte a formelor de divertisment este faptul că desfășurarea, în majoritatea timpului monotonă, a vieții individului aramează în subconștientul acestuia o nevoie ușor masochistă de "fenomenal". Formele de divertisment de tip "horror" constituie una dintre cele mai ușoare metode de satisfacere a acestei nevoi prin traume minore la nivelul psihicului individului.

În continuare, voi descrie câteva exemple de jocuri horror, punând accent pe tehnicile de speriere a jucătorului:

- **Slender: The Eight Pages** (2012): jucătorul controlează un personaj fără o poveste într-o perspectivă first person (persoana I), aflat într-o pădure pe timp de noapte, având la dispoziție doar o lanternă. Scopul acestuia este să strângă 8 pagini aflate în diferite locații în spațiul jocului, în timp ce evită contactul cu

Slender, personajul negativ. Slender urmărește jucătorul tot timpul, inițial de la distanță, fără să fie văzut (contactul vizual prelungit încheie jocul), devenind tot mai agresiv pe măsură ce jucătorul găsește paginile.

- **Five Nights at Freddy's** (2014): jucătorul intră în pielea unui personaj proaspăt angajat ca paznic de noapte al unei pizzerii, controlând jocul într-o manieră point and click (jucătorul nu are libertatea de a mișca personajul după bunul plac). Scopul jocului este acela de a rezista cinci nopți, în condițiile în care în incinta restaurantului se află mai multe figurine animatronice cu intenții malițioase, personajul poate să observe spațiul doar pornind niște camere de supraveghere și singurul mecanism defensiv este o pereche de uși care stau închise doar consumând foarte mult, iar cantitatea de energie electrică pe care o are la dispoziție este foarte mică.
- **Dead Space** (2008): jucătorul preia controlul unui personaj controlabil într-o perspectivă third person (jucătorul vede "peste umărul" personajului) care se găsește singur pe o navă spațială populată de extraterestri care îi sar deseori în cale prin găuri în pereți și tavan. Povestea amplă a jocului plimbă personajul printr-o multitudine de spații și situații.
- **The Suffering** (2004): jucătorul preia controlul în mod third person unui personaj proaspăt ajuns într-o pușcărie. Închisoarea este în scurt timp infestată de creaturi supranaturale precum umanoizi cu lame ca de sabie în loc de membre. Personajul își petrece majoritatea timpului trecând prin spații claustre și slab iluminate, iar jocul este condus de o poveste.
- **Amnesia: The Dark Descent** (2010): jucătorul intră în pielea unui personaj (în mod first person) prins într-un castel întunecat și aparent pustiu. O mecanică inovatoare a jocului este introducerea conceptului de sănătate psihică a personajului, care se deteriorează când acesta se află în întuneric, asistă la fenomene supranaturale sau are contact vizual cu monștrii ce bântuie castelul.
- **S.T.A.L.K.E.R.: Shadow of Chernobyl** (2007): jucătorul intră în pielea unui personaj (first person) aflat în zona evacuată din jurul faimoasei centrale nucleare ucrainiene, populată acum de căutători de artefacte radioactive, bandiți care atacă acești căutători, și creaturi mutante. În scopul său de a-și descoperi trecutul, personajul călătorește singur de-a lungul regiunii, străbătând ocazional laboratoare științifice subterane sovietice abandonate, în care este întâmpinat de anomalii și creaturi monstruoase.

1.3 Concluzie

Studiind aceste exemple menționate în subcapitolul anterior, se pot observa mai multe trăsături comune care definesc experiența horror într-un joc video.

De exemplu, în toate aceste jocuri, personajul controlat de către jucător se află singur într-un mediu ostil și întunecat. Senzația de singurătate are ca scop să amplifice frica provocată de spațiul necunoscut și neprimitor în care se desfășoară totul.

Un alt element care se regăsește în toate aceste exemple este prezența unei entități sau a unui grup de entități care are ca scop hărțuirea personajului controlat de jucător. În general, acest conflict este unul care defavorizează personajul, oferind entităților negative abilități supranaturale sau limitând abilitățile ofensive ale protagonistului.

O trăsătură care probabil că nu reiese din descrierea exemplelor este sonorizarea acestor jocuri. Marea majoritate a jocurilor din categoria horror își bazează atmosfera pe sunetele percepute de jucător. Pe marginea acestui subiect, se poate face observația că există tehnici considerate clișeice (și implicit lipsite de inspirație) care apar și în filmele categorisite în același gen, precum sunete puternice declanșate brusc în urma unei liniști, corelate cu un eveniment neprevăzut în joc sau film (de exemplu, apariția subită a unui inamic însoțită de un zgomot puternic, în condițiile în care până atunci spațiul era pustiu și liniștea predomina atmosfera).

Concluzionând, pentru realizarea unui joc horror care să își poată atinge scopul, trebuie considerate aspectele descrise în acest subcapitol.

1.4 Contribuții

În scopul realizării acestei lucrări am dezvoltat un joc 3D în Unity, în cadrul căruia jucătorul preia controlul unui personaj într-o perspectivă de tip first person (persoana I - jucătorul observa lumea prin ochii personajului). Acest personaj se va afla într-un spațiu necunoscut mereu, și scopul lui va fi acela de a păstra distanța față de o entitate negativă care îl urmărește cu intenții malițioase pentru un timp setat în prealabil. Motivul pentru care spațiul va fi mereu necunoscut este acela că harta pe care se va afla jucătorul va fi generată aleatoriu pentru fiecare sesiune de joc.

Modul în care am atins acest obiectiv a fost scrierea unor scripturi în limbajul de programare C# și realizarea unor sunete folosind un microfon și Fruity Loops Studio (un DAW - digital audio workstation) pe care le-am aplicat unor obiecte în Unity care, în cele din urmă, rezultă în jocul prezentat mai sus.

2. Fundamente Teoretice

În acest capitol se vor găsi descrieri cu un caracter teoretic al unor concepte care au influențat procesul de dezvoltare al aplicației prezentate în această lucrare, precum conceptul de horror, efectul Doppler și noțiunea de Perlin Noise.

2.1 Conceptul de “Horror”

După cum am menționat și în subcapitolul 1.1, între industria jocurilor video și industria cinematografică există o relație puternică de asemănare. Astfel, multe concepte și tehnici găsite în arta filmului pot fi translate sau aplicate și în procesul de dezvoltare al unui joc.

Atât filmele, cât și jocurile se pot împărți în mai multe categorii, dintre care cea horror este printre cele mai ușor de observat. Un aspect interesant în privința acestui gen este faptul că este definit de modul în care intenționează să afecteze consumatorul. În timp ce unele genuri precum science fiction și western sunt definite de către acțiunile descrise și timpul și spațiul în care acestea au loc, altele, precum comedie și horror sunt definite pe baza unor anumite emoții provocate la nivelul consumatorului.

Reacțiile stârnite de produsele din această categorie sunt atât de importante încât ajung să determine dacă produsul este unul bun sau unul prost; de exemplu, un film horror bun este unul care reușește să sperie cât mai tare privitorul, la fel cum un film horror care nu reușește să atingă acest efect este considerat ca fiind unul prost, iar acest exemplu se poate aplica la fel și la genul “dramă” (un film bun emoționează privitorul, unul prost îl lasă indiferent) precum și “comedie” (produsul care amuză consumatorul este unul bun, cel care eșuează în acest aspect este unul prost). [8]

Totuși, pentru a conceptualiza această noțiune, un prim pas ar fi definirea emoției corelate cu genul horror. În mod evident, este vorba despre frică, unicul scop al produselor din acest gen fiind îngrozirea publicului. Pentru atingerea acestei frici, se poate observa faptul că de-a lungul acestui gen se găsesc unele motive recurente, precum scenariile apocaliptice (“28 Days Later” (film, 2002), “Herbert West Reanimator” (H.P. Lovecraft, 1921), “Dead Island” (joc, 2011) - începând din anii ‘80 și până acum, subiectele care prezintă apocalipse cu zombie au fost o temă horror utilizată aproape până în punctul epuizării - toate generațiile care coexistă în momentul de față cunosc noțiunea de zombie, deci aceasta poate fi considerată a fi clișeică, având în vedere faptul că această temă a apărut în mii de filme, cărți și jocuri) și mituri regionale sau urbane (“The Shining” (Stephen King, 1977), “The Blair Witch Project” (film, 1999), “Half Life” (joc, 1998) - există un număr imens de filme, cărți, și jocuri care tratează teme legate de zone bânuite, fenomene paranormale și întâlniri cu forme de viață extraterestre).

De asemenea, există mai multe modalități prin care un film, o carte sau un joc horror poate șoca consumatorul. De exemplu, în lumea cinematografică circulă noțiunea de “body

horror”, care reprezintă un curent observat în mai multe filme al căror element distinctiv este conținutul grafic explicit ce prezintă mutilarea și desfigurarea personajelor. Modul în care interacționează aceste materiale cu psihicul consumatorului este următorul: proiectele care abordează această cale exagerat de “grafică” (nu este un fenomen limitat la film, se găsesc nenumărate jocuri care abuzează de așa ceva, precum și cărți - de exemplu, romanele în care Sven Hassel prezintă ororile războiului) forțează imaginația cititorului, privitorului sau a jucătorului să reproducă senzațiile prezentate, cauzându-i acestuia neplăceri.

O altă modalitate de inducere a fricii se observă în proiectele ce fac parte din subgenul “psychological horror” (a căror discreție, din punctul meu personal de vedere, merită mai multă apreciere decât materialele atât de grafice încât ajung pe muchia gratuității - evident, aceasta este o temă subiectivă). Acestea au o abordare total diferită a prezentării subiectului ales de autor; în general au un ritm mai puțin agitat, iar teroarea, în loc de a fi o reacție în mare parte fizică, este în mare parte rezultatul imaginației consumatorului. Un exemplu bun în această direcție este filmul “The Blair Witch Project” (1999), în care conținutul grafic extrem este foarte puțin întâlnit: acest film prezintă povestea a trei tineri ce decid să investigheze o pădure despre care circulă zvonul că ar fi un spațiu generos în privința fenomenelor paranormale. Întreaga acțiune este văzută de către audiență prin camerele de filmat pe care le-au luat tinerii după ei pentru a documenta expediția. De-a lungul întregului film, nu apare nici măcar o dată pe peliculă vreo creatură monstruoasă, întreaga atmosferă fiind creată de spațiul în care se află protagoniștii și de reacțiile lor la diferitele evenimente care au loc (de exemplu, găsirea unor figurine din crenguțe spânzurate în copacii din jurul cortului lor). Filmul se încheie abrupt, în momentul în care camera este scăpată pe jos de unul dintre personaje într-o situație deja tensionată. Un alt motiv pentru care acest film este un bun subiect de studiu de caz este modul în care conceptul de “horror” depășește filmul în sine; lansarea acestuia a fost însoțită inițial și de mitul că tinerii din film nu erau actori, ci materialul este veritabil și nu se mai știe nimic despre protagoniști.

Astfel, se ajunge la o altă noțiune care definește o lucrare horror de calitate, și anume modul în care materialul la care a fost expus consumatorul continuă să îl macine pe acesta chiar și după ce a fost sfârșit. Acest efect este cel mai ușor de observat în cazul în cazul publicului tânăr (motivul pentru care există limite inferioare de vârstă pe anumite materiale nu este unul fictiv). De exemplu, un copil de 7-8 ani nu dispune de maturitatea necesară înțelegerii faptului că materialul la care a fost expus este o lucrare de ficțiune - acesta poate suferi traume reale; un alt exemplu de imaturitate (combinată cu traume precum poveștile din bătrâni auzite de la bunici sadici în frageda copilărie) în această privință este existența superstițiilor, și mai ales a oamenilor care cred în ele.

2.2 Perlin Noise

Motivul pentru care am decis să tratez acest subiect în capitolul de fundamente teoretice este unul evident: generarea aleatorie a hărții din aplicația prezentată se bazează în mare parte pe acest concept.

