1.Introducere

1.1 Context

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

În fiecare an aceasta atinge câte un nou apogeu, mulţumită nenumăratelor corporaţii, studio-uri de dezvoltare şi dezvoltatori independenţi care publică necontenit mii de jocuri pe un număr în continuă creştere 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) şi pot fi împărţite în multiple categorii, precum acţiune, aventură, simulatoare şi horror.

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

1.2 Motivaţie

Toate jocurile care intră în categoria horror împărtăşesc o trăsătură comună. Ele ţintesc să sperie consumatorul. Motivul pentru care există această categorie în mare parte a formelor de divertisment este faptul că desfăşurarea, în majoritatea timpului monotonă, a vieţii individului armează în subconştientul acestuia o nevoie uşor masochistă de “fenomenal”. Formele de divertisment de tip “horror” constituie una dintre cele mai uşoare 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 dispoziţie doar o lanternă. Scopul acestuia este să strângă 8 pagini aflate în diferite locaţii în spaţiul jocului, în timp ce evită contactul cu

Slender, personajul negativ. Slender urmăreşte jucătorul tot timpul, iniţial de la distanţă, fără să fie văzut (contactul vizual prelungit încheie jocul), devenind tot mai agresiv pe măsură ce jucătorul găseşte 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 mişca personajul după bunul plac). Scopul jocului este acela de a rezista cinci nopţi, în condiţiile în care în incinta restaurantului se află mai multe figurine animatronice cu intenţii maliţioase, personajul poate să observe spaţiul doar pornind nişte camere de supraveghere şi singurul mecanism defensiv este o pereche de uşi care stau închise doar consumând foarte mult, iar cantitatea de energie electrică pe care o are la dispoziţie este foarte mică.
- **Dead Space** (2008): jucătorul preia controlul unui personaj controlabil într-o perspecitvă third person (jucătorul vede "peste umărul" personajului) care se găseşte singur pe o navă spaţială populată de extratereştri care îi sar deseori în cale prin găuri în pereţi şi tavan. Povestea amplă a jocului plimbă personajul printr-o multitudine de spaţii şi situaţii.
- **The Suffering** (2004): jucătorul preia controlul în mod third person unui personaj proaspăt ajuns într-o puşcărie. Închisoarea este în scurt timp infestată de creaturi supranaturale precum umanoizi cu lame ca de sabie în loc de membre. Personajul îşi petrece majoritatea timpului trecând prin spaţii claustrante şi 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 şi 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 monştrii 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, bandiţi care atacă aceşti căutători, şi creaturi mutante. În scopul său de a-şi descoperi trecutul, personajul călătoreşte singur de-a lungul regiunii, străbătând ocazional laboratoare ştiinţifice subterane sovietice abandonate, în care este întâmpinat de anomalii şi creaturi monstruoase.

1.3 Concluzie

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

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

Un alt element care se regăseşte în toate aceste exemple este prezenţa unei entităţi sau a unui grup de entităţi care are ca scop hărţuirea personajului controlat de jucător. În general, acest conflict este unul care defavorizează personajul, oferind entităţilor negative abilităţi supranaturale sau limitând abilităţile ofensive ale protagonistului.

O trăsătură care probabil că nu reiese din descrierea exemplelor este sonorizarea acestor jocuri. Marea majoritate a jocurilor din categoria horror îşi bazează atmosfera pe sunetele percepute de jucător. Pe marginea acestui subiect, se poate face observaţia că există tehnici considerate clişeice (şi implicit lipsite de inspiraţie) care apar şi în filmele categorisite în acelaşi gen, precum sunete puternice declanşate brusc în urma unei linişti, corelate cu un eveniment neprevăzut în joc sau fillm (de exemplu, apariţia subită a unui inamic însoţită de un zgomot puternic, în condiţiile în care până atunci spaţiul era pustiu şi liniştea predomina atmosfera).

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

1.4 Contribuţii

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 spaţiu necunoscut mereu, şi scopul lui va fi acela de a păstra distanţa faţă de o entitate negativă care îl urmăreşte cu intenţii maliţioase pentru un timp setat în prealabil. Motivul pentru care spaţiul 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# şi realizarea unor sunete folosind un microfon şi 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 influenţat procesul de dezvoltare al aplicaţiei prezentate în această lucrare, precum conceptul de horror, efectul Doppler şi noţiunea de Perlin Noise.

2.1 Conceptul de “Horror”

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

Atât filmele, cât şi jocurile se pot împărţi în mai multe categorii, dintre care cea horror este printre cele mai uşor de observat. Un aspect interesant în privinţa acestui gen este faptul ca este definit de modul în care intenţionează să afecteze consumatorul. În timp ce unele genuri precum science fiction si western sunt definite de către acţiunile descrise şi timpul şi spaţiul în care acestea au loc, altele, precum comedie şi horror sunt definite pe baza unor anumite emoţii provocate la nivelul consumatorului.

Reacţiile 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 reuşeşte să sperie cât mai tare privitorul, la fel cum un film horror care nu reuşeşte să atingă acest efect este considerat ca fiind unul prost, iar accest exemplu se poate aplica la fel şi la genul “dramă” (un film bun emoţionează privitorul, unul prost îl lasă indiferent) precum şi “comedie” (produsul care amuză consumatorul este unul bun, cel care eşuează în acest aspect este unul prost). [8]

Totuşi, pentru a conceptualiza această noţiune, un prim pas ar fi definirea emoţiei 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 scenarii apocaliptice (“28 Days Later” (film,2002), “Herbert West Reanimator” (H.P. Lovecraft, 1921), “Dead Island” (joc, 2011) - începând din anii ‘80 şi până acum, subiectele care prezintă apocalipse cu zombie au fost o temă horror utilizată aproape până în punctul epuizării - toate generaţiile care coexistă în momentul de faţă cunosc noţiunea de zombie, deci aceasta poate fi considerată a fi clişeică, având în vedere faptul că această temă a apărut în mii de filme, cărţi şi jocuri) şi 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, cărţi, şi jocuri care trateaza teme legate de zone bântuite, fenomene paranormale şi întâlniri cu forme de viaţă extraterestre).

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

horror", care reprezintă un curent observat în mai multe filme al căror element distinctiv este conţinutul grafic explicit ce prezintă mutilarea şi desfigurarea personajelor. Modul în care interacţionează 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 aşa ceva, precum şi cărţi - de exemplu, romanele în care Sven Hassel prezintă ororile războiului) forţează imaginaţia cititorului, privitorului sau a jucătorului să reproducă senzaţiile 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 discreţie, din punctul meu personal de vedere, merită mai multă apreciere decât materialele atât de grafice încât ajung pe muchia gratuităţii - evident, aceasta este o temă subiectivă). Acestea au o abordare total diferită a prezentării subiectului ales de autor; în general au un ritm mai puţin agitat, iar teroarea, în loc de a fi o reacţie în mare parte fizică, este în mare parte rezultatul imaginaţiei consumatorului. Un exemplu bun în acestă direcţie este filmul "The Blair Witch Project" (1999), în care conţinutul grafic extrem este foarte puţin întâlnit: acest film prezintă povestea a trei tineri ce decid să investigheze o pădure despre care circulă zvonul că ar fi un spaţiu generos în privinţa fenomenelor paranormale. Întreaga acţiune este văzută de către audienţă prin camerele de filmat pe care le-au luat tinerii după ei pentru a documenta expediţia. De-a lungul întregului film, nu apare nici măcar o dată pe peliculă vreo creatură monstruoasă, întreaga atmosferă fiind creată de spaţiul în care se află protagoniştii şi de reacţiile lor la diferitele evenimente care au loc (de exemplu, găsirea unor figurine din crenguţe 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 situaţie deja tensionată. Un alt motiv pentru care acest film este un bun subiect de studiu de caz este modul în care conceptul de "horror" depăşeşte filmul în sine; lansarea acestuia a fost însoţită iniţial şi de mitul că tinerii din film nu erau actori, ci materialul este veritabil şi nu se mai ştie nimic despre protagonişti.

