Cuprins

Capitolul 1. Introducere – Contextul proiectului (Heading 1 style)

Rezumat la lucrare. Ultimul capitol scris.

Titlul capitolului se bazează pe Heading 1 style, numerotat cu o cifra (x. Nume capitol), font Times New Roman de 14, Bold.

Ce se scrie aici:

- Contextul
- Conturarea domeniului exact al temei
- reprezintă cca. 5% din lucrare

1.1. Contextul proiectului (Heading 2 style)

Fontul folosit implicit în acest document este Times New Roman, dimensiune de 12, conform *Normal style*, cu spaţiere la 1 rând (Paragraph, Line spacing de 1.0) şi *Justify*.

Pentru prima linie din fiecare paragraf se foloseşte indentare (implicit in *Normal Style*), iar între paragrafe succesive nu se lasă distanţă suplimentară.

1.1.1. (Heading 3 style)

Fiecare tabel introdus în lucrare este numerotat astfel: Tabel x.y, unde x reprezintă numărul capitolului iar y numărul tabelului din capitol. Se lasă un rând liber între tabel şi paragraful anterior, respectiv posterior.

Tabel 1.1 (Insert caption->Tabel)

Times new roman (12)	xxxx	xxxx	xxxx	

Fiecare figură introdusă în text este citată (de ex: în figura x.y este prezentată ...) şi numerotată. Numerotarea se face astfel Figura x.y unde x reprezintă numărul capitolului iar y numărul figurii în acel capitol. Folosiţi (Insert caption->Figura).

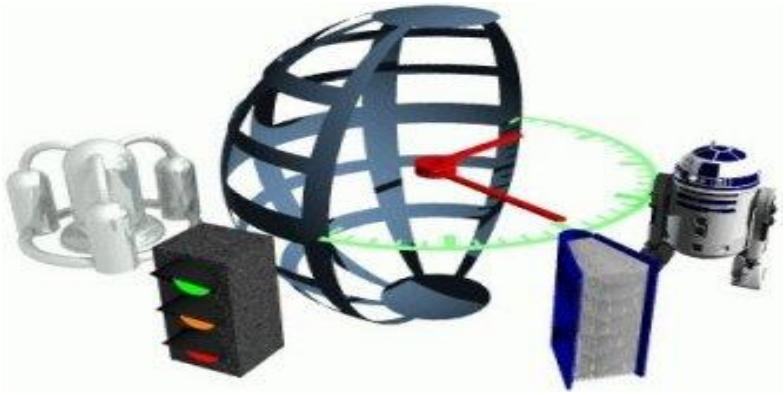


Figura 1.1 Numele figurii (insert->reference->caption->Figura)

Fiecare capitol începe pe pagină nouă.

Capitolul 2. Obiectivele Proiectului

Descrierea contextului + Justificarea necesitatii temei + definire tema cu obiective si scop.

În momentul de față ne aflăm într-un context care nu este foarte plăcut. Această perioadă în care toată lumea a stat mai mult în case, școlile au fost închise, iar accesul la lumea exterioară a fost foarte limitat, am ajuns să folosim din ce în ce mai mult calculatorul. Am ajuns să folosim calculatorul pentru a învăța de la distanță, pentru a ne plăti facturi, pentru a ne îndeplini datoria la locul de muncă și, în special, pentru a face activități recreaționale sau pentru a înlocui activitățile pe care le-am fi făcut, în mod normal, afară.