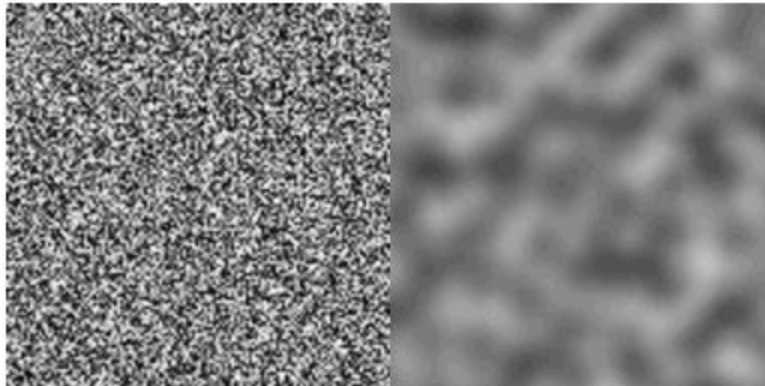


fig. 1: stânga - zgomot incoerent, dreapta - Perlin Noise [5]

Perlin Noise este o metodă de generare a zgomotului gândită și implementată pentru prima dată de Ken Perlin în anul 1983. Motivul pentru care acesta a decis să dezvolte acest procedeu este faptul că la momentul respectiv grafica pe calculator avea un aspect mult prea mecanic, sau sintetic (“machine-like”). În prezent, această tehnică este folosită preponderent în industria efectelor vizuale, mulțumită gradului de realism pe care aceasta o produce. Folosind Perlin Noise, se generează suprafețe de obiecte, foc, fum și nori cu un aspect realist, imitând apariția aparent aleatorie a acestora în natură.

Definiția formală a acestuia, după cum a formulat-o Matt Zucker în lucrarea sa “Perlin Noise Math FAQ” (2001) este următoarea: Perlin Noise este o funcție folosită pentru generarea unui zgomot coerent peste un spațiu, în condițiile în care coerența zgomotului reprezintă faptul că pentru oricare două puncte din acel spațiu valoarea funcției se schimbă cu finețe trecând de la un punct la celălalt, fără să existe discontinuități. [5]

Ce reprezintă, de fapt, o funcție de zgomot este o funcție care pentru niște coordonate într-un spațiu furnizează un număr real cu valoarea cuprinsă între -1 și 1. Aceste funcții pot fi aplicate unui spațiu, indiferent de numărul acestuia de dimensiuni.[5] Astfel, pe lângă aplicarea 2D pe care am utilizat-o în acest proiect, pot fi implementate versiuni și pentru o singură dimensiune (rezultatul va avea forma unui grafic) sau trei dimensiuni (caz în care va exista câte o valoare pentru fiecare punct din acel spațiu).

În cele ce urmează, voi descrie generarea de Perlin Noise după modelul explicației lui Matt Zucker în “The Perlin Noise math FAQ”. [5]

În cazul 2D, funcția va arăta în felul acesta:

$$\text{noise2d}(x, y) = z$$

fig. 2: funcția în cazul 2D

Unde x , y și z sunt numere reale.

Funcția va fi definită pe un spațiu în care pentru fiecare valoare întreagă există o dreaptă paralelă cu axa corespunzătoare într-un sistem cartezian, iar fiecare punct rațional se va afla într-un pătrat format din aceste drepte, după cum se poate observa în desenul următor:

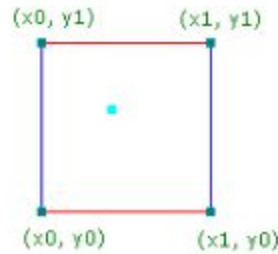


fig. 3: punctul (x, y) cuprins între dreptele corespunzătoare numerelor întregi între care se află cuprins [5]

Următorul pas este definirea unei funcții

$$g(x_{\text{grid}}, y_{\text{grid}}) = (g_x, g_y)$$

fig. 4: definirea funcției g

care pentru fiecare punct din “grid” asignează un vector pseudo-random de lungime 1 în \mathbf{R}^2 , unde proprietatea “pseudo-random” înseamnă că vectorii sunt aparent aleatori, dar pentru același punct va return mereu același vector, iar fiecare vector are o șansă egală de a fi ales.

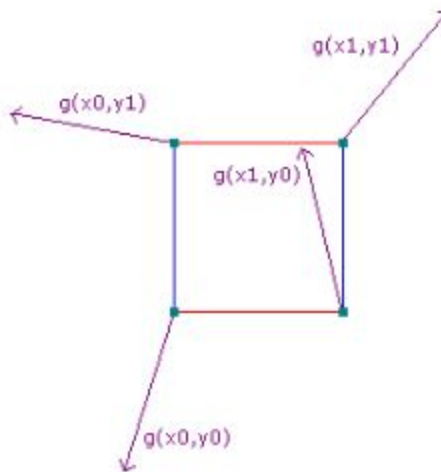


fig. 5: vectorii returnați de funcția g pentru punctele care înconjoară (x, y) [5]

De asemenea, se va genera câte un vector și din fiecare punct al pătratului înspre punctul (x, y) , după cum urmează:

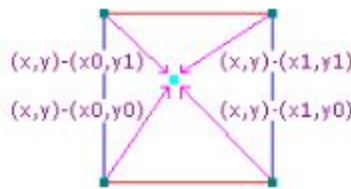


fig. 6: vectorii orientați spre punctul pentru care se calculează valoarea zgomotului [5]

Calculul valorii efective a zgomotului în punctul (x, y) se face calculând influența fiecărui vector pseudo-random în output-ul final, acesta fiind o medie ponderată a acestor influențe.

Matt Zucker a notat aceste valori în următorul mod:

$$\begin{aligned} s &= g(x_0, y_0) \cdot ((x, y) - (x_0, y_0)) , \\ t &= g(x_1, y_0) \cdot ((x, y) - (x_1, y_0)) , \\ u &= g(x_0, y_1) \cdot ((x, y) - (x_0, y_1)) , \\ v &= g(x_1, y_1) \cdot ((x, y) - (x_1, y_1)) . \end{aligned}$$

fig. 7: notația celor patru influențe [5]

unde produsul vectorial se calculează astfel: $(a, b) \cdot (c, d) = ac + bd$.

Reprezentarea grafică a unor valori s, t, u, v (acestea sunt pseudo-random!) arată în felul următor:

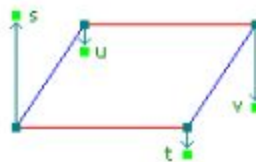


fig. 8: reprezentarea grafică a vectorilor de influență s, t, u, v "ieșind" din planul \mathbb{R}^2 [5]

Valoarea zgomotului reprezintă o medie a acestor vectori de influență.

3. Platforma de Dezvoltare

În momentul de față, piața platformelor de dezvoltare a jocurilor se bucură de un apogeu, oferind o paletă imensă de opțiuni. Aceste opțiuni sunt atât de accesibile acum, încât oricine are o idee și suficientă motivație poate începe dezvoltarea unui joc din confortul propriului apartament. Cele mai populare platforme, precum Unreal Engine, Unity, RPG Maker VX Ace au ajuns într-un stadiu atât de matur, încât pentru crearea unui joc modest nici nu sunt necesare foarte multe cunoștințe de programare. De asemenea, merită menționat faptul că fiecare platformă, pe lângă documentația de bază, este susținută și de câte o comunitate imensă formată din entuziaști. Astfel, în cazul în care un dezvoltator, indiferent de nivelul de experiență, se găsește pus în situația de a se confrunta cu o problemă pe care nu o poate rezolva de unul singur, are șanse foarte mari de a-și găsi întrebarea gata formulată pe forumurile comunităților. Și în cazul în care nu a mai fost nimeni care să întrebe înaintea lui, cu siguranță va găsi pe cineva care s-a confruntat cu o situație similară și e gata să îi răspundă.

3.1 Opțiunile disponibile

În funcție de scopul produsului finit, există posibilitatea alegerii dintre mai multe platforme, printre care și cele enumerate mai sus. În cazul de față, știam de la bun început că doresc să dezvolt un joc a cărui acțiune se va desfășura într-un spațiu tridimensional, și, implicit, aria de platforme dintre care puteam alege s-a restrâns. Ținând cont de faptul că bugetul alocat proiectului era foarte restrâns, am luat în considerare doar platformele gratuite. De asemenea, doream să lucrez cu cele mai noi tehnologii disponibile, și astfel am ajuns pus în fața unei alegeri între două platforme: Unity și Unreal Engine 4. Pentru a lua o decizie cât mai potrivită, am luat în calcul mai mulți factori.

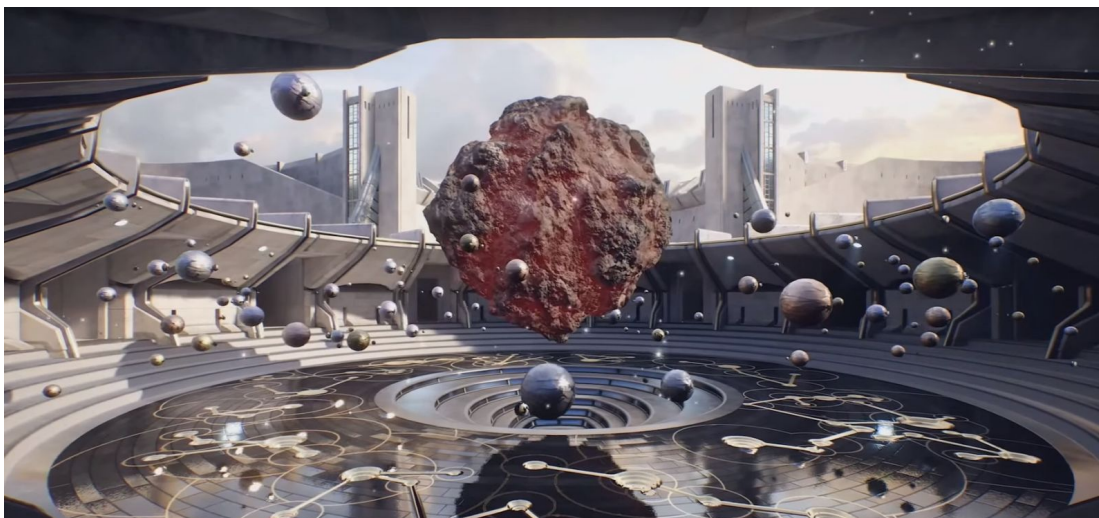


fig. 9: un frame dintr-o aplicație creată în Unreal Engine

În primul rând, trebuie precizat faptul că am mai avut o experiență foarte sumară cu Unity acum câțiva ani ca rezultat al curiozității personale, precum și cu alte platforme precum Game Maker (în 2006) și RPG Maker. Așadar, aveam o foarte vagă idee despre cum ar putea decurge dezvoltarea unui joc. Deși aveam o ușoară preferință pentru Unity, dată fiind experiența relativ recentă și posibilitatea de dezvoltare cross-platform, am fost foarte tentat și de alegerea Unreal Engine, știind că acesta din urmă a fost baza succesului comercial al multor titluri. În alte cuvinte Unity era o opțiune atrăgătoare datorită facilității, iar Unreal Engine era o opțiune atrăgătoare datorită calității aproape inegalabile produsului final. De asemenea, merită menționat faptul că Unreal Engine a ajuns la îndemâna publicului mult mai recent decât Unity (care avea deja un avantaj de câțiva ani în creșterea comunității). Acestea fiind spuse, am început să mă documentez în legătură cu procesul de dezvoltare și cu opțiunile disponibile în scopul realizării ideii mele atât în Unity, cât și în Unreal Engine și am ales într-un final Unity, ținând cont și de presiunea timpului pe care îl aveam la dispoziție.

3.2 Unity

Unity este un game-engine scris în C și C++ lansat inițial în 2005 (versiunea 1.0) al cărui ultim release stabil a avut loc acum câteva luni (martie 2016, versiunea 5.3.4). Unity se bucură de popularitatea pe care o are mulțumită mai multor factori: oferă soluții pentru dezvoltarea aplicațiilor atât 2D cât și 3D ce pot fi rulate pe un număr mare de platforme (Linux, Mac, Windows, Android, iOS, WebGL, BlackBerry, Xbox, Play Station, Nintendo Wii-U & 3DS, etc.). Portabilitatea nu se limitează doar la mediile în care aplicațiile Unity pot fi rulate. Astfel, dezvoltatorii nu sunt forțați să opteze pentru o anumită platformă în vederea dezvoltării, și au posibilitatea de a utiliza atât Windows și MacOS, cât și Linux (Ubuntu - încă în beta).

Unul dintre aspectele care s-au dovedit a fi foarte utile în procesul de dezvoltare al aplicației mele este posibilitatea de a scrie și utiliza scripturi pentru aproape orice, cele mai utilizate limbaje fiind C# și Javascript. Personal, am ales să scriu în C# (în condițiile în care nu mai folosisem niciodată C# până atunci și găsisem chiar și posibilitatea de a folosi python), principalele motive fiind ușurința aplicării unei gândiri POO și faptul că marea majoritate a comunității încurajează utilizarea acestor limbaje pentru a ușura rezolvarea (cu sau fără ajutor extern) a potențialelor probleme ce pot apărea de-a lungul procesului de dezvoltare.