Astfel, se ajunge la o altă noţiune care defineşte o lucrare horror de calitate, şi anume modul în care materialul la care a fost expus consumatorul continuă să îl macine pe acesta chiar şi după ce a fost sfârşit. Acest efect este cel mai uşor 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 înţelegerii faptului că materialul la care a fost expus este o lucrare de ficţiune - acesta poate suferi traume reale; un alt exemplu de imaturitate (combinată cu traume precum poveştile din bătrâni auzite de la bunici sadici în frageda copilărie) în această privinţă este existenţa superstiţiilor, şi 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 hărţii din aplicaţia prezentată se bazează în mare parte pe acest concept.

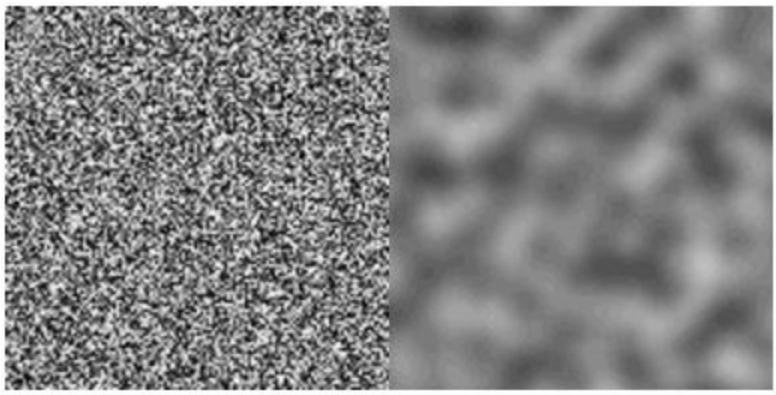

fig. 1: stânga - zgomot incoerent, dreapta - Perlin Noise [5]

Perlin Noise este o metodă de generare a zgomotului gândită şi 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, mulţumită gradului de realism pe care aceasta o produce. Folosind Perlin Noise, se generează suprafeţe de obiecte, foc, fum şi nori cu un aspect realistic, imitând apariţia aparent aleatorie a acestora în natură.

Definiţia formală a acestuia, după cum a formulat-o Matt Zucker în lucrarea sa “Perlin Noise Math FAQ” (2001) este următoarea: Perlin Noise este o funcţie folosită pentru generarea unui zgomot coerent peste un spaţiu, în condiţiile în care coerenţa zgomotului reprezintă faptul că pentru oricare două puncte din acel spaţiu valoarea funcţiei se schimbă cu fineţe trecând de la un punct la celălalt, fără să existe discontinuităţi. [5]

Ce reprezintă, de fapt, o funcţie de zgomot este o funcţie care pentru nişte coordonate într-un spaţiu furnizează un număr real cu valoarea cuprinsă între -1 şi 1. Aceste funcţii pot fi aplicate unui spaţiu, indiferent de numărul acestuia de dimensiuni.[5] Astfel, pe lângă aplicarea 2D pe care am utilizat-o în acest proiect, pot fi implementate versiuni şi 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 spaţiu).

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

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

```
noise2d(x, y) = z
```

fig. 2: funcţia în cazul 2D

Unde x, y şi z sunt numere reale.

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

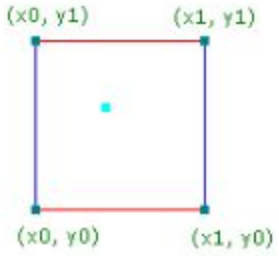

fig. 3: punctul (x,y) cuprins între dreptele corespondente numerelor întregi între care se află cuprins [5]

Următorul pas este definirea unei funcţii

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

fig. 4: definirea funcţiei 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 acelaşi punct va return mereu acelaşi vector, iar fiecare vector are o şansă egală de a fi ales.

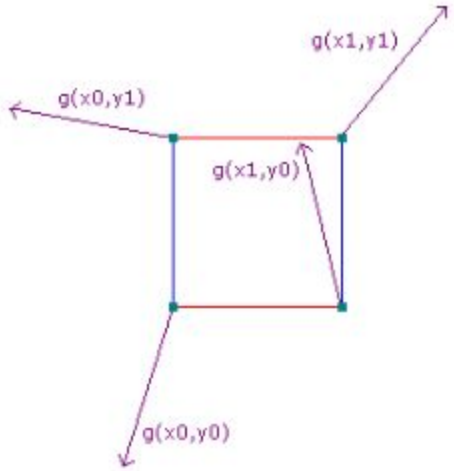

fig. 5: vectorii returnaţi de funcţia g pentru punctele care înconjoară (x,y) [5]

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

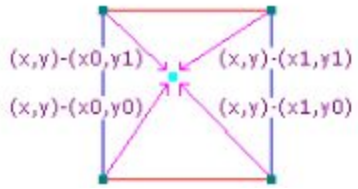

fig.6: vectorii orientați spre punctul pentru care se calculează valoarea zgomotului [5]

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

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: notaţia celor patru influenţe [5]

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

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

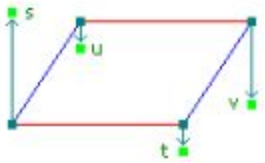

fig. 8: reprezentarea grafică a vectorilor de influenţă s, t, u, v "ieşind" din planul $\boldsymbol{R}^2$ *[5]*

Valoarea zgomotului reprezintă o medie a acestor vectori de influenţă.

3.Platforma de Dezvoltare

În momentul de faţă, piaţa platformelor de dezvoltare a jocurilor se bucură de un apogeu, oferind o paletă imensă de opţiuni. Aceste opţiuni sunt atât de accesibile acum, încât oricine are o idee şi suficientă motivaţie poate începe dezvoltarea unui joc ðin 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 cunoştinţe de programare. De asemenea, merită menţionat faptul că fiecare platformă, pe lângă documentaţia de bază, este susţinută şi de câte o comunitate imensă formată din entuziaşti. Astfel, în cazul în care un dezvoltator, indiferent de nivelul de experienţă, se găseşte pus în situaţia de a se confrunta cu o problemă pe care nu o poate rezolva de unul singur, are şanse foarte mari de a-şi găsi întrebarea gata formulată pe forumurile comunităţilor. Şi în cazul în care nu a mai fost nimeni care să întrebe înaintea lui, cu siguranţă va găsi pe cineva care s-a confruntat cu o situaţie similară şi e gata să îi răspundă.

3.1 Opţiunile disponibile

În funcţie de scopul produsului finit, există posibilitatea alegerii dintre mai multe platforme, printre care şi cele enumerate mai sus. În cazul de faţă, ştiam de la bun început că doresc să dezvolt un joc a cărui acţiune se va desfăşura într-un spaţiu tridimensional, şi, implicit, aria de platforme dintre care puteam alege s-a restrâns. Ţinând 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, şi astfel am ajuns pus în faţa unei alegeri între două platforme: Unity şi Unreal Engine 4. Pentru a lua o decizie cât mai potrivită, am luat în calcul mai mulţi factori.