Multă lume se recrează prin intermediul jocului și prin satisfacerea curiozităților. Există o mulțime de companii sau persoane individuale care fac jocuri pentru toată lumea. În ultima vreme a aparut o creștere în popularitatea jocurilor VR, adică jocuri care pot fi jucate folosind un set de ochelari VR. Aceste jocuri oferă o cu totul altă dinamică jucatului pe calculator. În loc să stăm pe un scaun, în fața unui ecran, putem sta în picioare și să ne mișcăm liber datorita setului VR. Pe deasupra, mediul oferit de această tehnologie este mult mai imersiv, apropiindu-se foarte mult de realitate. Asftel, cu toate că ne aflăm într-un mediu restrâns, și anume, încăperea noastră, putem sa facem mișcare, putem să ne transpunem în orice realitate dorim și putem să ne distrăm pe cinste în tot acest timp. Pe lângă aceste jocuri VR, există și multe alte jocuri foarte populare. Dintre acestea, cele mai populare și cele mai jucate sunt cele generate procedural. Este și normal să fie așa. Persoanele se plictisesc destul de repede când vine vorba de a face un lucru repetitiv. De aceea, jocurile ca Minecraft, Terraria, Don’t Starve, etc. sunt unele din jocurile cele mai jucate și cele mai vândute din toate timpurile. Toate acestea au un element comun, și anume, statistic niciodată, atunci când începi un joc nou, nu vei avea aceași experiență pe care ai avut-o în alte momente în care ai jucat acel joc. Acest lucru face ca jucătorii să fie în continuare interesați de joc, aducând mai multă popularitate acestuia sau mai multe venituri prin intermediul microtranzacțiilor.

Date fiind aceste lucruri, tema mea, și anume, simularea unei lumi subacvatice în VR, combină aceste lucruri. Aplicația va simula o lume subacvatică realistă și generată procedural. Persoanele vor putea sa interactioneze cu mediu înconjurator și sa îl navigheze prin intermediul unui set VR, lucru care îi va oferi o experiență de neuitat. Obiectivele aplicației sunt următoarele: generarea procedurală a terenului de pe fundul unei mări sau al unui ocean, simularea vieții subacvatice (peștii și mișcarea realistă în bancuri), implementarea controlului și interacțiunii folosind ochelarii VR, animarea lumii prin intermediul shaderelor pentru a-i oferi credibilitate, aplicarea de shadere camerei și spațiului vizual pentru a simula cu acuratețe lumea subacvatică.

Scopul aplicației este oferirea unei activități plăcute care se poate desfășura în casă, unei baze pentru dezvoltarea unui joc de magnitudinea celor mai vândute jocuri și

oferirea unui mod mai interesant decat un videoclip pentru învățarea lucrurilor despre lumea subacvatică, adică un scop educațional.

În acest capitol se prezintă tema propriu zisă (sub forma unei teme de proiectare/cercetare, formulată exact, cu obiective clare – 2-3 pagini şi eventuale figuri explicative).

Reprezintă cca. 10% din lucrare.

Capitolul 3. Studiu Bibliografic

Generalitati despre VR/AR + Generalitati mediu acvatic + Aplicatii cunoscute in domeniu (+ alg) + Studiu comparativ al acestor aplicatii – avantaje/dezavantaje

Realitatea virtuală (VR) este o experiență simulată care poate fi similară sau complet diferită de lumea reală. Aplicațiile realității virtuale includ divertismentul (de exemplu, jocuri video), educația (de exemplu, instruire medicală) și afaceri (de exemplu, întâlniri virtuale). Alte tipuri distincte de tehnologie în stil VR includ realitatea augmentată și realitatea mixtă , uneori denumită realitate extinsă sau XR.

Se poate distinge între două tipuri de VR; VR captivant și VR în rețea bazat pe text (cunoscut și sub denumirea de „Cyberspace"). VR imersiv vă schimbă vizualizarea, atunci când vă mișcați capul. În timp ce ambele VR-uri sunt adecvate pentru instruire, Cyberspace este preferat pentru învățarea la distanță. În unele cazuri, aceste două tipuri sunt chiar complementare unele cu altele. Această pagină se concentrează în principal pe VR-ul captivant.

O metodă prin care realitatea virtuală poate fi realizată este realitatea virtuală bazată pe simulare . Simulatoarele de conducere, de exemplu, oferă șoferului la bord impresia că conduce efectiv un vehicul real prin prezicerea mișcării vehiculului cauzată de intrarea conducătorului auto și prin readucerea la conducere a indicațiilor vizuale, de mișcare și audio corespunzătoare.