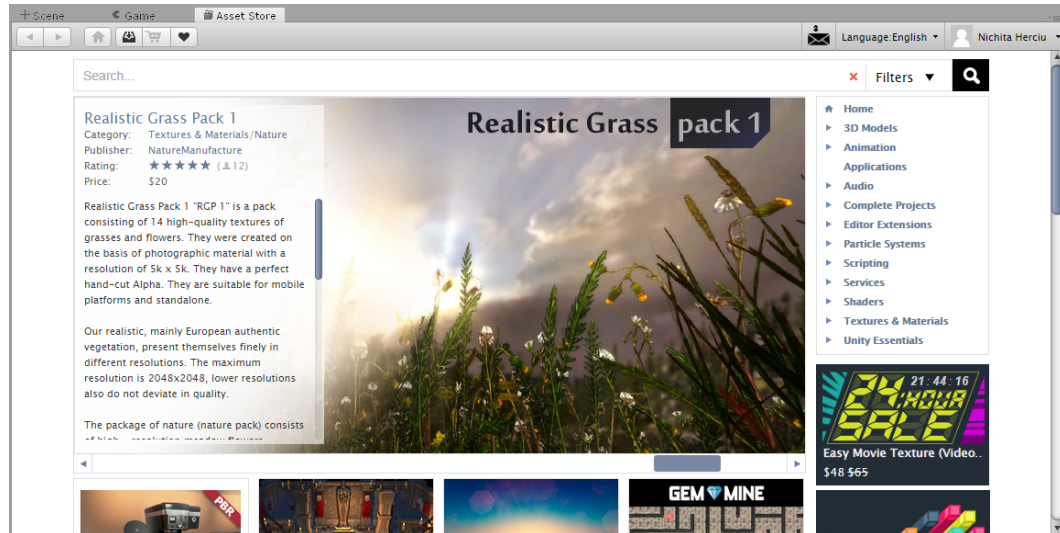


fig. 10: Unity Asset Store

3.3 Assets

Un alt aspect esențial care contribuie la experiența dezvoltării în Unity este prezența Asset Store-ului. Asset Store este o platformă de cumpărare a unor assets create în majoritatea timpului de către comunitate. Un asset poate fi un grup de modele, sunete, scripturi, texturi, etc ce formează un tot unitar ce poate fi importat cu ușurință într-o aplicație Unity. Pentru o mai ușoară exemplificare a noțiunii de asset, poate fi observată importanța asset-urilor în proiectul de față. În această aplicație sunt folosite două asset-uri: FPSController și Staff of Pain.

FPSController este un asset care crează un personaj controlabil de către utilizator cu o perspectivă “first-person” (persoana I - utilizatorul observă universul jocului prin ochii personajului). Acest asset este un pachet care include scripturi pentru detectarea și tratarea input-ului utilizatorului, sunete pentru pași, un collider (un obiect cu proprietăți fizice, folosit în cazul de față pentru a detecta coliziuni cu suprafața podelei) și o cameră atașată acestui collider (camera reprezintă perspectiva din care poate privi un utilizator universul aplicației). Majoritatea asset-urilor, precum acesta, conțin și unul sau mai multe “prefabs” care reprezintă un obiect al jocului prefabricat ce include toate caracteristicile menționate mai sus și le configurează pentru a lucra împreună ca un tot-unitar.

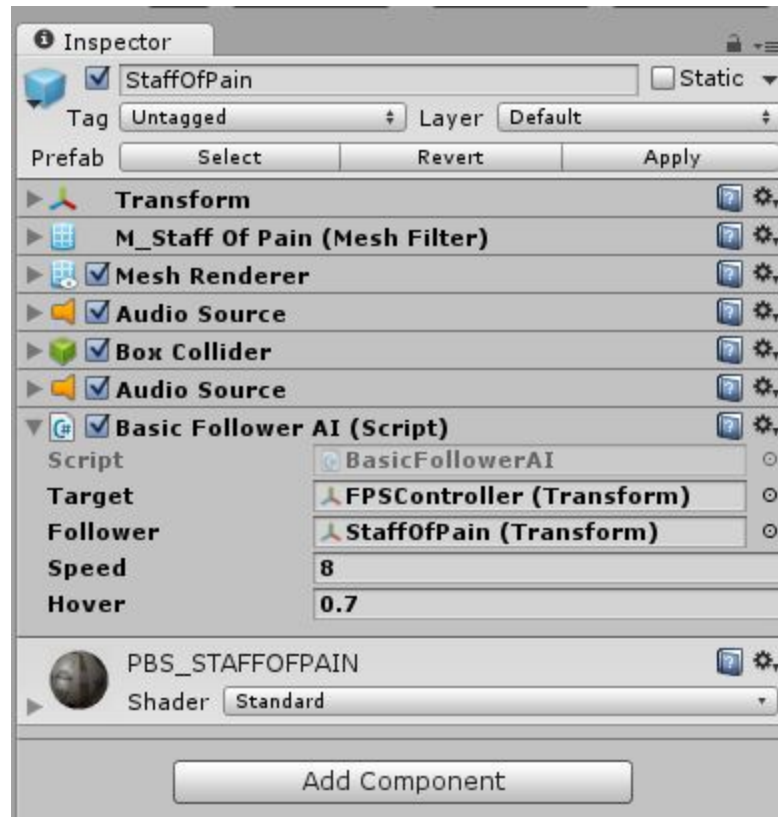


fig. 11: Scriptul pentru AI-ul entității aplicat pe Staff Of Pain prefab

Al doilea asset pe care l-am folosit este Staff Of Pain, pe care l-am găsit gratuit pe Asset Store (<https://www.assetstore.unity3d.com/en/#!/content/48820>), publicat de către Sergi Nicolas, un membru al comunității, pe 5 noiembrie 2015. Acesta conține un model 3D pentru o baghetă și texturile aferente. Am decis să folosesc acest asset deoarece doream un model care să fie în ton cu atmosfera jocului pentru entitatea negativă, și timpul nu mi-ar fi permis să învăț să modelez cu o asemenea precizie profesională. Așadar, am importat acest asset în aplicația mea și i-am adăugat scriptul făcut de mine pentru AI-ul inamicului, sunete și un collider pentru a avea niște proprietăți fizice.

3.4 Sunet

Pentru crearea sunetelor folosite în aplicație am folosit Fruity Loops Studio, un Digital Audio Workstation produs de Image Line. Am ales să folosesc FL Studio deoarece este un tool foarte versatil, pe care l-am folosit la numeroase alte proiecte pe parcursul câtorva ani. Folosind FL Studio, am creat cu ajutorul unor sintetizatoare niște sunete ambientale, și am modificat niște sunete înregistrate de mine pentru a-i oferi entității negative din universul jocului o “voce”.

4. Tehnologii folosite

În acest capitol voi prezenta sumar tehnologiile utilizate în dezvoltarea aplicației prezentate, și anume “End”, jocul realizat în Unity bazat pe scripturi în C#. La sfârșitul fiecărui subcapitol se va găsi o trimitere la bibliografie, unde subiectele pot fi aprofundate.

4.1 Limbajul C#

C# este un limbaj de programare dezvoltat de compania Microsoft, inițial în cadrul proiectului .NET, care mai apoi a fost acceptat ca standard de către Ecma (ECMA-334) și ISO (ISO/IEC 23270:2006).

C#, ca limbaj de programare este unul de scop general, orientat-obiect, bazat pe componente. Fiind un limbaj de scop general, implicit oferă multe variante de rezolvare pentru o paletă variată de probleme. C# permite construirea de aplicații web prin intermediul ASP .NET, aplicații desktop prin WPF (Windows Presentation Foundation) sau aplicații mobile pe Windows Phone. C# are un impact chiar și în piața de cloud computing, fiind în strânse relații cu dezvoltarea platformei Windows Azure. [1]

4.2 Platforma .NET

.NET este o platformă dezvoltată de compania Microsoft ce include limbaje de programare, un runtime și bibliotecile framework-ului, ce permite dezvoltarea unui număr mare de tipuri de aplicații. C# este unul dintre limbajele de programare incluse în .NET, pe lângă Visual Basic, F#, C++ și altele.

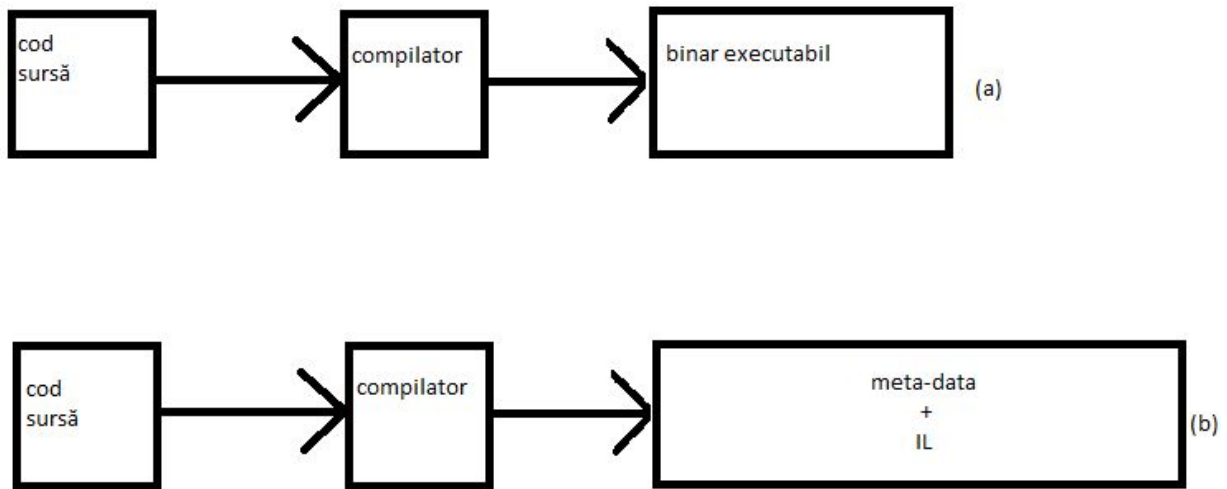


fig. 12:
 (a) compilare tradițională
 (b) CLR

Numele formal al runtime-ului este Common Language Runtime (CLR). Limbajele de programare care vizează acest runtime se compilează într-un IL (Intermediate Language). CLR este, practic, un VM (Virtual Machine) care rulează IL și oferă mai multe servicii, precum gestionarea memoriei, gestionarea excepțiilor, securitate, garbage collection, etc.[1]

4.2 .NET și C# în contextul Unity

Majoritatea platformelor de dezvoltare a jocurilor pun la dispoziția utilizatorului atât un GUI, cât și posibilitatea de a scrie în diverse limbaje de programare diferite elemente sau aspecte, atât ale aplicației, cât și a platformei. Unity nu este o excepție de la această regulă, după cum se va putea observa în continuare.

În legătură cu partea de scriptare din Unity, la începutul dezvoltării oricărui proiect va fi ridicat un semn de întrebare în privința limbajului de programare ce va fi folosit. Unity propune două soluții “oficiale”, și anume C#, și o variantă ajustată pentru Unity a limbajului Javascript. Unity nu impune nicio restricție în privința acestei decizii, permițând utilizarea concomitentă atât a limbajului C#, cât și Javascript, deși acest amestec este privit ca “bad practice” deoarece poate duce la confuzie și conflicte la compilare.

Abordarea recomandată (și prezentă și în proiectul de față) este de a alege un singur limbaj și a-l utiliza oriunde este necesar în proiect. Limbajul ales, după cum reiese și din capitolele anterioare, este C# în cazul de față, din mai multe motive. În primul rând, pentru a clarifica, nu există un limbaj “rău” pentru programare în Unity între Javascript și C#. Motivul

principal al acestei alegeri este faptul că informațiile și conceptele dobândite de dezvoltator înainte de a începe lucrul în Unity se transferă cu ușurință în C# (de exemplu, gândirea orientată-obiect). C# are de asemenea o poziție importantă în istoria creșterii Unity, fiind strâns legat de .NET, .NET fiind folosit în Unity sub numele de “Mono”, și semănând cu C++, care a fost folosit intens în dezvoltarea jocurilor.[2]

5. Prezentarea aplicației

În acest capitol voi prezenta aplicația propriu-zisă, voi explica procesul de dezvoltare al acesteia, problemele întâmpinate, și soluțiile găsite la aceste probleme. În subcapitolele ce urmează, voi folosi exemple bazate pe bucăți de cod, care se vor remarca față de restul textului printr-un font diferit. Exemplu:

```
/*Aceasta este o linie de cod
Aceasta este o altă linie de cod
Toate aceste linii de cod au fost într-un bloc comentat*/
```

fig. 13: un exemplu de formatare a codului sursă în acest document

5.1 Specificații funcționale

Aplicația reprezintă un joc 3D definit, în mare, de două aspecte:

- Un spațiu generat aleatoriu, cu o atmosferă menită să neliniștească utilizatorul;
- O entitate cu un caracter negativ de care utilizatorul trebuie să se ferească.