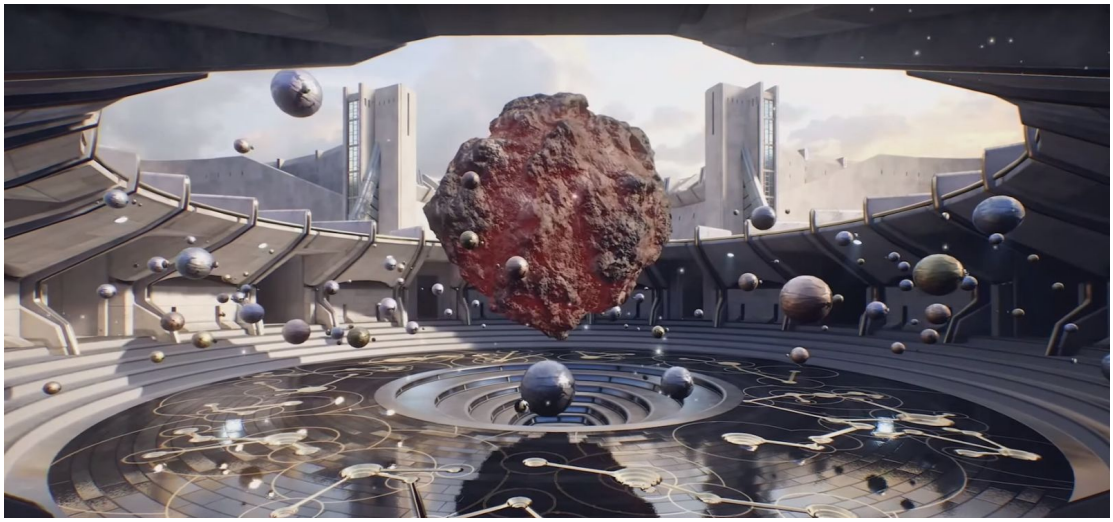

fig. 9: un frame dintr-o aplicaţie creată în Unreal Engine

În primul rând, trebuie precizat faptul că am mai avut o experienţă foarte sumară cu Unity acum câţiva ani ca rezultat al curiozităţii personale, precum şi cu alte platforme precum Game Maker (în 2006) şi RPG Maker. Aşadar, aveam o foarte vagă idee despre cum ar putea decurge dezvoltarea unui joc. Deşi aveam o uşoară preferinţă pentru Unity, dată fiind experienţa relativ recentă şi posibilitatea de dezvoltare cross-platform, am fost foarte tentat şi de alegerea Unreal Engine, ştiind că acesta din urmă a fost baza succesului comercial al multor titluri. În alte cuvinte Unity era o opţiune atrăgătoare datorită facilităţii, iar Unreal Engine era o opţiune atrăgătoare datorită calităţii aproape inegalabile produsului final. De asemenea, merită menţionat faptul că Unreal Engine a ajuns la îndemâna publicului mult mai recent decât Unity (care avea deja un avantaj de câţiva ani în creşterea comunităţii). Acestea fiind spuse, am început să mă documentez în legătură cu procesul de dezvoltare şi cu opţiunile disponibile în scopul realizării ideii mele atât în Unity, cât şi în Unreal Engine şi am ales într-un final Unity, ţinând cont şi de presiunea timpului pe care îl aveam la dispoziţie.

3.2 Unity

Unity este un game-engine scris în C şi C++ lansat iniţial 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 mulţumită mai multor factori: oferă soluţii pentru dezvoltarea aplicaţiilor atât 2D cât şi 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 aplicaţiile Unity pot fi rulate. Astfel, dezvoltatorii nu sunt forţaţi să opteze pentru o anumită platformă în vederea dezvoltării, şi au posibilitatea de a utiliza atât Windows şi MacOS, cât şi Linux (Ubuntu - încă în beta).

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

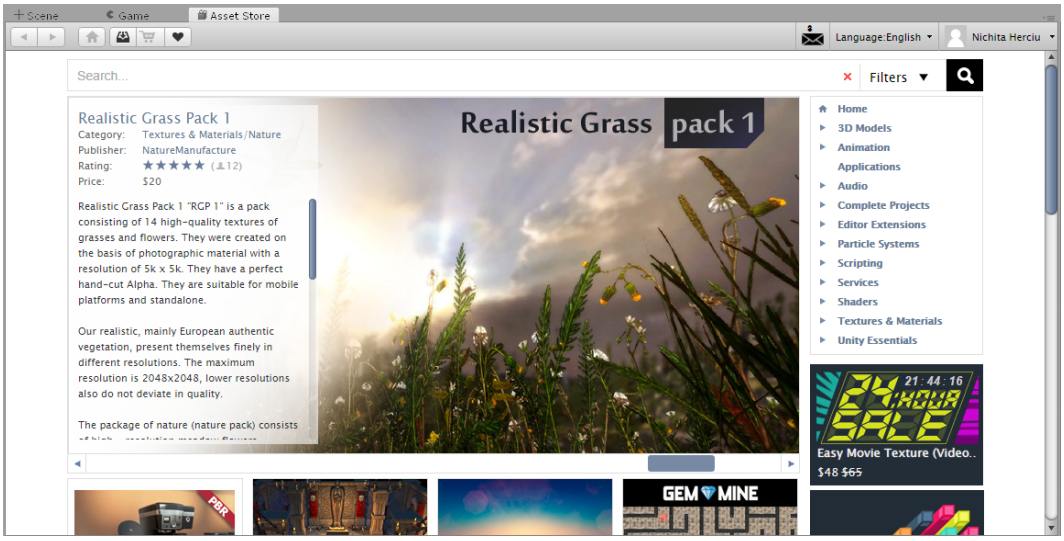

fig. 10: Unity Asset Store

3.3 Assets

Un alt aspect esenţial care contribuie la experienţa dezvoltării în Unity este prezenţa 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 uşurinţă într-o aplicaţie Unity. Pentru o mai uşoară exemplificare a noţiunii de asset, poate fi observată importanţa asset-urilor în proiectul de faţă. În această aplicaţie sunt folosite două asset-uri: FPSController şi 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 şi tratarea input-ului utilizatorului, sunete pentru paşi, un collider (un obiect cu proprietăţi fizice, folosit în cazul de faţă pentru a detecta coliziuni cu suprafaţa podelei) şi o cameră ataşată acestui collider (camera reprezintă perspectiva din care poate privi un utilizator universul aplicaţiei). Majoritatea asset-urilor, precum acesta, conţin şi unul sau mai multe “prefabs” care reprezintă un obiect al jocului prefabricat ce include toate caracteristicile menţionate mai sus şi le configurează pentru a lucra împreună ca un tot-unitar.

fig. 11: Scriptul pentru AI-ul entităţii 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 comunităţii, pe 5 noiembrie 2015. Acesta conţine un model 3D pentru o baghetă şi texturile aferente. Am decis să folosesc acest asset deoarece doream un model care să fie în ton cu atmosfera jocului pentru entitatea negativă, şi timpul nu mi-ar fi permis să învăţ să modelez cu o asemenea precizie profesională. Aşadar, am importat acest asset în aplicaţia mea şi i-am adăugat scriptul făcut de mine pentru AI-ul inamicului, sunete şi un collider pentru a avea nişte proprietăţi fizice.

3.4 Sunet

Pentru crearea sunetelor folosite în aplicaţie 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 nişte sunete ambientale, şi am modificat nişte sunete înregistrate de mine pentru a-i oferi entităţii negative din universul jocului o “voce”.

4. Tehnologii folosite

În acest capitol voi prezenta sumar tehnologiile utilizate în dezvoltarea aplicaţiei prezentate, şi anume “End”, jocul realizat în Unity bazat pe scripturi în C#. La sfârşitul 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, iniţial în cadrul proiectului .NET, care mai apoi a fost acceptat ca standard de către Ecma (ECMA-334) şi 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 aplicaţii web prin intermediul ASP .NET, aplicaţii desktop prin WPF (Windows Presentation Foundation) sau aplicaţii mobile pe Windows Phone. C# are un impact chiar şi în piaţa de cloud computing, fiind în strânse relaţii 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 şi bibleotecile framework-ului, ce permite dezvoltarea unui număr mare de tipuri de aplicaţii. C# este unul dintre limbajele de programare incluse în .NET, pe lângă Visual Basic, F#, C++ şi altele.