Cu realitatea virtuală bazată pe imaginea avatarului , oamenii se pot alătura mediului virtual sub forma unui videoclip real, precum și a unui avatar. Se poate participa la mediul virtual distribuit 3D sub forma unui avatar convențional sau a unui videoclip real. Utilizatorii își pot selecta propriul tip de participare pe baza capacității sistemului.

În realitatea virtuală bazată pe proiectoare, modelarea mediului real joacă un rol vital în diferite aplicații de realitate virtuală, cum ar fi navigația robotului, modelarea construcțiilor și simularea avionului. Sistemele de realitate virtuală bazate pe imagini au câștigat popularitate în comunitățile de grafică pe computer și viziune computerizată . În generarea de modele realiste, este esențial să înregistrați cu precizie datele 3D dobândite; de obicei, o cameră este utilizată pentru modelarea obiectelor mici la mică distanță.

Realitatea virtuală bazată pe desktop implică afișarea unei lumi virtuale 3D pe un ecran de birou obișnuit, fără a utiliza niciun echipament specializat de urmărire a poziției VR . Multe jocuri video moderne la prima persoană pot fi utilizate ca exemplu, folosind diverse declanșatoare, personaje receptive și alte astfel de dispozitive interactive pentru a face utilizatorul să se simtă ca și cum ar fi într-o lume virtuală. O critică obișnuită a acestei forme de imersiune este că nu există nici un sentiment de vedere periferică , limitând capacitatea utilizatorului de a ști ce se întâmplă în jurul lor.

Mediul subacvatic se referă la regiunea de sub suprafața și scufundată în apă lichidă într-o caracteristică naturală sau artificială (numită corp de apă), cum ar fi un ocean , mare , lac , iaz , rezervor , râu , canal sau acvifer . Unele caracteristici ale mediului subacvatic sunt universale, dar multe depind de situația locală.

Apa lichidă a fost prezentă pe Pământ pentru cea mai mare parte a istoriei planetei . Se consideră că mediul subacvatic este locul originii vieții pe Pământ și rămâne regiunea ecologică cea mai critică pentru susținerea vieții și a habitatului natural al majorității organismelor vii. Mai multe ramuri ale științei sunt dedicate studiului acestui mediu sau a unor părți sau aspecte specifice ale acestuia.

O serie de activități umane sunt desfășurate în părțile mai accesibile ale mediului subacvatic. Acestea includ cercetare, scufundări subacvatice pentru muncă sau recreere și război subacvatic cu submarine. Cu toate acestea, mediul subacvatic este ostil oamenilor în multe feluri și deseori inaccesibil și, prin urmare, relativ puțin explorat.

Documentare bibliografica are ca obiectiv prezentarea stadiului actual al domeniului/sub-domeniului în care se situează tema. În redactarea acestui capitol (în general a întregului document) se va ţine cont de cunoștințele acumulate la disciplinele dedicate din semestrul 2, anul 4 (Metodologia Întocmirii Proiectelor, etc.), precum si la celelalte discipline relevante temei abordate.

Acest capitol reprezintă cca. 15% din lucrare.

Referinţele se scriu în secţiunea *Bibliografie*. Formatul referinţelor trebuie sa fie de tipul *IEEE* sau asemănător. Introducerea şi formatarea referinţelor în bibliografie, respectiv citarea în text, se poate face manual sau folosind instrumentele de lucru menţionate în ultimele paragrafe din acest capitol.

In secţiunea *Bibliografie* sunt exemple de referinţe pentru articol la conferinţe sau seminarii [1], articol în jurnal [2], sau cărţi [3]. Referinţele spre aplicaţii sau resurse online (pagini de internet) trebuie sa includă cel puţin o denumire sugestivă pe lângă link-ul propriu zis [4], plus alte informaţii dacă sunt disponibile (autori, an, etc.). Referinţele care prezintă doar link spre resursa online se vor plasa în footer-ul paginii unde sunt referite.

Citarea referinţelor în text este obligatorie, vezi exemplul de mai jos (în funcţie de tema proiectului se poate varia modul de prezentare a metodei/aplicaţiei).