Astfel, utilizatorul va prelua controlul unui personaj fără un nume, fără o poveste, prin ochii căruia va percepe universul jocului, într-un spațiu mereu necunoscut (la începerea fiecărui joc va fi generată o hartă aleatorie).

Pus în această situație, utilizatorul ar putea să nu intuiască în primele momente de joc scopul acestuia, dar totul se clarifică în momentul în care entitatea începe să se apropie de personaj. Utilizatorul va înțelege că este cazul să evite această entitate datorită efectelor audio emise de către aceasta. Pentru a adăuga mai multă tensiune jocului, mediul pe care îl va străbate personajul va fi unul întunecat, roșiatic, cu un cer sângieru și o topografie ieșită din comun (munți, cratere, etc.). De asemenea, la inițierea jocului va fi afișat pe ecran și un timer care va descrește, ce simbolizează numărul de secunde pe care jucătorul trebuie să le mai supraviețuiască pentru a câștiga jocul.

Întreaga atmosferă a jocului se bazează pe presiunea psihologică la care este supus jucătorul în timpul expunerii la aplicație. Cei mai importanți factori ce contribuie la construirea acestei presiuni sunt:

- senzația inițială de singurătate
- liniștea inițială (până să se afle în proximitatea entității negative, personajul își va auzi doar pașii)
- mediul sterp, cu aspect ostil (cer, munți și cratere în diverse tonalități de roșu, de la roșu aprins, la negru)

- prezența entității negative (sunete inumane, aspect înfricoșător, perseverență în hărțuirea personajului)

Entitatea negativă va fi rezultatul aplicării unui script ce simulează o inteligență artificială pe un model 3D cu un aspect deranjant. Mulțumită scriptului, entitatea va avea următoarele caracteristici:

- va urmări fără pauză personajul, indiferent dacă a existat sau nu contact vizual între cei doi
- viteza cu care se deplasează variază în funcție de distanța față de personaj (Scriptul dispune de niște parametri care definesc distanțele la care să se modifice viteza de călătorie. Scopul acestei schimbări de viteză este de a ține mereu entitatea la o distanță potrivită față de personaj - viteza sa normală va fi mai mică decât cea a personajului în fugă, dar de la un anumit prag de distanță viteza i se poate, dubla, tripla, sau chiar înzecă pentru a ajunge victima din urmă)
- va avea abilitatea de a se teleporta într-o poziție pseudo-aleatorie față de cea curentă, dacă condițiile sunt îndeplinite (dacă entitatea se află într-un range față de personaj care să-i permită să meargă cu viteza minimă, aceasta va avea o probabilitate mică de a avansa instantaneu o distanță mare înspre personaj, puțin orientat spre stânga sau spre dreapta personajului. Aceasta se întâmplă pentru a oferi un caracter neprevăzut entității negative, iar orientarea spre stânga sau spre dreapta -aleasă și ea tot aleatoriu- are ca scop prelungirea jocului, sau creșterea șansei de supraviețuire a jucătorului - astfel se evită o coliziune produsă în momentul teleportării între entitate și jucător, coliziune care ar încheia jocul)

Generarea spațiului este realizată utilizând perlin noise, procedeu ce va fi explicat în subcapitolele ce urmează.

5.2 Arhitectura aplicației

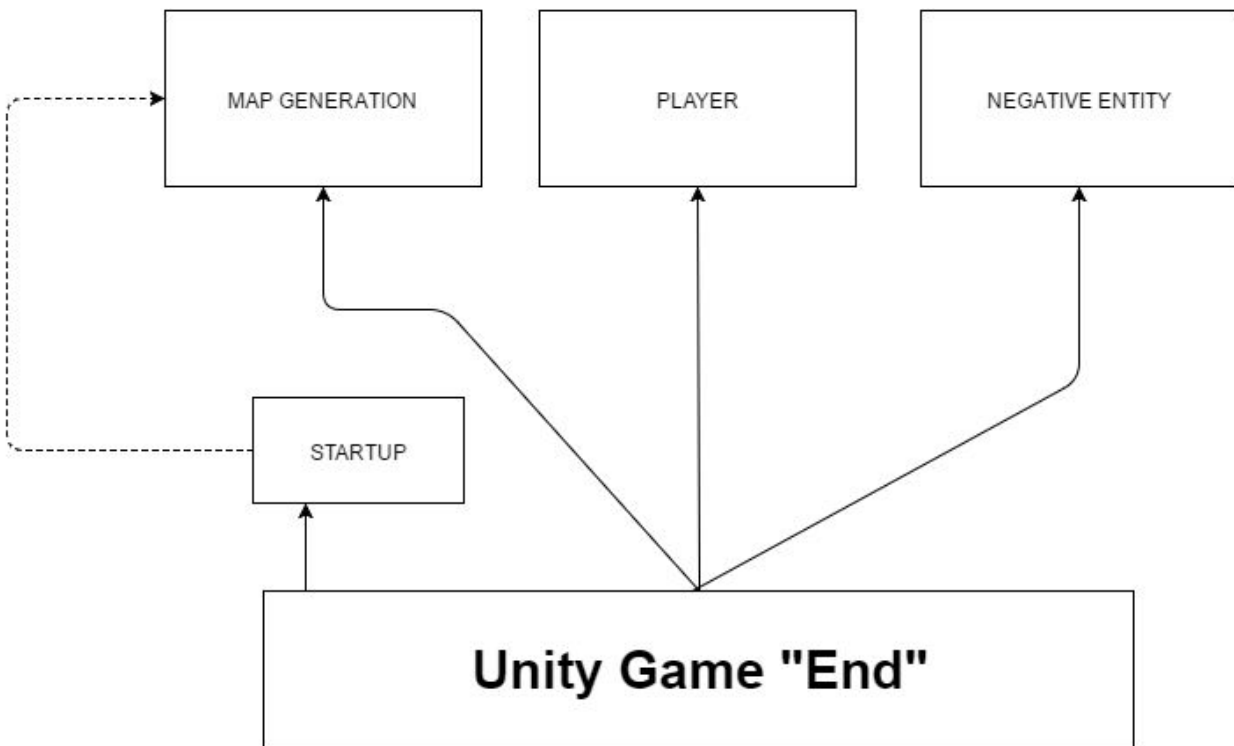


fig. 14: O diagramă simplificată a aplicației

Pentru o reprezentare fidelă a arhitecturii unui proiect dezvoltat în Unity, trebuie clarificat faptul că aceste aplicații nu sunt orientate-obiect, ci bazate pe componente. În scopul înțelegerii noțiunii de component în Unity, se va defini întâi conceptul de GameObject.

GameObject este cel mai important tip de obiect prezent în Unity. Orice element al unei aplicații este un GameObject. Totuși, un GameObject nu poate face nimic de unul singur, ci are nevoie niște proprietăți pentru a putea deveni ceva. Aceste proprietăți sunt în componente. Astfel, un GameObject poate fi privit ca un container pentru mai multe componente, în care pot fi păstrate sunete, lumini, animații, script-uri, etc.

În mare, aplicația poate fi privită ca o îmbinare a patru elemente ce definesc jocul, sau patru GameObjects principale. Intuitiv, primele trei pot fi determinate ușor:

- Un GameObject corespondent hărții sau a spațiului în care se petrece acțiunea;
- Un GameObject ce reprezintă jucătorul;
- Un GameObject ce reprezintă entitatea negativă.

Al patrulea element de o importanță majoră în aplicația prezentată este un GameObject diferit. Acesta nu poate fi văzut de către jucător, și personajul nu poate nici să interacționeze în mod direct cu el. Este vorba despre un GameObject care orchestrează jocul. La începutul jocului, va genera o hartă și va declanșa un timer.

Vom detalia acum rolul fiecărui element dintre cele menționate mai sus.

5.2.1 Harta

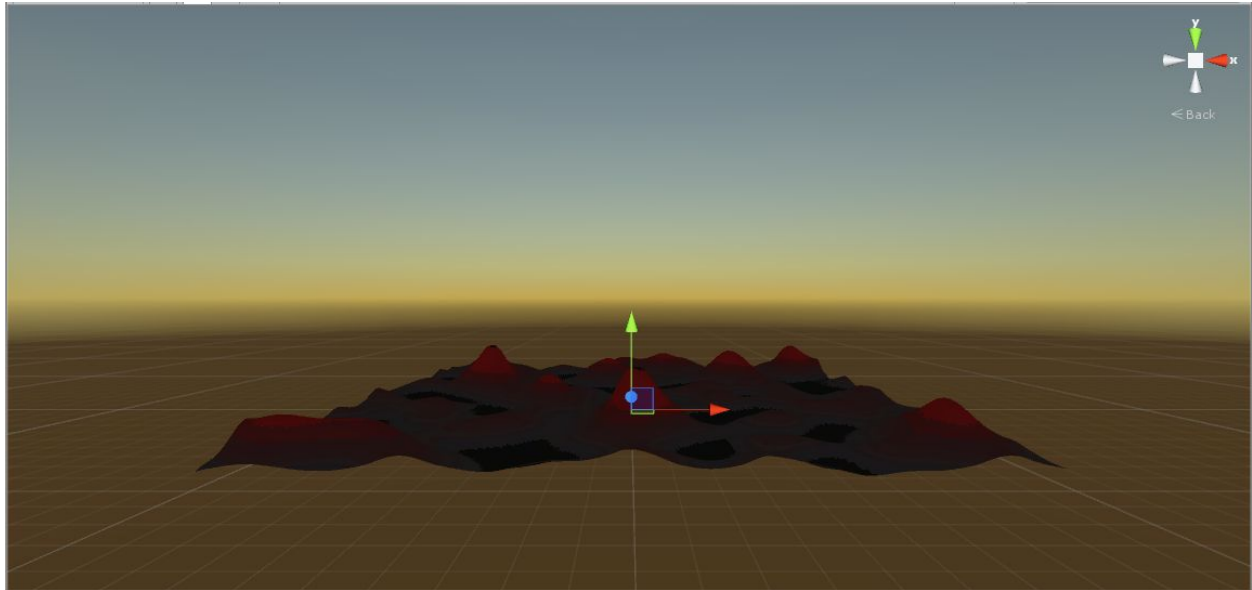


fig. 15: Un exemplu de hartă generată

În privința hărții jocului, în explicația anterioară a fost omis un detaliu semnificativ în scopul înțelegerii arhitecturii de bază a aplicației: pentru obținerea hărții nu este folosit un singur GameObject, ci două; unul se ocupă de generarea hărții (MapGenerator), celălalt (Mesh) reprezintă un mesh bazat pe harta generată în prealabil. Jucătorul va avea contact direct doar cu al doilea dintre aceste două obiecte.

Am specificat mai sus că al doilea GameObject utilizat pentru reprezentarea hărții este un mesh. Un mesh este un grup de triunghiuri aranjate într-un spațiu tridimensional astfel încât crează impresia unui obiect solid. Un triunghi este definit de trei puncte sau vârfuri. Noțiunii de mesh îi corespunde și o clasă numită sugestiv “Mesh”. În această clasă, triunghiurile sunt stocate printr-un vector ce conține toate vârfurile; fiecare triunghi este definit de câte trei elemente ale vectorului. Astfel elementele 0, 1 și 2 ale vectorului ar reprezenta primul triunghi, 3, 4 și 5 ar reprezenta al doilea triunghi, etc.[3]

MapGenerator este un GameObject ce conține două componente esențiale: un script de generare a hărții și un script de afișare a acesteia.