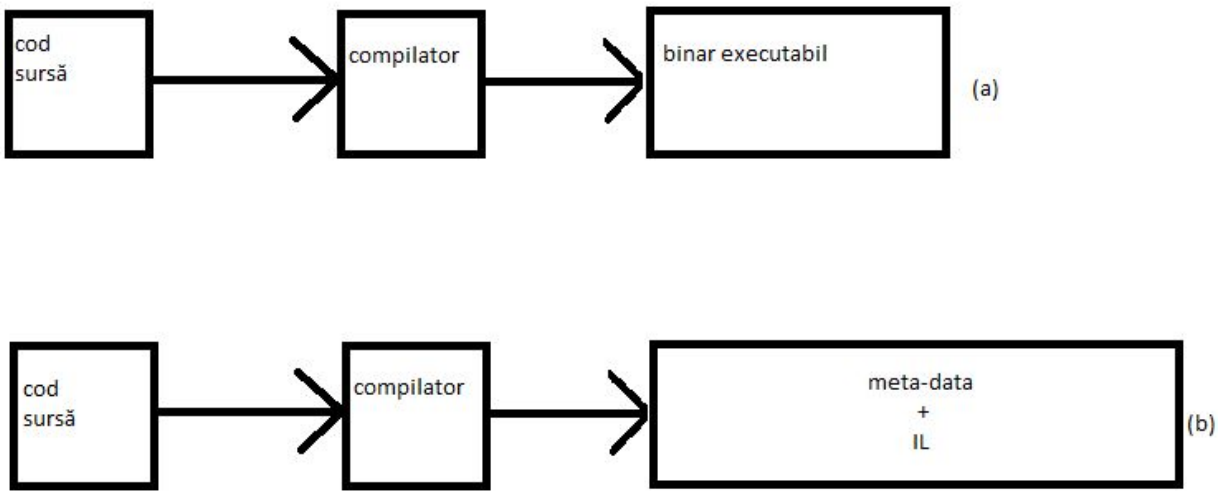

fig. 12:
(a) compilare tradiţională
(b) CLR

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 şi oferă mai multe servicii, precum gestionarea memoriei, gestionarea excepţiilor, securitate, garbage collection, etc.[1]

4.2 .NET şi C# în contextul Unity

Majoritatea platformelor de dezvoltare a jocurilor pun la dispoziţia utilizatorului atât un GUI, cât şi posibilitatea de a scripta în diverse limbaje de programare diferite elemente sau aspecte, atât ale aplicaţiei, cât şi a platformei. Unity nu este o excepţie 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 privinţa limbajului de programare ce va fi folosit. Unity propune două soluţii “oficiale”, şi anume C#, şi o variantă ajustată pentru Unity a limbajului Javascript. Unity nu impune nicio restricţie în privinţa acestei decizii, permiţând utilizarea concomitentă atât a limbajului C#, cât şi Javascript, deşi acest amestec este privit ca “bad practice” deoarece poate duce la confuzie şi conflicte la compilare.

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

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

5. Prezentarea aplicaţiei

În acest capitol voi prezenta aplicația propriu-zisă, voi explica procesul de dezvoltare al acesteia, problemele întâmpinate, şi soluțiile găsite la aceste probleme. În subcapitolele ce urmează, voi folosi exemple bazate pe bucăţi de cod, care se vor remarca faţă 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 Specificaţii funcţionale

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, roşiatic, cu un cer sângeriu şi o topografie ieşită din comun (munți, cratere, etc.). De asemenea, la inițierea jocului va fi afişat pe ecran şi un timer care va descreşte, ce simbolizează numărul de secunde pe care jucătorul trebuie să le mai supraviețuiască pentru a câştiga 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
- liniştea inițială (până să se afle în proximitatea entității negative, personajul îşi va auzi doar paşii)
- mediul sterp, cu aspect ostil (cer, munți şi cratere în diverse tonalități de roşu, de la roşu aprins, la negru)

- prezenţa entităţii negative (sunete inumane, aspect înfricoşător, perseverenţă în hărţuirea personajului)

Entitatea negativă va fi rezultatul aplicării unui script ce simulează o inteligenţă artificială pe un model 3D cu un aspect deranjant. Mulţumită 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 funcţie de distanţa faţă de personaj (Scriptul dispune de nişte parametri care definesc distanţele la care să se modifice viteza de călătorie. Scopul acestei schimbări de viteză este de a ţine mereu entitatea la o distanţă potrivită faţă de personaj - viteza sa normală va fi mai mică decât cea a personajului în fugă, dar de la un anumit prag de distanţă viteza i se poate, dubla, tripla, sau chiar înzeci pentru a ajunge victima din urmă)
- va avea abilitatea de a se teleporta într-o poziţie pseudo-aleatorie faţă de cea curentă, daca condiţiile sunt îndeplinite (dacă entitatea se află într-un range faţă de personaj care să-i permită sa meargă cu viteza minimă, aceasta va avea o probabilitate mică de a avansa instantaneu o distanţă mare înspre personaj, puţin orientat spre stânga sau spre dreapta personajului. Aceasta se întâmplă pentru a oferi un caracter neprevăzut entităţii negative, iar orientarea spre stânga sau spre dreapta -aleasă şi ea tot aleatoriu- are ca scop prelungirea jocului, sau creşterea şansei de supravieţuire a jucătorului - astfel se evită o coliziune produsă în momentul teleportării între entitate şi jucător, coliziune care ar încheia jocul)