În articolul [1] autorii prezintă un sistem pentru detecţia obstacolelor în mişcare folosind stereoviziune şi estimarea mişcării proprii. Metoda se bazează pe …*trecere în revistă a algoritmilor, structurilor de date, funcţionalitate, aspecte specifice temei proiectului etc*….. Discuţie *avantaje – dezavantaje*.

Instrumentele de lucru pentru **MS Word 2003** şi instrucţiuni de folosire găsiţi la:

Pentru **MS Word 2007** şi **MS Word 2010** se poate folosi sistemul integrat de gestiune bibliografiei, *References, Citations & Bibliography*. Mai multe informaţii se găsesc în documentaţia online de la MS Office.

Capitolul 4. Analiză şi Fundamentare Teoretică

Algoritmii folositi +
Scheme, etc. Nu fac referire la implementare +

În cadrul acestei aplicații, pentru a oferi o experiență cât mai convingătoare, a trebuit să folosesc algoritmi pentru generarea dinamică a mediului înconjurător, pentru crearea comportamentului realist al peștilor, pentru simularea unui mediu subacvatic convingător și pentru interacțiunea cu mediul VR.

4.1. Generarea dinamica a mediului

4.1.1. Informații generale

Pentru generarea dinamică a mediului înconjurator se folosește algoritmul Marching-cubes. Algoritmul Marching Cubes a fost publicat in anul 1987 de către Lorensen și Cline cu scopul de a extrage o suprafață poligonala a unei isosuprafețe dintr-un spațiu tridimensional discret. Aplicațiile acestuia se extind în diferite domenii. De exemplu, în domeniul medical este folosit pentru scanarea imaginilor MRI și CT și oferirea unei vizualizări tridimensionale a mai multor imagini scanate de-a lungul craniului unei persoane.

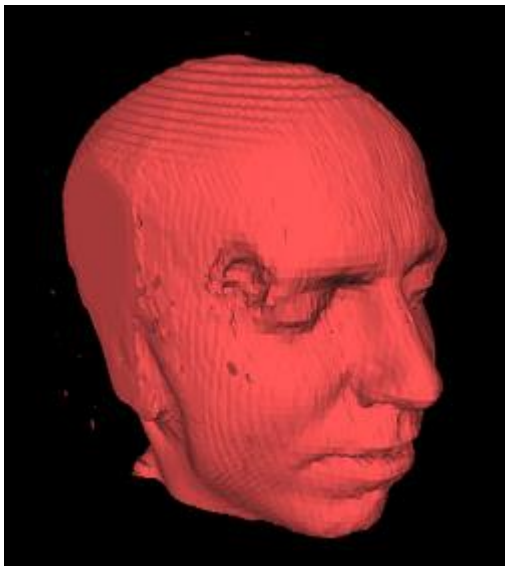


Figure 1 https://en.wikipedia.org/wiki/Marching_cubes

In domeniul design-ului, algoritmul este folosit pentru modelare 3D prin intermediul așa numitelor metaballs sau a altor metasuprafețe și pentru efecte speciale.

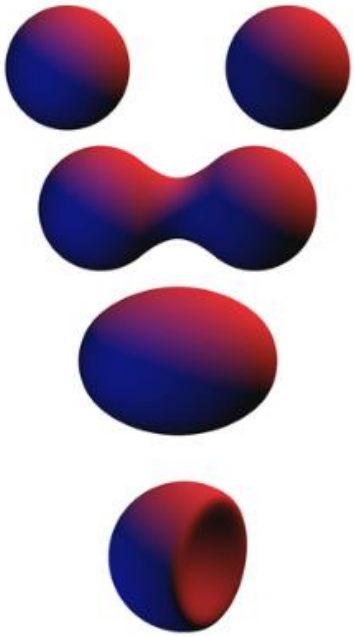


Figure 2 https://en.wikipedia.org/wiki/Metaballs

4.1.2. Algoritmul propriu zis

Acest algoritm rezolvă problema formării unei fețe aproximate la o isosuprafață (o suprafata constantă dată) printr-un câmp scalar redus la o matrice tridimensională. Considerând faptul că fiecare astfel de celulă tridimensională va fi reprezentată prin vârfuri și valori scalare la fiecare dintre aceste vârfuri, este necesar sa creăm o reprezentare cât mai apropiată a isosuprafeței. Astfel, isosuprafața poate să treacă prin celulă (printr-un vârf sau prin mai multe laturi create de aceste vârfuri), poate sa treacă pe deasupra ei sau pe dedesuptul ei.