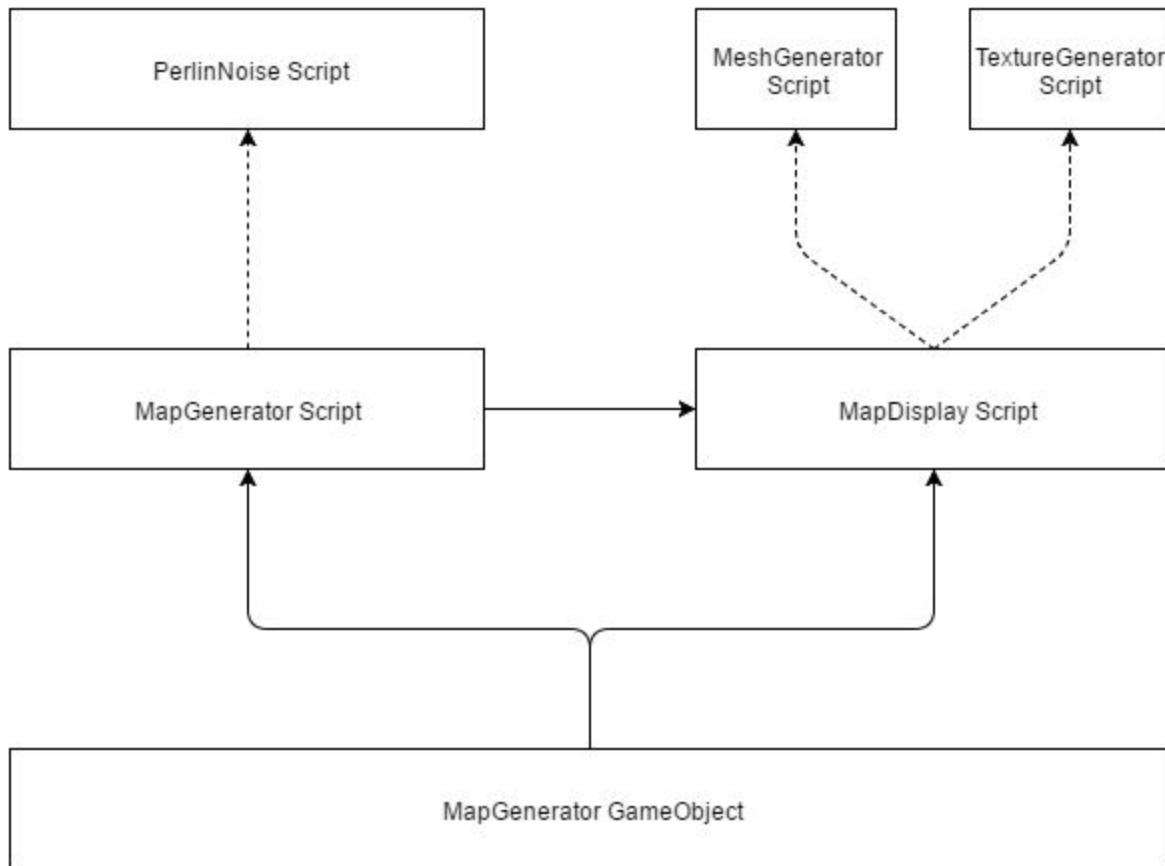


fig. 16: O diagramă ce reprezintă structura obiectului MapGenerator, dezvoltată după ideea lui Sebastian Lague (<https://www.youtube.com/user/Cercopithecان>)

Pentru a înțelege modul în care se generează harta, vom aminti variantele în care s-ar fi putut realiza acest aspect. În cele ce urmează, prin hartă ne vom referi la un vector bidimensional de biți (care, în mod intuitiv, pot lua doar o valoare dintre {0,1}).

Având în vedere că scopul principal este generarea aleatoare a unei hărți de joc, prima variantă la care s-ar putea gândi oricine ar fi asignarea aleatorie a valorilor fiecărui bit. Algoritmul ar fi unul trivial, și ar putea fi condensat în câteva linii de cod:

```

For i in array.length{
  For j in array.Width{
    array[i][j] = Random();
  }
}
  
```

fig. 17: o sugestie de implementare a unui algoritm random noise

Un asemenea algoritm, deși ar fi foarte ușor de implementat, prezintă o problemă ce îl lasă inutilizabil. Fiecare bit din tablou va avea o valoare absolut aleatorie, fără să existe vreo

legătură între valoarea bitului curent și valorile vecinilor săi. Cuvântul care descrie această situație este "incoerență". Dacă fiind această problemă a coerenței, se poate observa că o hartă generată absolut aleatoriu nu ar putea fi interpretată ca grupuri de biți de aceeași valoare care să constituie elemente ce ar putea reprezenta mai târziu munți, dealuri, văi, etc.

Așadar, în scopul atingerii obiectivului inițial, vom renunța la algoritmul care asignează valori aleatoare independente vectorului ce reprezintă harta. În scopul simulării realismului, ne vom întoarce la modul în care am definit harta și vom face o mică ajustare: în locul unui vector bidimensional de biți, vom considera un vector bidimensional de numere (raționale, pentru acest exemplu). Astfel, fiecare valoare va reprezenta "înălțimea" elementului în situația transpunerii tabloului într-un spațiu tridimensional. De exemplu, pentru situația propusă mai devreme a generării unei hărți ce conține forme topografice diverse, precum munți, dealuri și văi, o hartă de biți, chiar și coerentă, nu va fi suficientă pentru modelarea spațiului deoarece nu am putea exprima diferența de înălțime între un vârf de munte și un punct dintr-o vale. În cazul folosirii unui vector de valori raționale, deja se poate contura tipul de spațiu descris mai sus.

Pentru a rezolva problema coerenței, vom recurge la un alt algoritm de generare a zgomotului. Perlin Noise este un algoritm creat de Ken Perlin menit să genereze texturi de zgomot coerent; această coerență semnificând lipsa discontinuităților în textura generată. Acest algoritm este unul foarte popular în industria jocurilor și a filmelor, fiind folosit în diverse scopuri, precum simularea focului sau a unor nori.

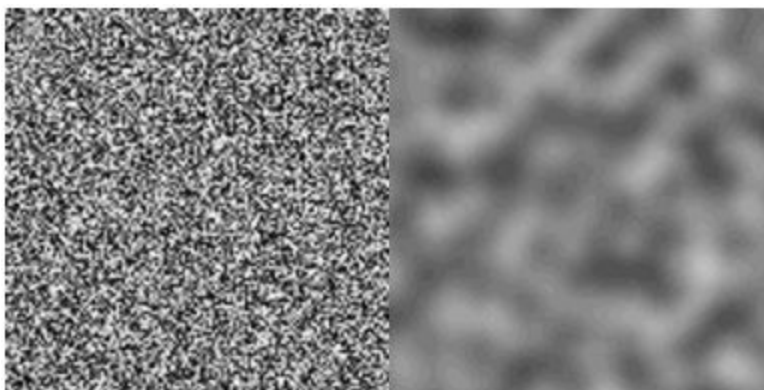


fig. 18: stânga - zgomot incoerent, dreapta - zgomot coerent [5]

După cum se poate observa și în imagine, în cazul zgomotului coerent tranzițiile de la o valoare la alta sunt făcute treptat, fără să existe treceri bruște.

Din fericire, în Unity există deja o implementare a acestui algoritm: este pusă la dispoziția dezvoltatorilor funcția `PerlinNoise` din colecția `Mathf`. Această funcție primește ca parametri două valori de tip `float`, ce reprezintă coordonatele x și y ale unui punct "sample" din plan, și returnează o valoare `float` între 0.0 și 1.0. Fiind o funcție pseudo-random, pentru același sample va returna mereu aceeași valoare.

În cazul aplicației de față scriptul MapGenerator folosește scriptul PerlinNoise (a se observa în diagramă). În PerlinNoise se găsește o clasă cu același nume, care are o singură metodă: GenerateNoiseMap.

Această metodă primește ca parametri înălțimea și lățimea hărții (int mapWidth, int mapHeight) și o variabilă de tip float numită scale, care dictează finețea hărții generate. Metoda returnează un vector bidimensional de tip float în care se află valorile emise.

```
public static float[,] GenerateNoiseMap(int mapWidth, int
mapHeight, float scale)
{
    System.Random prng = new System.Random();
    float mainRandomX = prng.Next(-1000, 1000);
    float mainRandomY = prng.Next(-1000, 1000);

    float[,] noiseMap = new float[mapWidth, mapHeight];

    if (scale <= 0)
    {
        scale = 0.0001f;
    }

    for (int y = 0; y < mapHeight; y++)
    {
        for (int x = 0; x < mapWidth; x++)
        {
            float detailRandomX = prng.Next(-1, 1);
            float detailRandomY = prng.Next(-1, 1);
            float sampleX = x / scale + mainRandomX +
detailRandomX/scale ;
            float sampleY = y / scale + mainRandomY +
detailRandomY/scale ;

            float perlinValue = Mathf.PerlinNoise(sampleX,
sampleY);
            noiseMap[x, y] = perlinValue;
        }
    }

    return noiseMap;
}
```

fig. 19: codul metodei de generare a hărții de zgomot

După cum se poate observa în cod, primul lucru care se petrece este definirea unui generator de numere pseudo-random. Odată definit acest generator, stabilim valorile a două variabile numite `mainRandomX` și `mainRandomY`, ambele aparținând intervalului $[-1000, +1000]$. Aceste două variabile nu își vor schimba niciodată valoarea de-a lungul întregului proces de generare al unei hărți. În cele ce urmează, se declară un vector bidimensional gol în care urmează să fie stocate valorile generate și se verifică dacă variabila `scale` este mai mică sau egală cu 0, iar în cazul în care condiția este îndeplinită, `scale` primește o valoare pozitivă foarte apropiată de zero. Pasul următor este parcurgerea element cu element a vectorului. În timpul acestei parcurgeri, la fiecare pas se vor defini două noi valori aleatoare, `detailRandomX` și `detailRandomY`, în intervalul $[-1, 1]$, de data aceasta. Se construiesc apoi, tot pentru fiecare element, cele două sample-uri necesare pentru apelarea funcției `PerlinNoise`. Așadar, pentru fiecare element din vector se generează două sample-uri corespundente coordonatei `x` și respectiv coordonatei `y`. În aceste sample-uri vom găsi:

- Valorile coordonatelor curente, raportate la `scale`;
- Valorile `mainRandom` - acestea dictează formele principale ale reliefului hărții, fiind consistente între iterații (au fost definite o singură dată);
- Valorile `detailRandom`, raportate la `scale` - aceste valori sunt alese aleatoriu la fiecare pas, pentru a genera detalii ale reliefului ce pot fi comparate cu stânci, gropi, etc.

Odată construite sample-urile, se generează o valoare `perlin` din acestea, care este stocată mai apoi în hartă la coordonatele corespunzătoare.

După ce algoritmul iterează prin toate elementele, returnează harta generată.

Având în vedere faptul că funcția `PerlinNoise` generează doar un array de valori de tip `float`, e timpul să aruncăm o privire de ansamblu. După cum spuneam mai devreme, scriptul `MapGenerator` folosește rezultatul returnat de funcția prezentată mai devreme.

Se observă cu ușurință că `MapGenerator` este de fapt o clasă derivată din clasa `MonoBehaviour` (o clasă de bază din care putem deriva orice script). Precum majoritatea claselor implementate în cadrul acestui proiect, ea dispune de mai multe variabile, majoritatea acestora fiind publice. Astfel, ele pot fi manipulate prin intermediul interfeței editorului Unity. În `MapGenerator`, printre altele, vom găsi:

```
public int mapWidth;
public int mapHeight;
public float meshHeightMultiplier;
public AnimationCurve meshHeightCurve;
public float noiseScale;
```

fig. 20: variabilele clasei MapGenerator

Intuitiv, vom descoperi că `mapWidth`, `mapHeight` reprezintă dimensiunile hărții, iar `noiseScale` determină finețea acesteia. După cum era de așteptat, aceste variabile sunt folosite la

apelul funcției PerlinNoise. Observăm doi parametri noi, și anume meshHeightMultiplier, și meshHeightCurve, ce permit controlul asupra mesh-ului ce urmează a fi generat din harta de zgomot.

Tot între variabilele clasei se va găsi ceva nou:

```
public TerrainType[] regions;
```

fig. 21: definiția tabloului de regiuni

Acesta este un vector unidimensional de tip TerrainType care stochează regiuni. Pentru a înțelege ce reprezintă aceste regiuni, vom arunca acum o privire și la definiția structurii TerrainType, care se găsește tot în scriptul MapGenerator:

```
public struct TerrainType
{
    public string name;
    public float height;
    public Color colour;
}
```

fig. 22: definiția structurii TerrainType

Așadar, structura TerrainType reține un nume, o înălțime, și o culoare. Numele unui tip de regiune este suficient de sugestiv; înălțimea este o valoare de tip float care precizează de la ce înălțime pe hartă vom găsi această regiune, iar culoarea reprezintă chiar culoarea pe care o vom găsi și în joc în acest tip de regiune.