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

5.2 Arhitectura aplicaţiei

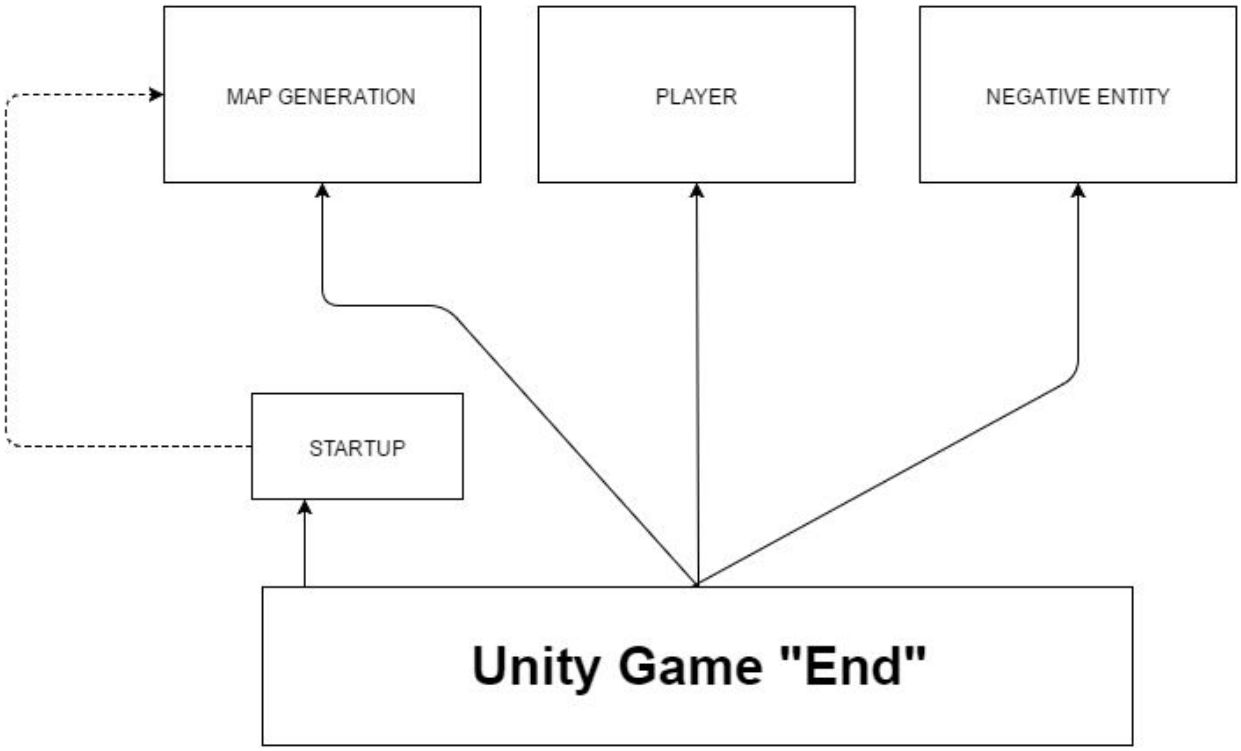

fig. 14: O diagramă simplificată a aplicației

Pentru o reprezentare fidelă a arhitecturii unui proiect dezvoltat în Unity, trebuie clarificat faptul că aceste aplicaţii nu sunt orientate-obiect, ci bazate pe componente. În scopul înţelegerii noţiunii 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 aplicaţii este un GameObject. Totuşi, un GameObject nu poate face nimic de unul singur, ci are nevoie nişte proprietăţi pentru a putea deveni ceva. Aceste proprietăţi sunt în componente. Astfel, un GameObject poate fi privit ca un container pentru mai multe componente, în care pot fi păstrate sunete, lumini, animaţii, script-uri, etc.

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

- Un GameObject corespondent hărţii sau a spaţiului în care se petrece acţiunea;
- Un GameObject ce reprezintă jucătorul;
- Un GameObject ce reprezintă entitatea negativă.

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

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

5.2.1 Harta

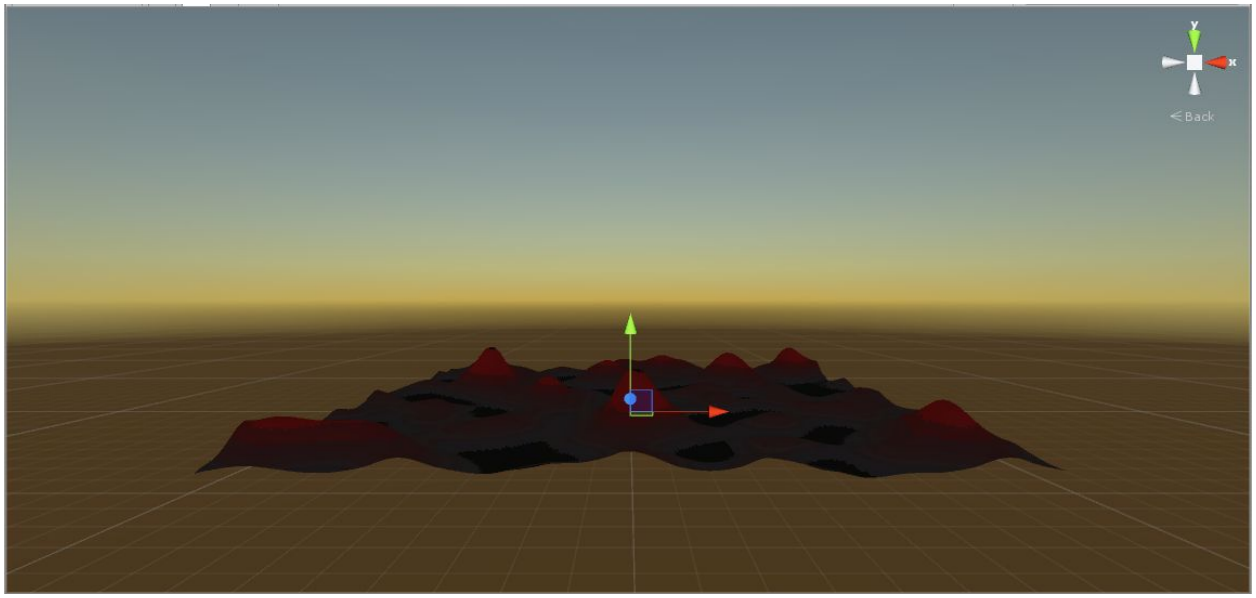

fig. 15: Un exemplu de hartă generată

În privinţa hărţii jocului, în explicaţia anterioară a fost omis un detaliu semnificativ în scopul înţelegerii arhitecturii de bază a aplicaţiei: pentru obţinerea hărţii nu este folosit un singur GameObject, ci două; unul se ocupă de generarea hărţii (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 hărţii este un mesh. Un mesh este un grup de triunghiuri aranjate într-un spaţiu tridimensional astfel încât crează impresia unui obiect solid. Un triunghi este definit de trei puncte sau vârfuri. Noţiunii de mesh îi corespunde şi o clasă numită sugestiv “Mesh”. În această clasă, triunghiurile sunt stocate printr-un vector ce conţine toate vârfurile; fiecare triunghi este definit de câte trei elemente ale vectorului. Astfel elementele 0, 1 şi 2 ale vectorului ar reprezenta primul triunghi, 3, 4 şi 5 ar reprezenta al doilea triunghi, etc.[3]

MapGenerator este un GameObject ce conţine două componente esenţiale: un script de generare a hărţii şi un script de afişare a acesteia.

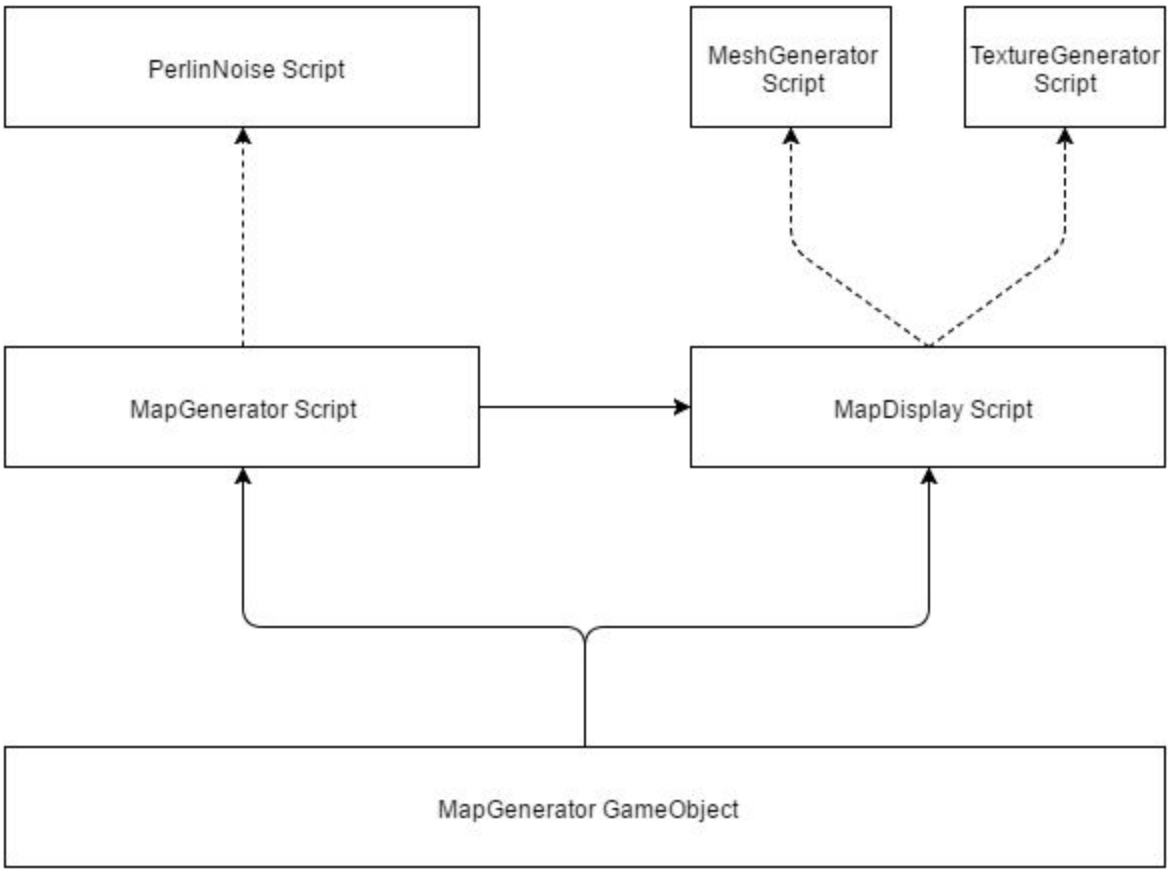

fig. 16: O diagramă ce reprezintă structura obiectului MapGenerator

Pentru a înţelege 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 biţi (care, în mod intuitiv, pot lua doar o valoare dintre {0,1}).