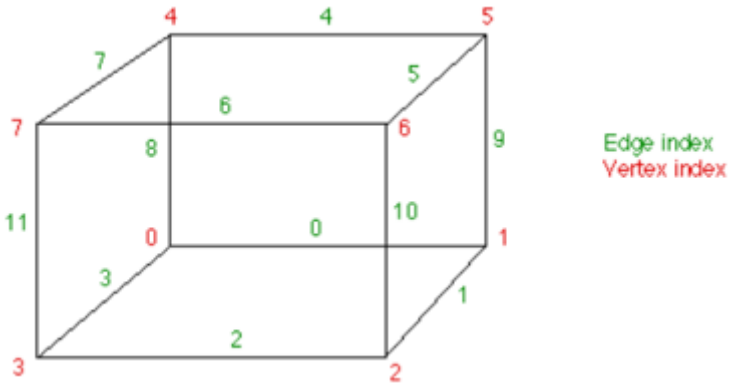


Figure 3 http://paulbourke.net/geometry/polygonise/

Pentru a reprezenta fiecare posibilitate, caracterizarea acesteia se va face prin numărul de vârfuri care au valori deasupra sau dedesuptul isosuprafeței. Prin această metodă putem ușor să ne dăm seama dacă o isosuprafață trece prin o latură generată de catre doua vârfuri. De exemplu, dacă isosuprafața trece deasupra unui vârf si dedesuptul altui vârf, iar vârfurile sunt adiacente, atunci putem spune că isosuprafața trece prin latura dată de aceste vârfuri.

Exemplu:

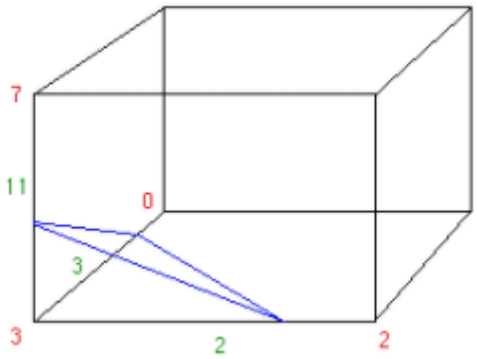


Figure 4 http://paulbourke.net/geometry/polygonise/

Cea mai dificilă parte a acestui algoritm este numărul mare de posibilități de combinare a fețelor pentru fiecare solutie astfel încât acestea să se conecteze corect cu celulele alăturate. Există un numar de 256 de combinații. (15 combinatii si permutarile acestora).