Astfel, în MapGenerator vom avea un vector de regiuni, lăsând posibilitatea modificării totale a aspectului cromatic al spațiului de joc.

```
void OnValidate()
{
    if (mapWidth < 1)
    {
        mapWidth = 1;
    }
    if (mapHeight < 1)
    {
        mapHeight = 1;
    }
}
```

fig. 23: metoda OnValidate

Mai este observabil faptul că în MapGenerator sunt prezente două metode, și anume “OnValidate” și “generateMap”. OnValidate - metodă moștenită din MonoBehaviour, după cum sugerează și numele, validează niște parametri. În cazul de față se validează dimensiunile hărții: dacă una dintre dimensiuni este mai mică decât 1, primește valoarea 1. Astfel, se evită situația în care harta de zgomot ar fi definită eronat.

Următorul aspect este și ultimul pe care îl vom studia din acest script; este vorba despre metoda generateMap, trăsătura esențială a acestui script.

Codul metodei:

```
public void generateMap()
{
    float[,] noiseMap = PerlinNoise.GenerateNoiseMap(mapWidth, mapHeight,
noiseScale);
    Color[] colourmap = new Color[mapWidth * mapHeight];
    for (int y = 0; y < mapHeight; y++)
    {
        for (int x = 0; x < mapWidth; x++)
        {
            float currentHeight = noiseMap[x, y];
            for (int i = 0; i < regions.Length; i++)
            {
                if (currentHeight < regions[i].height)
                {
                    colourmap[y * mapWidth + x] = regions[i].colour;
                    break;
                }
            }
        }
    }

    MapDisplay display = FindObjectOfType<MapDisplay>();
    display.DrawMesh(MeshGenerator.GenerateTerrainMesh(noiseMap,
meshHeightMultiplier, meshHeightCurve),
TextureGenerator.TextureFromColourMap(colourmap, mapWidth, mapHeight));
}
```

fig. 24: codul metodei generateMap()

Primul lucru care se petrece în momentul apelului acestei metode este generarea hărții de zgomot cu ajutorul funcției GenerateNoiseMap din scriptul PerlinNoise, cu parametrii pe care îi găsim definiți în corpul clasei: mapWidth, mapHeight și noiseScale.

Următorul element pe care îl definim este o hartă de culori, reprezentată printr-un array unidimensional. Motivul pentru care nu folosim un array bidimensional în acest scop se va vedea în scriptul TextureGenerator.

În cele ce urmează, se vor parcurge harta de zgomot și tabloul de regiuni pentru a construi vectorul de culori. În mod intuitiv, harta va fi parcursă prin cei doi indici corespondenți înălțimii și lățimii punctului curent în hartă, iar regiunile vor fi și ele parcurse în ordinea în care

au fost stocate. Deși vectorul de culori are o singură dimensiune, se dorește replicarea hărții de zgomot în culori în interiorul acestuia, așadar el va fi parcurs ca atare: se va înmulți “înălțimea” curentă în hartă cu înălțimea maximă pentru a simula un indice al rândului în matrice, la care se adună pur și simplu indicele coloanei ($y * \text{mapWidth} + x$).

Având acum atât harta de zgomot, cât și cea de culori, tot ce rămâne de făcut este crearea unui mesh și a unei texturi pentru acest mesh, și afișarea lui. Pentru această sarcină am creat `MapDisplay`, o altă clasă derivată și ea tot din `MonoBehaviour`.

```
public Renderer textureRenderer;
public MeshFilter meshFilter;
public MeshRenderer meshRenderer;
public MeshCollider meshCollider;
```

fig. 25: variabilele clasei MapDisplay

În această clasă se definesc patru variabile:

- `textureRenderer`: un obiect de tip `Renderer` - `renderer` este folosit pentru afișarea pe ecran a oricărui obiect;
- `meshFilter`: oferă un mesh unui `MeshRenderer`;
- `meshRenderer`: un `MeshRenderer`, intuitiv, afișează pe ecran un mesh;
- `meshCollider`: este un obiect de tip `MeshCollider`, acesta va fi baza interacțiunii fizice între hartă și entități (jucătorul și entitatea negativă).

Pe lângă aceste variabile, în `MapDisplay` este definită și o metodă numită `DrawMesh` căreia îi revine sarcina de a desena mesh-ul:

```
public void DrawMesh (MeshData meshData, Texture2D texture)
{
    meshFilter.sharedMesh = meshData.CreateMesh();
    meshRenderer.sharedMaterial.mainTexture = texture;
    meshCollider.sharedMesh = meshFilter.sharedMesh;
    meshCollider.enabled = true;
}
```

fig. 26: codul metodei DrawMesh

Aceasta primește ca parametri datele ce definesc mesh-ul stocat într-o clasă numită `MeshData`, și o textură. Vom detalia imediat și modul în care se obțin acești parametri. În cadrul metodei sunt setate corespunzător mesh-ul și colliderul acestuia, după ce mesh-ul este generat la apelul funcției `CreateMesh`.

În scriptul MeshGenerator se găsesc definițiile a două clase, și anume MeshGenerator și MeshData. În linii mari, MeshGenerator se va ocupa de generarea datelor ce vor defini un mesh rezultat din harta de zgomot creată anterior, returnând un obiect MeshData.

Vom arunca o privire peste codul celor două clase.

```
public class MeshData
{
    public Vector3[] vertices;
    public int[] triangles;
    public Vector2[] uvs;
    int triangleIndex;

    public MeshData(int meshWidth, int meshHeight)
    {
        vertices = new Vector3[meshWidth * meshHeight];
        uvs = new Vector2[meshWidth * meshHeight];
        triangles = new int[(meshWidth - 1) * (meshHeight - 1) * 6];
    }
    public void AddTriangle (int a, int b, int c)
    {
        triangles[triangleIndex] = a;
        triangles[triangleIndex + 1] = b;
        triangles[triangleIndex + 2] = c;
        triangleIndex += 3;
    }

    public Mesh CreateMesh()
    {
        Mesh mesh = new Mesh();
        mesh.vertices = vertices;
        mesh.triangles = triangles;
        mesh.uv = uvs;
        mesh.RecalculateNormals();
        return mesh;
    }
}
```

fig. 27: codul clasei MeshData

MeshData, după cum am mai menționat, este o clasă menită să modeleze datele necesare creării unui Mesh. Precum celelalte clase prezentate până acum, și aceasta are un grup de variabile definite în corpul ei:

- vertices: un tablou unidimensional de obiecte de tip Vector3 - Vector3 este un tip de obiect în Unity ce modelează un vector într-un spațiu tridimensional;
- triangles: un tablou unidimensional ce reține triunghiurile după vârfuri;
- uvs: UVs în Unity este o notație a unui sistem de coordonate carteziene într-un spațiu 2D, cu originea în stânga jos; folosim notația UV deoarece XYZ sunt deja

folosite în definirea spațiului și astfel se evită crearea confuziilor. În MeshData există un tablou unidimensional de obiecte de tip Vector2 pentru acești vectori;

- triangleIndex: numără vârfurile ce compun triunghiuri - această variabilă va crește cu 3 pentru fiecare triunghi adăugat.

Aruncând o privire peste codul clasei, se observă că aceasta are trei metode: MeshData - un constructor, AddTriangle - o metodă pentru adăugarea triunghiurilor și CreateMesh - metodă utilizată pentru crearea și returnarea unui Mesh din triunghiurile reținute.

Constructorul acestei clase primește ca parametri două valori de tip int; acestea reprezintă dimensiunile hărții. În cadrul acestei metode se inițializează tablourile vertices, uvs și triangles. Nu există niciun mister în inițializarea tablourilor vertices și uvs - dimensiunile acestora sunt pur și simplu produsul celor două dimensiuni ale hărții. În cazul tabloului de triunghiuri, lucrurile stau puțin diferit, totuși. În cod se observă că numărul de triunghiuri rezultate din cele două dimensiuni este $(\text{lățime}-1) * (\text{înălțime}-1) * 6$. Pentru a înțelege această formulă, vom folosi un exemplu ușor. Vom considera o hartă de dimensiune $2*2 \rightarrow$ patru elemente. Aceste elemente vor reprezenta vârfurile triunghiurilor, așadar le vom numera de la 0 la 3:

0	1
2	3

În cazul acestei hărți, triunghiurile ce compun mesh-ul vor fi următoarele: $(0 \rightarrow 3, 3 \rightarrow 2, 2 \rightarrow 0)$ și $(3 \rightarrow 0, 0 \rightarrow 1, 1 \rightarrow 3)$. Se observă cu ușurință faptul că triunghiurile sunt luate în sens invers trigonometric. Așadar, în calculul numărului de triunghiuri înmulțim (lățime - 1) cu (înălțime - 1) deoarece extremitățile hărții vor fi deja incluse în triunghiuri existente. Motivul pentru care se înmulțește acest produs apoi cu 6 este următorul: în fiecare "pătrat" (un pătrat este un set de patru vârfuri vecine într-o hartă, precum exemplul de mai sus) vor fi câte două triunghiuri a câte trei vârfuri ($2*3 = 6$).

Metoda de adăugare a triunghiurilor este o funcție de tip void ce primește ca parametri trei valori de tip integer, ce reprezintă vârfurile unui triunghi. În cadrul acestei funcții vârfurile sunt adăugate în vector în ordine, iar variabila triangleIndex este incrementată cu 3.

Metoda de creare a mesh-ului este o funcție ce returnează un obiect de tip mesh care nu primește parametri, deoarece toate datele sunt stocate în variabilele clasei. În cadrul acestei metode este inițializat un mesh nou, căruia îi sunt setate vârfurile (tabloul vertices răspunde acestei cerințe), triunghiurile (tabloul triangles) și UVs (tabloul uvs). Mesh-ul abia creat este apoi returnat.

În cele ce urmează, vom studia modul în care această clasă este folosită dintr-o perspectivă mai cuprinzătoare. În altă ordine de idei, vom studia clasa MeshGenerator și modul în care aceasta utilizează clasa MeshData.

```

public static class MeshGenerator {

    public static MeshData GenerateTerrainMesh (float[,] heightmap, float
heightMultiplier, AnimationCurve heightCurve)
    {
        int width = heightmap.GetLength(0);
        int height = heightmap.GetLength(1);
        float topLeftX = (width - 1) / -2f;
        float topLeftZ = (height - 1) / 2f;
        MeshData meshData = new MeshData(width, height);
        int vertexIndex = 0;

        for (int y = 0; y < height; y++)
        {
            for (int x = 0; x < width; x++)
            {
                meshData.vertices[vertexIndex] = new Vector3(topLeftX + x,
heightCurve.Evaluate(heightmap[x, y]) * heightMultiplier, topLeftZ - y);
                meshData.uvs[vertexIndex] = new Vector2(x / (float)width, y /
(float)height);

                if (x < width - 1 && y < height - 1)
                {
                    meshData.AddTriangle(vertexIndex, vertexIndex + width + 1,
vertexIndex + width);
                    meshData.AddTriangle(vertexIndex + width + 1, vertexIndex,
vertexIndex + 1);
                }

                vertexIndex++;
            }
        }
        return meshData;
    }
}

```

fig. 28: codul sursă al clasei MeshGenerator

Se observă faptul că această clasă nu stochează date deloc, ci doar definește o metodă de generare a unui Mesh corespunzător hărții de zgomot. Metoda GenerateTerrainMesh primește ca parametri harta, și doi parametri care au mai fost întâlniți în cadrul clasei MapGenerator, care în sfârșit primesc o utilitate: este vorba despre parametrul de tip float heightMultiplier, și parametrul de tip AnimationCurve heightCurve. Modul în care funcționează acești parametri este unul intuitiv, dar se bazează pe înțelegerea noțiunii de mesh. În cazul aplicației de față, la un moment t va exista o hartă de zgomot, un set de regiuni, o hartă de culori și un mesh format dintr-o mulțime de triunghiuri în același plan. Pentru a crea efectul dorit, și anume un teren cu

diverse forme de relief, vârfurile acestor triunghiuri vor trebui să primească valori și pe a treia axă într-un sistem tridimensional, și anume profunzimea. Această a treia valoare va fi calculată pornind de la valoarea zgomotului din punctul respectiv. Dacă ea ar lua pur și simplu valoarea zgomotului, harta ar avea un aspect aplatizat, și de aceea apare parametrul `heightMultiplier`. După cum sugerează și numele, această valoare va fi înmulțită cu numărul stocat în harta de zgomot, alterând astfel harta finală. Parametrul `heightCurve` este folosit pentru uniformizarea hărții.