Având în vedere că scopul principal este generarea aleatoare a unei hărţi de joc, prima variantă la care s-ar putea gândi oricine ar fi asignarea aleatorie a valorilor fiecărui bit. Algoritmul ar fi unul trivial, şi 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, deşi ar fi foarte uşor 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 şi valorile vecinilor săi. Cuvântul care descrie această

situaţie este “incoerenţă”. Dată fiind această problemă a coerenţei, se poate observa că o hartă generată absolut aleatoriu nu ar putea fi interpretată ca grupuri de biţi de aceeaşi valoare care să constituie elemente ce ar putea reprezenta mai târziu munţi, dealuri, văi, etc.

Aşadar, în scopul atingerii obiectivului iniţial, vom renunţa 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 şi vom face o mică ajustare: în locul unui vector bidimensional de biţi, vom considera un vector bidimensional de numere (raţionale, pentru acest exemplu). Astfel, fiecare valoare va reprezenta “înălţimea” elementului în situaţia transpunerii tabloului într-un spaţiu tridimensional. De exemplu, pentru situaţia propusă mai devreme a generării unei hărţi ce conţine forme topografice diverse, precum munţi, dealuri şi văi, o hartă de biţi, chiar şi coerentă, nu va fi suficientă pentru modelarea spaţiului deoarece nu am putea exprima diferenţa de înălţime între un vârf de munte şi un punct dintr-o vale. În cazul folosirii unui vector de valori raţionale, deja se poate contura tipul de spaţiu descris mai sus.

Pentru a rezolva problema coerenţei, 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 coerenţă semnificând lipsa discontinuităţilor în textura generată. Acest algoritm este unul foarte popular în industria jocurilor şi a filmelor, fiind folosit în diverse scopuri, precum simularea focului sau a unor nori.

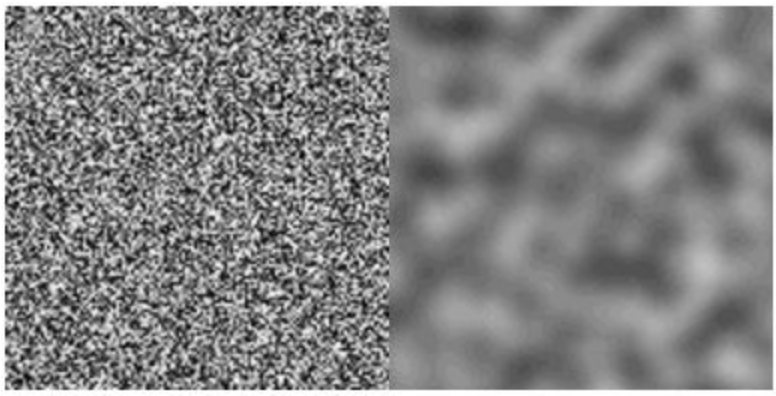

fig. 18: stânga - zgomot incoerent, dreapta - zgomot coerent [5]

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

Din fericire, în Unity există deja o implementare a acestui algoritm: este pusă la dispoziţia dezvoltatorilor funcţia PerlinNoise din colecţia Mathf. Această funcţie primeşte ca parametri două valori de tip float, ce reprezintă coordonatele x şi y ale unui punct “sample” din plan, şi returnează o valoare float între 0.0 şi 1.0. Fiind o funcţie pseudo-random, pentru acelaşi sample va returna mereu aceeaşi 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 şi mainRandomY, ambele aparţinând intervalului [-1000,+1000]. Aceste două variabile nu îşi vor schimba niciodată valoarea de-a lungul întregului proces de generare al unei hărţi. În cele ce urmează, se declară un vector bidimensional gol în care urmează să fie stocate valorile generate şi se verifică dacă variabila scale este mai mică sau egală cu 0, iar în cazul în care condiţia este îndeplinită, scale primeşte 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 şi detailRandomY, în intervalul [-1,1], de data aceasta. Se construiesc apoi, tot pentru fiecare element, cele două sample-uri necesare pentru apelarea funcţiei PerlinNoise. Aşadar, pentru fiecare element din vector se generează două sample-uri corespondente coordonatei x şi respectiv coordonatei y. În aceste sample-uri vom găsi:

- Valorile coordonatelor curente, raportate la scale;
- Valorile mainRandom - acestea dictează formele principale ale reliefului hărţii, fiind consistente între iteraţii (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 funcţia 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 foloseşte rezultatul returnat de funcţia prezentată mai devreme.

Se observă cu uşurinţă 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 interfeţei 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 hărţii, iar noiseScale determină fineţea acesteia. După cum era de aşteptat, aceste variabile sunt folosite la

apelul funcţiei PerlinNoise. Observăm doi parametri noi, şi anume meshHeightMultiplier, şi 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: definiţia tabloului de regiuni

Acesta este un vector unidimensional de tip TerrainType care stochează regiuni. Pentru a înţelege ce reprezintă aceste regiuni, vom arunca acum o privire şi la definiţia structurii TerrainType, care se găseşte tot în scriptul MapGenerator:

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

}
```

fig. 22: definiţia structurii TerrainType

Aşadar, structura TerrainType reţine un nume, o înălţime, şi o culoare. Numele unui tip de regiune este suficient de sugestiv; înălţimea este o valoare de tip float care precizează de la ce înălţime pe hartă vom găsi această regiune, iar culoarea reprezintă chiar culoarea pe care o vom găsi şi în joc în acest tip de regiune.

Astfel, în MapGenerator vom avea un vector de regiuni, lăsând posibilitatea modificării totale a aspectului cromatic al spaţiului 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, şi anume "OnValidate" şi "generateMap". OnValidate - metodă moştenită din MonoBehaviour, după cum sugerează şi numele, validează nişte parametri. În cazul de faţă se validează dimensiunile hărţii: dacă una dintre dimensiuni este mai mică decât 1, primeşte valoarea 1. Astfel, se evită situaţia în care harta de zgomot ar fi definită eronat.

Următorul aspect este şi ultimul pe care îl vom studia din acest script; este vorba despre metoda generateMap, trăsătura esenţială 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 hărţii de zgomot cu ajutorul funcţiei GenerateNoiseMap din scriptul PerlinNoise, cu parametrii pe care îi găsim definiţi în corpul clasei: mapWidth, mapheight şi 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 şi tabloul de regiuni pentru a construi vectorul de culori. În mod intuitiv, harta va fi parcursă prin cei doi indici corespondenţi înălţimii şi lăţimii punctului curent în hartă, iar regiunile vor fi şi ele parcurse în ordinea în care

au fost stocate. Deşi vectorul de culori are o singură dimensiune, se doreşte replicarea hărţii de zgomot în culori în interiorul acestuia, aşadar el va fi parcurs ca atare: se va înmulţi "înălţimea" curentă în hartă cu înălţimea maximă pentru a simula un indice al rândului în matrice, la care se adună pur şi simplu indicele coloanei (y*mapWidth + x).

Având acum atât harta de zgomot, cât şi cea de culori, tot ce rămâne de făcut este crearea unui mesh şi a unei texturi pentru acest mesh, şi afişarea lui. Pentru această sarcină am creat MapDisplay, o altă clasă derivată şi 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 afişarea pe ecran a oricărui obiect;
- meshFilter: oferă un mesh unui MeshRenderer;
- meshRenderer: un MeshRenderer, intuitiv, afişează pe ecran un mesh;
- meshCollider: este un obiect de tip MeshCollider, acesta va fi baza interacţiunii fizice între hartă şi entităţi (jucătorul şi entitatea negativă).

Pe lângă aceste variabile, în MapDisplay este definită şi 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 primeşte ca parametri datele ce definesc mesh-ul stocat într-o clasă numită MeshData, şi o textură. Vom detalia imediat şi modul în care se obţin aceşti parametri. În cadrul metodei sunt setate corespunzător mesh-ul şi colliderul acestuia, după ce mesh-ul este generat la apelul funcţiei CreateMesh.

În scriptul MeshGenerator se găsesc definiţiile a două clase, şi anume MeshGenerator şi 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 menţionat, este o clasă menită să modeleze datele necesare creării unui Mesh. Precum celelalte clase prezentate până acum, şi 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 spaţiu tridimensional;
- triangles: un tablou unidimensional ce reţine triunghiurile după vârfuri;
- uvs: UVs în Unity este o notaţie a unui sistem de coordonate carteziene într-un spaţiu 2D, cu originea în stânga jos; folosim notaţia UV deoarece XYZ sunt deja

folosite în definirea spaţiului şi astfel se evită crearea confuziilor. În MeshData există un tablou unidimensional de obiecte de tip Vector2 pentru aceşti vectori;

- triangleIndex: numără vârfurile ce compun triunghiuri - această variabilă va creşte 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 şi CreateMesh - metodă utilizată pentru crearea şi returnarea unui Mesh din triunghiurile reţinute.

Constructorul acestei clase primeşte ca parametri două valori de tip int; acestea reprezintă dimensiunile hărţii. În cadrul acestei metode se iniţializează tablourile vertices, uvs şi triangles. Nu există niciun mister în iniţializarea tablourilor vertices şi uvs - dimensiunile acestora sunt pur şi simplu produsul celor două dimensiuni ale hărţii. În cazul tabloului de triunghiuri, lucrurile stau puţin diferit, totuşi. În cod se observă că numărul de triunghiuri rezultate din cele două dimensiuni este (lăţime-1)*(înălţime-1)*6. Pentru a înţelege această formulă, vom folosi un exemplu uşor. Vom considera o hartă de dimensiune 2*2 → patru elemente. Aceste elemente vor reprezenta vârfurile triunghiurilor, aşadar le vom numerota de la 0 la 3:

0 1
2 3

În cazul acestei hărţi, triunghiurile ce compun mesh-ul vor fi următoarele: (0 → 3, 3 → 2, 2 → 0) şi (3 → 0, 0 → 1, 1 → 3). Se observă cu uşurinţă faptul că triungiurile sunt luate în sens invers trigonometric. Aşadar, în calculul numărului de triunghiuri înmulţim (lăţime -1) cu (înălţime - 1) deoarece extremităţile hărţii vor fi deja incluse în triunghiuri existente. Motivul pentru care se înmulţeşte 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 funcţie de tip void ce primeşte ca parametri trei valori de tip integer, ce reprezintă vârfurile unui triunghi. În cadrul acestei funcţii vârfurile sunt adăugate în vector în ordine, iar variabila triangleIndex este incrementată cu 3.

Metoda de creare a mesh-ului este o funcţie ce returnează un obiect de tip mesh care nu primeşte parametri, deoarece toate datele sunt stocate în variabilele clasei. În cadrul acestei metode este iniţializat un mesh nou, căruia îi sunt setate vârfurile (tabloul vertices răspunde acestei cerinţe), triunghiurile (tabloul triangles) şi 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 şi 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 hărţii de zgomot. Metoda GenerateTerrainMesh primește ca parametri harta, şi doi parametri care au mai fost întâlniţi în cadrul clasei MapGenerator, care în sfârşit primesc o utilitate: este vorba despre parametrul de tip float heightMultiplier, şi parametrul de tip AnimationCurve heightCurve. Modul în care funcţionează aceşti parametri este unul intuitiv, dar se bazează pe înţelegerea noţiunii de mesh. În cazul aplicaţiei de față, la un moment t va exista o hartă de zgomot, un set de regiuni, o hartă de culori şi un mesh format dintr-o mulţime de triunghiuri în acelaşi plan. Pentru a crea efectul dorit, şi anume un teren cu

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

Primul pas din această metodă este determinarea dimensiunilor hărţii, lucru realizat uşor folosind funcţia GetLength. Următorul pas este stabilirea marginilor hărţii, urmând regula explicată mai sus, în cadrul explicaţiei constructorului clasei MeshData. În cele ce urmează, se declară un nou obiect de tip MeshData şi un index pentru numărul de vârfuri. Se parcurge harta construind vectorul de vârfuri şi cel uvs, iar în cazul în care poziţia curentă nu este pe margine, se adaugă şi un triunghi. În cele din urmă, funcţia 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 şi în codul clasei MapDisplay, vom găsi şi 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 hărţii acestei aplicaţii este modul în care se obţine textura acesteia. În scopul realizării acestui aspect am folosit scriptul TextureGenerator. În acest script se află definiţia unei clase ce poartă acelaşi 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) şi a returna un obiect de tip Texture2D care să corespundă acestei hărţi.

```
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 uşurinţă în definiţia 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 şi primeşte ca parametri harta de culori şi dimensiunile hărţii în format 2D (înălţimea şi lăţimea). În cadrul acestei metode se iniţializează un obiect Texture2D numit în mod inspirat "texture" care primeşte dimensiunile hărţii. Modul de filtrare al texturii este setat pe FilterMode.Point (pixelii texturii devin pătraţi la distanţă mică). Un alt parametru al obiectelor de tip Texture2D este wrapMode, care în cazul de faţă este setat pe Clamp. Pentru wrapMode există două variante: repeat şi clamp. Acest parametru tratează situaţia în care coordonatele UV nu se încadrează în intervalul [0,1]. Modul repeat, după cum sugerează şi numele, multiplică textura creând un efect de repetiţie, iar clamp prinde textura de ultimul pixel de la margine. Motivul pentru care este folosit modul clamp este evident: harta de culori, ce reprezintă aşezarea regiunilor este în strânsă relaţie cu harta de zgomot; în cazul în care textura rezultată ar prezenta artefacte de repetiţie, 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 şi textura este returnată.

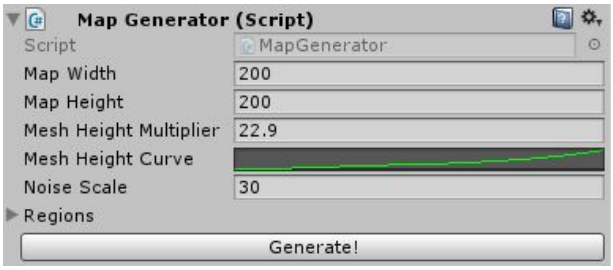

fig. 32: butonul adăugat în interfață pentru MapGenerator

Tot pe marginea subiectului hărţii jocului mai există o clasă care nu a fost menţionată până acum. Aceasta nu influenţează harta propriu-zisă, ci interfaţa Unity în ceea ce priveşte generarea

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

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

fig. 33: acţionarea butonului Generate

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

5.2.2 Jucătorul

Având în vedere faptul că proiectul prezentat în această lucrare este un joc, modul în care utilizatorul interacţionează cu aplicaţia nu ar trebui să surprindă pe nimeni. Userul va observa şi va interacţiona 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 controlatăde către utilizator printr-o perspecitvă “first-person” (utilizatorul observă universul jocului prin ochii personajului). Tot ce face acest asset este să includă scripturi pentru detectarea şi tratarea input-ului utilizatorului, sunete pentru paşi, un collider (un obiect cu proprietăţi fizice, folosit în cazul de față pentru a detecta coliziuni cu suprafaţa podelei) şi o cameră ataşată 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 ataşată de acest asset. Astfel, în mediul întunecos în care se află jucătorul, senzaţia de stres va fi amplificată datorită acestei lumini care acoperă o suprafaţă mică în comparaţie cu dimensiunile hărţii,

limitând câmpul vizual al acestuia. Evident, la fel ca toţi parametrii ce caracterizează FPSController, şi proprietăţile 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 obţinerii 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 inteligenţă artificială - “creierul” entităţii, acesta va controla modul în care entitatea se deplasează, și va încheia jocul în anumite condiţii;
- 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 condiţiile optime pentru efectul provocat de acesta.