Primul pas din această metodă este determinarea dimensiunilor hărții, lucru realizat ușor folosind funcția `GetLength`. Următorul pas este stabilirea marginilor hărții, urmând regula explicată mai sus, în cadrul explicației constructorului clasei `MeshData`. În cele ce urmează, se declară un nou obiect de tip `MeshData` și un index pentru numărul de vârfuri. Se parcurge harta construind vectorul de vârfuri și cel `uvs`, iar în cazul în care poziția curentă nu este pe margine, se adaugă și un triunghi. În cele din urmă, funcția returnează obiectul `MeshData` generat.

```
display.DrawMesh(MeshGenerator.GenerateTerrainMesh(noiseMap,
meshHeightMultiplier, meshHeightCurve),
TextureGenerator.TextureFromColourMap(colourmap, mapWidth, mapHeight));
```

fig. 29: apelul metodei DrawMesh în MapGenerator

Dacă aruncăm acum din nou o privire în codul scriptului `MapGenerator`, vom observa cum este apelată metoda `GenerateTerrainMesh`. Aruncând din nou o privire și în codul clasei `MapDisplay`, vom găsi și modul în care datele din `MeshData` devin un mesh concret:

```
public void DrawMesh (MeshData meshData, Texture2D texture)
{
    meshFilter.sharedMesh = meshData.CreateMesh();
    meshRenderer.sharedMaterial.mainTexture = texture;
    meshCollider.sharedMesh = meshFilter.sharedMesh;
    meshCollider.enabled = true;
}
```

fig. 30: generarea unui mesh utilizând MeshData în cadrul metodei DrawMesh din MapDisplay

Ultimul detaliu rămas de explicat în cazul generării hărții acestei aplicații este modul în care se obține textura acesteia. În scopul realizării acestui aspect am folosit scriptul `TextureGenerator`. În acest script se află definiția unei clase ce poartă același nume, în cadrul căreia există o metodă: `TextureFromColourMap`.

Scopul acestei metode este acela de a prelua o hartă de culori (coincide cu harta de culori care se generează în metoda `generateMap` din clasa `MapGenerator`) și a returna un obiect de tip `Texture2D` care să corespundă acestei hărți.

```

public static class TextureGenerator {

    public static Texture2D TextureFromColourMap (Color[] colourMap, int
width, int height)
    {
        Texture2D texture = new Texture2D(width, height);
        texture.filterMode = FilterMode.Point;
        texture.wrapMode = TextureWrapMode.Clamp;
        texture.SetPixels(colourMap);
        texture.Apply();
        return texture;
    }
}

```

fig. 31: definiția clasei TextureGenerator

După cum se poate observa cu ușurință în definiția clasei TextureGenerator, nici această clasă nu are nevoie de variabile interne. În cadrul ei este definită doar metoda TextureFromColourMap care returnează un obiect de tip Texture2D și primește ca parametri harta de culori și dimensiunile hărții în format 2D (înălțimea și lățimea). În cadrul acestei metode se inițializează un obiect Texture2D numit în mod inspirat "texture" care primește dimensiunile hărții. Modul de filtrare al texturii este setat pe FilterMode.Point (pixelii texturii devin pătrați la distanță mică). Un alt parametru al obiectelor de tip Texture2D este wrapMode, care în cazul de față este setat pe Clamp. Pentru wrapMode există două variante: repeat și clamp. Acest parametru tratează situația în care coordonatele UV nu se încadrează în intervalul [0,1]. Modul repeat, după cum sugerează și numele, multiplică textura creând un efect de repetiție, iar clamp prinde textura de ultimul pixel de la margine. Motivul pentru care este folosit modul clamp este evident: harta de culori, ce reprezintă așezarea regiunilor este în strânsă relație cu harta de zgomot; în cazul în care textura rezultată ar prezenta artefacte de repetiție, reprezentarea regiunilor nu ar mai fi una fidelă.

În cele din urmă, texturii îi este aplicat colourMap-ul primit ca parametru, toate configurările sunt aplicate și textura este returnată.

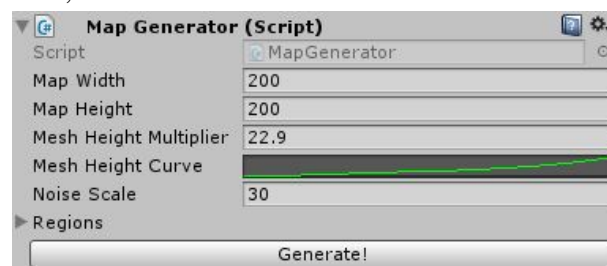


fig. 32: butonul adăugat în interfață pentru MapGenerator

Tot pe marginea subiectului hărții jocului mai există o clasă care nu a fost menționată până acum. Aceasta nu influențează harta propriu-zisă, ci interfața Unity în ceea ce privește generarea

```
public override void OnInspectorGUI()
{
    MapGenerator mapgen = (MapGenerator)target;
    DrawDefaultInspector();

    if (GUILayout.Button("Generate!"))
    {
        mapgen.generateMap();
    }
}
```

fig. 33: acționarea butonului Generate

hărții. MapGeneratorEditor este o clasă derivată din clasa de bază Editor. În cadrul ei este definit un buton care se atașează obiectului MapGenerator care are rolul de a apela funcția generateMap.

5.2.2 Jucătorul

Având în vedere faptul că proiectul prezentat în această lucrare este un joc, modul în care utilizatorul interacționează cu aplicația nu ar trebui să surprindă pe nimeni. Userul va observa și va interacționa cu universul jocului prin ochii (First Person sau persoana întâi) unui personaj fără un nume sau o poveste.

Aruncând din nou o privire la diagrama aplicației de la subcapitolul 5.2, vom observa faptul că apare o componentă a jocului numită “Player”. Acest Player nu este altceva decât un asset al cărui nume este FPSController.

După cum spuneam în capitolul 3.3 (Assets), FPSController este un asset care reprezintă o entitate ce poate fi controlată de către utilizator printr-o perspectivă “first-person” (utilizatorul observă universul jocului prin ochii personajului). Tot ce face acest asset este să includă scripturi pentru detectarea și tratarea input-ului utilizatorului, sunete pentru pași, un collider (un obiect cu proprietăți fizice, folosit în cazul de față pentru a detecta coliziuni cu suprafața podelei) și o cameră atașată acestui collider (camera reprezintă perspectiva din care poate privi un utilizator universul aplicației).

Singura modificare pe care i-am adus-o acestui asset standard (pe lângă ajustarea unor parametri precum viteza de deplasare) a fost adăugarea unei surse de lumină atașată de acest asset. Astfel, în mediul întunecos în care se află jucătorul, senzația de stres va fi amplificată datorită acestei lumini care acoperă o suprafață mică în comparație cu dimensiunile hărții,

limitând câmpul vizual al acestuia. Evident, la fel ca toți parametrii ce caracterizează FPSController, și proprietățile acestei lumini pot fi modificate după bunul plac.

5.2.3 Entitatea negativă

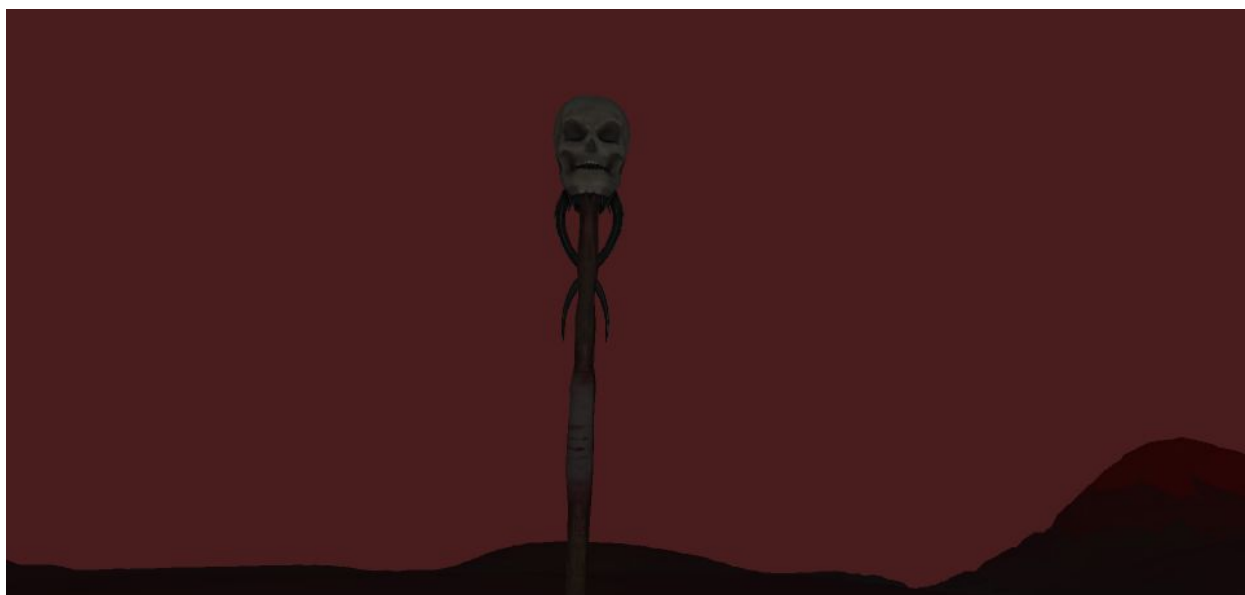


fig. 34: entitatea negativă, văzută prin ochii personajului în universul jocului

Entitatea negativă reprezintă factorul principal de stres în cadrul acestui joc care intră în categoria “horror”. În vederea obținerii unui efect de neliniște cât mai intens, ea este rezultatul combinării a trei elemente:

- un model 3D menit să sperie jucătorul prin aspectul vizual - am folosit un asset publicat de Sergi Nicolas, “Staff of Pain”;
- un script de inteligență artificială - “creierul” entității, acesta va controla modul în care entitatea se deplasează, și va încheia jocul în anumite condiții;
- sunet - două surse audio, în privința cărora voi intra în detalii imediat.

Ținând cont de faptul că scopul acestei entități este să aplice cât mai multă presiune psihică pe jucător, vom pleca de la premiza că sunetul are cel mai important rol, în timp ce restul elementelor vor crea condițiile optime pentru efectul provocat de acesta.

Sunetul este protagonistul acestei terori din mai multe motive, dintre care cel mai important este faptul că în evoluția omului, precum și în cazul multor alte mamifere, auzul a avut un rol major în mecanismul defensiv al individului [6]. Deși era dormitul în peșteri și a feritului de fiare sălbatice a fost de mult timp depășită, instinctele în privința sunetului au rămas ascuțite, permițând astfel echipelor de sunet din cadrul proiectelor horror și artiștilor să

exploateze această slăbiciune ereditară. În mod evident, pentru obținerea acestui efect este necesară folosirea unor sunete neobișnuite, la care utilizatorul să fie expus cât mai puțin în prealabil. Astfel, la întâlnirea unui astfel de sunet, jucătorul nu va fi pregătit să reacționeze.

În aplicația de față, “arhitectura” sunetului nu este una deosebit de complexă, dar este eficientă. În joc există trei surse de sunet: una inclusă în jucător (pașii acestuia), și două atașate entității negative. Cât timp entitatea nu este în vecinătatea jucătorului, acesta își va auzi doar pașii (fapt ce contribuie de asemenea la starea generală de neliniște). În momentul în care aceasta începe să se apropie, însă, jucătorul va auzi treptat cele două surse de sunet: de la distanță va auzi doar un sunet “ambiental”, asemănător într-o oarecare măsură cu un efect sonor produs de vânt (acest sunet a fost obținut utilizând un sintetizator audio numit Sytrus, folosit în contextul FL Studio); a doua sursă de sunet va fi auzită doar la o distanță mai mică, pentru a ajuta jucătorul să plaseze entitatea în spațiu (și acest fapt contribuie la senzația generală de stres). Acest al doilea sunet a fost obținut prin manipularea unei înregistrări a propriei mele voci, transformând-o într-un sunet inuman, oferindu-i și un fond “ambiental” precum prima sursă de sunet, dar mai discret. Toate aceste sunete se vor suprapune și vor crește în intensitate cu cât distanța dintre jucător și entitate este mai mică.