Sunetul este protagonistul acestei terori din mai multe motive, dintre care cel mai important este faptul că în evoluţia omului, precum şi în cazul multor alte mamifere, auzul a avut un rol major în mecanismul defensiv al individului [6]. Deşi era dormitului în peşteri şi a feritului de fiare sălbatice a fost de mult timp depăşită, instinctele în privinţa sunetului au rămas ascuțite, permițând astfel echipelor de sunet din cadrul proiectelor horror şi artiştilor să

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

În aplicaţia de faţă, "arhitectura" sunetului nu este una deosebit de complexă, dar este eficientă. În joc există trei surse de sunet: una inclusă în jucător (paşii acestuia), şi două ataşate entităţii negative. Cât timp entitatea nu este în vecinătatea jucătorului, acesta îşi va auzi doar paşii (fapt ce contribuie de asemenea la starea generală de nelinişte). În momentul în care aceasta începe să se apropie, însă, jucătorul va auzi treptat cele două surse de sunet: de la distanţă va auzi doar un sunet "ambiental", asemănător într-o oarecare măsură cu un efect sonor produs de vânt (acest sunet a fost obţinut utilizând un sintetizator audio numit Sytrus, folosit în contextul FL Studio); a doua sursă de sunet va fi auzită doar la o distanţă mai mică, pentru a ajuta jucătorul să plaseze entitatea în spaţiu (şi acest fapt contribuie la senzaţia generală de stres). Acest al doilea sunet a fost obţinut prin manipularea unei înregistrări a propriei mele voci, transformând-o într-un sunet inuman, oferindu-i şi un fond "ambiental" precum prima sursă de sunet, dar mai discret. Toate aceste sunete se vor suprapune şi vor creşte în intensitate cu cât distanţa dintre jucător şi entitate este mai mică.

Nu trebuie pierdut din vedere faptul că aceste sunete sunt redate într-un spaţiu tridimensional. Unity permite aplicarea efectului Doppler pe sunet, iar acest fapt are o importanţă majoră în crearea unei atmosfere neobişnuite. Pe scurt, efectul Doppler se referă la schimbarea pitch-ului (înălţimea) unui sunet în situaţia în care sursa care îl emite este în mişcare sau observatorul este în mişcare. Utilizarea efectului Doppler în situaţia de faţă contribuie considerabil la crearea unor sunete ca din altă lume.

Uşor remarcabil este faptul că puterea sunetului ar fi anulată dacă entitatea ar fi una statică: nu ar fi nimic înfricoşător dacă entitatea stă într-un singur loc. Aici intervine scriptul entităţii, pe care îl putem considera a fi "creierul" acesteia. Vom arunca, aşadar, o privire la BasicFollowerAI.

BasicFollowerAI este o altă clasă derivată din MonoBehaviour, în cadrul căreia se găsesc mai multe variabile şi implementările a trei metode: Start, Update, şi 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 prezenţa a două variabile publice de tip Transform, şi anume target şi follower. După cum sugerează şi numele acestor variabile, una va reprezenta obiectul urmărit (în

cazul de faţă playerul - FPSController) şi respectiv obiectul care urmăreşte targetul (StaffOfPain, în cazul de faţă). Următoarele două variabile configurează deplasarea propriu-zisă a entităţii: speed va fi viteza de deplasare, iar hover va ajuta entitatea să plutească deasupra pământului (contribuind la efectul fantomatic al prezenţei entităţii). Aceşti parametri pot fi modificaţi din interfaţa 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 primeşte valoarea true în condiţiile în care jocul este terminat în defavoarea jucătorului.

Metoda Start, moştenită din MonoBehaviour se va ocupa de iniţializarea 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 entităţi. Update, şi ea o funcţie moştenită din MonoBehaviour, este o metodă care se apelează pentru fiecare frame al aplicaţiei. În cadrul acestei metode se află implementarea deplasării entităţii negative. Modul în care aceasta funcţionează este următorul: în fiecare frame este calculată o nouă poziţie a entităţii, în funcţie de mai mulţi factori - principalul fiind poziţia 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, situaţia în care distanţa este mai mare decât 100 unităţi

Primul pas în această metodă este observarea distanţei între “victimă” şi urmăritor. În cazul în care aceasta este mai mare decât 100 de unităţi, variabila actualspeed primeşte valoarea 80. Astfel, dacă distanţa între cele două entităţi este suficient de mare, viteza entităţii negative creşte pentru a putea ţine pasul cu targetul, şi în acest mod se păstrează un ritm alert al jocului şi un nivel de tensiune ridicat. Deplasarea propriu-zisă se face prin orientarea entităţii cu faţa spre target folosind funcţia transform.LookAt şi folosirea funcţiei transform.Translate pentru a deplasa entitatea înainte (înainte va însemna în direcţia jucătorului, doarece obiectul este deja orientat cu faţa spre el) şi 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 distanţa are o valoare între 100 şi 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 menţionat până acum: posibilitatea de teleportare într-o poziţie pseudo-aleatorie pentru a induce confuzia jucătorului.

Modul în care funcţionează 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 aceeaşi 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 iniţial cu faţa spre jucător se va deplasa un număr mare de unităţi spre stânga, şi va avansa apoi şi în direcţia înainte - fără a se mai reorienta, acelaşi lucru se întâmplă simetric şi 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 distanţa dintre cele două entităţi este chiar mai mică şi decât 30, entitatea negativă se va deplasa pur şi simplu cu viteza normală direct înspre target, în modul descris în prezentarea situaţiei în care distanţa este mai mare decât 100 de unităţi.

```
        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 găseşte o verificare: dacă distanţa între cele două entităţi este mai mică decât valoarea de 5 unităţi, 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) şi pe ecran va fi afişat mesajul “You finally came to an END.” (care ar putea fi tradus prin “În sfârşit 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 şi cazul acestuia de faţă. Obiectul startup este doar un container al scriptului ce poartă acelaşi nume, care se ocupă de “orchestrarea” jocului ca un întreg: se va ocupa de generarea hărţii şi utilizarea unui timer care să permită jucătorului să câştige 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: definiţia clasei Startup

După cum se poate observa din definiţia clasei Startup, aceasta este şi 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 moştenite din MonoBehaviour: Awake, Update şi OnGUI, deci structura clasei este similară cu cea a entităţii 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 (descris în subcapitolul 5.2.1) care primeşte o referinţă la obiectul MapGenerator deja existent prin utilizarea funcţiei 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 stabileşte valoarea nulă.

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

6. Concluzii

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

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

Ştiind deja că produsul finit va reprezenta un joc 3D în care jucătorul controlează un personaj anonim într-un spaţiu întunecat şi mereu generat aleatoriu, ameninţat de o entitate cu intenţii maliţioase, am trecut la următoarea etapă, şi 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 modalităţilor în care puteau fi implementate conceptele stabilite, şi execuţia 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