Nu trebuie pierdut din vedere faptul că aceste sunete sunt redade într-un spațiu tridimensional. Unity permite aplicarea efectului Doppler pe sunet, iar acest fapt are o importanță majoră în crearea unei atmosfere neobișnuite. Pe scurt, efectul Doppler se referă la schimbarea pitch-ului (înălțimea) unui sunet în situația în care sursa care îl emite este în mișcare sau observatorul este în mișcare. Utilizarea efectului Doppler în situația de față contribuie considerabil la crearea unor sunete ca din altă lume.

Ușor remarcabil este faptul că puterea sunetului ar fi anulată dacă entitatea ar fi una statică: nu ar fi nimic înfricoșător dacă entitatea stă într-un singur loc. Aici intervine scriptul entității, pe care îl putem considera a fi “creierul” acesteia. Vom arunca, așadar, o privire la BasicFollowerAI.

BasicFollowerAI este o altă clasă derivată din MonoBehaviour, în cadrul căreia se găsesc mai multe variabile și implementările a trei metode: Start, Update, și OnGUI.

```
public Transform target;
public Transform follower;
public float speed = 7;
public float hover = 0.7f;
public int chanceToTeleport = 1000;
private float actualspeed;
private bool lost;
```

fig. 35: variabilele clasei BasicFollowerAI

Se observă prezența a două variabile publice de tip Transform, și anume target și follower. După cum sugerează și numele acestor variabile, una va reprezenta obiectul urmărit (în

cazul de față playerul - FPSController) și respectiv obiectul care urmărește targetul (StaffOfPain, în cazul de față). Următoarele două variabile configurează deplasarea propriu-zisă a entității: speed va fi viteza de deplasare, iar hover va ajuta entitatea să plutească deasupra pământului (contribuind la efectul fantomatic al prezenței entității). Acești parametri pot fi modificați din interfața Unity, fiind variabile publice. Ultimele două variabile sunt private, în acest mod utilizatorul nu are acces la ele. Variabila de tip float actualspeed este necesară deoarece entitatea negativă nu va avea o viteză constantă pe toată durata partidei de joc (vom vedea imediat cum se modifică). Variabila booleană lost este un parametru intern care primește valoarea true în condițiile în care jocul este terminat în defavoarea jucătorului.

Metoda Start, moștenită din MonoBehaviour se va ocupa de inițializarea variabilei booleene lost. Aceasta este apelată o singură dată, la începutul jocului.

Vom arunca acum o privire la implementarea metodei Update, care reprezintă nucleul logic al acestei entități. Update, și ea o funcție moștenită din MonoBehaviour, este o metodă care se apelează pentru fiecare frame al aplicației. În cadrul acestei metode se află implementarea deplasării entității negative. Modul în care aceasta funcționează este următorul: în fiecare frame este calculată o nouă poziție a entității, în funcție de mai mulți factori - principalul fiind poziția targetului (jucătorul).

```

        if (Vector3.Distance(follower.position, target.position) > 100 )
        {
            actualspeed = 80f;
            transform.LookAt(target);
            transform.Translate(Vector3.forward * actualspeed *
Time.deltaTime);
            transform.Translate(Vector3.up * hover * Time.deltaTime);
        }

```

fig. 36: implementarea metodei update, situația în care distanța este mai mare decât 100 unități

Primul pas în această metodă este observarea distanței între “victimă” și urmăritor. În cazul în care aceasta este mai mare decât 100 de unități, variabila actualspeed primește valoarea 80. Astfel, dacă distanța între cele două entități este suficient de mare, viteza entității negative crește pentru a putea ține pasul cu targetul, și în acest mod se păstrează un ritm alert al jocului și un nivel de tensiune ridicat. Deplasarea propriu-zisă se face prin orientarea entității cu fața spre target folosind funcția transform.LookAt și folosirea funcției transform.Translate pentru a deplasa entitatea înainte (înainte va însemna în direcția jucătorului, deoarece obiectul este deja orientat cu fața spre el) și în sus (pentru realizarea efectului de hover).

```

else if (Vector3.Distance(follower.position, target.position) > 30)
{
    System.Random prng = new System.Random();
    actualspeed = speed;
    transform.LookAt(target);
    int decision = prng.Next(1, chanceToTeleport);
    if (decision >= chanceToTeleport - 1)
    {
        actualspeed = 0f;
        if ((int)prng.Next(1, 500) % 2 == 0)
        {
            transform.Translate(Vector3.left * 2500f *
Time.deltaTime);
            transform.Translate(Vector3.forward * 2000f *
Time.deltaTime);
        }
        else
        {
            transform.Translate(Vector3.right * 2500f * 2 *
Time.deltaTime);
            transform.Translate(Vector3.forward * 2000f *
Time.deltaTime);
        }
    }
    transform.Translate(Vector3.forward * actualspeed *
Time.deltaTime);
    transform.Translate(Vector3.up * hover * Time.deltaTime);
}

```

fig. 37: implementarea metodei Update, situația în care distanța este între 100 și 30

Dacă distanța are o valoare între 100 și 30, entitatea negativă se va deplasa cu viteza setată de utilizator în variabila publică `speed`, dar se va introduce un detaliu comportamental care nu a mai fost menționat până acum: posibilitatea de teleportare într-o poziție pseudo-aleatorie pentru a induce confuzia jucătorului.

Modul în care funcționează această teleportare este următorul: la fel ca în scriptul `PerlinNoise`, se declară un pseudo-random number generator al cărui scop este să emită o valoare aleatorie pentru variabila de tip `int` “decision” într-un interval `[1,chanceToTeleport]` (`chanceToTeleport` este un parametru public). Dacă valoarea din `decision` coincide cu extremitatea superioară a intervalului, sau cu aceeași valoare minus 1, entitatea se va “teleporta”: se mai generează un număr aleatoriu în intervalul `[1,500]`, iar dacă acesta este par, entitatea se va teleporta spre stânga (entitatea orientată inițial cu fața spre jucător se va deplasa un număr mare de unități spre stânga, și va avansa apoi și în direcția înainte - fără a se mai reorienta, același lucru se întâmplă simetric și în cazul teleportării la dreapta), iar în cazul unei valori impare se va teleporta spre dreapta. Scopul acestei teleportări este de a adăuga un element neprevăzut în joc -

jucătorul nu va ști tot timpul unde se află entitatea, și astfel poate fi surprins dintr-o direcție neașteptată de către aceasta.

```

else
{
    transform.LookAt(target);
    transform.Translate(Vector3.forward * actualspeed *
Time.deltaTime);
    transform.Translate(Vector3.up * hover * Time.deltaTime);
}

```

fig. 38: implementarea metodei Update, distanța mai mică sau egală cu 30

În cele din urmă, dacă distanța dintre cele două entități este chiar mai mică și decât 30, entitatea negativă se va deplasa pur și simplu cu viteza normală direct înspre target, în modul descris în prezentarea situației în care distanța este mai mare decât 100 de unități.

```

if (Vector3.Distance(follower.position, target.position) < 5)
{
    lost = true;
}

```

fig. 39: implementarea metodei Update, situația în care distanța este mai mică decât 5

Tot în această metodă se mai găsește o verificare: dacă distanța între cele două entități este mai mică decât valoarea de 5 unități, variabila booleana lost va primi valoarea true.

În metoda OnGUI se va verifica valoarea acestei variabile booleene:

```

void OnGUI()
{
    if (lost)
    {
        GUI.Box(new Rect(10, 10, Screen.width, Screen.height), "You
finally came to an END.");
        Time.timeScale = 0;
    }
}

```

fig. 40: definiția metodei OnGUI

În cazul în care aceasta va avea valoarea True, jocul va fi încheiat prin oprirea timpului (timescale va lua valoarea 0 - avansarea timpului va fi astfel imposibilă) și pe ecran va fi afișat mesajul “You finally came to an END.” (care ar putea fi tradus prin “În sfârșit te-am prins”).

5.2.4 Startup GameObject

Fiecare dintre subcapitolele anterioare a prezentat câte un aspect “palpabil” al jocului - acestea puteau fi văzute de către jucător, dar nu este și cazul acestuia de față. Obiectul startup este doar un container al scriptului ce poartă același nume, care se ocupă de “orchestrarea” jocului ca un întreg: se va ocupa de generarea hărții și utilizarea unui timer care să permită jucătorului să câștige jocul.

```
public class StartupScript : MonoBehaviour {
    public float timer;
    void Awake()
    {
        MapGenerator mapgen = FindObjectOfType<MapGenerator>();
        mapgen.generateMap();
    }
    void Update()
    {
        timer -= Time.deltaTime;
        if (timer <= 0)
        {
            timer = 0;
        }
    }

    void OnGUI()
    {
        if (timer == 0)
        {
            GUI.Box(new Rect(10, 10, Screen.width, Screen.height),
"Congratulations, you have survived the end.");
            Time.timeScale = 0;
        }
        else
        {
            GUI.Box(new Rect(10, 10, 60, 20), "" + ((int)timer).ToString());
        }
    }
}
```

fig. 41: definiția clasei Startup

După cum se poate observa din definiția clasei Startup, aceasta este și ea derivată din clasa de bază MonoBehaviour. Singura variabilă prezentă în această clasă este timer-ul, care este una publică de tip float. Apar trei metode, toate fiind moștenite din MonoBehaviour: Awake, Update și OnGUI, deci structura clasei este similară cu cea a entității negative.

Motivul implementării metodei `Awake` în defavoarea metodei `Start` este faptul că tot ce este descris în această metodă are loc înainte de începerea propriu-zisă a jocului. Astfel, se declară un obiect de tip `MapGenerator` (descriș în subcapitolul 5.2.1) care primește o referință la obiectul `MapGenerator` deja existent prin utilizarea funcției `FindObjectOfType<>`. Având în vedere că această metodă se execută înaintea începerii propriu-zise a jocului, nu este surprinzător faptul că următorul pas este apelul metodei `generateMap`.

În implementarea metodei `Update` (care, după cum am mai spus, se apelează pentru fiecare frame al jocului) este gestionat timerul: din el este scăzută durata de timp care s-a scurs de la ultimul frame, iar în cazul în care rezultatul este mai mic sau egal cu zero, i se stabilește valoarea nulă.

Metoda `OnGUI` este locul în care se verifică valoarea timerului, și în situația în care acesta este 0, deci jocul s-a terminat în favoarea jucătorului, timpul este oprit precum în `BasicFollowerAI`, iar pe ecran este afișat mesajul de felicitare “Congratulations, you have survived the end.” (care se poate traduce prin “Felicitări, ai supraviețuit”).

6. Concluzii

“End” este un proiect al cărui scop este explorarea genului horror în piața actuală a jocurilor video. În vederea realizării acestuia am studiat mecanicile implementate în jocurile de acest gen din ultima decadă și conceptul de horror (prezent atât în jocuri, cât și în literatură și cinematografie).

Înțelegând mecanismele de bază utilizate în acest gen, următorul pas a fost unul natural: formularea unei idei de bază pentru jocul “End”.

Știind deja că produsul finit va reprezenta un joc 3D în care jucătorul controlează un personaj anonim într-un spațiu întunecat și mereu generat aleatoriu, amenințat de o entitate cu intenții malițioase, am trecut la următoarea etapă, și anume găsirea platformei potrivite pentru implementarea acestei idei.

În punctul în care am hotărât că Unity3D este cel mai potrivit engine pentru ideea de la care am plecat, tot ce a mai rămas de făcut a fost studierea modalităților în care puteau fi implementate conceptele stabilite, și execuția acestora.

7. Bibliografie

- [1] Joe Mayo, "C# Succinctly", 2015
- [2] Alan Thorn, "Mastering Unity Scripting", 2015 (ISBN: 978-1-78439-065-5)
- [3] Documentația oficială Unity3D, <http://docs.unity3d.com/>
- [4] Adrian Biagioli (Flafla2), "Understanding Perlin Noise", 2014, <http://flafla2.github.io/2014/08/09/perlinnoise.html>
- [5] Matt Zucker, "The Perlin Noise Math FAQ", 2001, <http://webstaff.itn.liu.se/~stegu/TNM022-2005/perlinnoiselinks/perlin-noise-math-faq.html>
- [6] V. Renee Taylor, "The Sound of Horror: Why Hearing Stuff is Scarier Than Actually Seeing Stuff", 2013, <http://nofilmschool.com/2013/11/why-hearing-stuff-is-scarier-than-actually-seeing-stuff>
- [7] Daniel A. Russell, "The Doppler Effect and Sonic Booms", 1997, <http://www.acs.psu.edu/drussell/Demos/doppler/doppler.html>
- [8] Barry Keith Grant, "Screams on Screens: Paradigms of Horror", 2010, <http://journals.sfu.ca/loading/index.php/loading/article/viewFile/85/82>