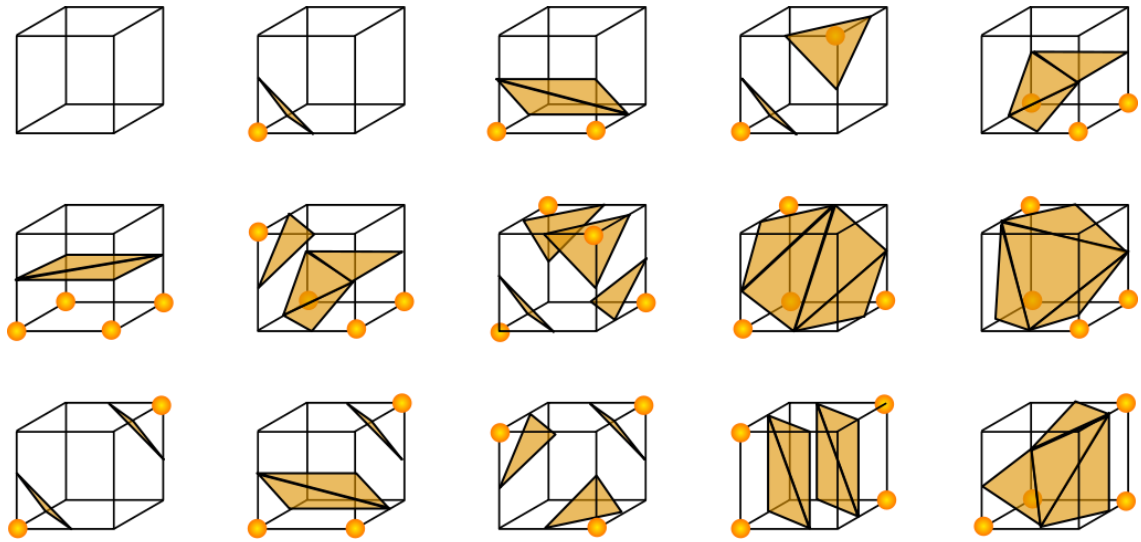


Figure 5 Combinatiile de baza

Pentru ca această parte să fie eficientă, se va folosi un tabel în care vom avea stocate toate laturile prin care poate să treacă isosuprafața. După cum am spus înainte, putem afla ușor prin ce laturi trece isosuprafața dacă știm pozitia acesteia față de vârfurile matricei. Acest lucru se determină astfel:

Considerand *cubeindex* o variabilă de 8 biți, se realizează, în funcție de isosuprafață operația de sau logic cu 2 la puterea indexului vârfului în cazul în care acesta este sub suprafață.

cubeindex = 0;
dacă vârf[0] < isosuprafață atunci cubeindex |= 1;
dacă vârf[1] < isosuprafață atunci cubeindex |= 2;
dacă vârf[2] < isosuprafață atunci cubeindex |= 4;
dacă vârf[3] < isosuprafață atunci cubeindex |= 8;
dacă vârf[4] < isosuprafață atunci cubeindex |= 16;
dacă vârf[5] < isosuprafață atunci cubeindex |= 32;
dacă vârf[6] < isosuprafață atunci cubeindex |= 64;
dacă vârf[7] < isosuprafață atunci cubeindex |= 128;

După aceste comparații se construiește variabila *cubeindex*. Biții acesteia vor fi folosiți pentru accesarea informației din tabelul de laturi (care va returna un număr de 12 biți corespunzători celor 12 laturi sau 0/0xff în cazul în care nu s-a trecut prin nicio latură). Dacă luăm exemplul din Figura 4, singurul vârf de sub isosuprafață este 3, deci cubeindex = 0000 1000, care este echivalent cu 8, iar elementul din tabela de laturi de la pozitia 8 va avea valoarea 1000 0000 1100 deoarece isosuprafața trece prin laturile 2, 3 și 11 (fiecare latura corespunde de la dreapta la stânga cu o cifra binara. Valoarea 0 ne spune că nu se trece prin latura cu acel indice, iar valoarea 1 ne spune că se trece prin latura cu acel indice. În final, pentru a avea o precizie cât mai bună, se calculează intersecția punctelor prin interpolare liniară. Astfel obținem formula:

$$\frac{X - X1}{X2 - X1} = \frac{Y - Y1}{Y2 - Y1}$$

X1, X2 si Y1, Y2 sunt coordonatele vârfurilor ce alcătuiesc latura intersectată, iar X si Y sunt coordonatele punctului de intersecție a isosuprafeței cu latura.

Odată ce am calculat punctele de intersecție putem să trecem la ultima parte a acestui algoritm. Trebuie sa formăm fețele corecte din pozitiile pe care le intersecteaza isosuprafata cu marginile cubului. Se foloseste un tabel de triunghiuri (triTable) în care sunt stocate atâtea fețe de triunghiuri câte sunt necesare pentru a alcătui acea suprafață. Un element dintr-un astfel de tabel este reprezentat de o listă de vârfuri care, luate câte trei, formează triunghiuri. De exemplu, un astfel de element ar fi:

{11, 2, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}

Valoarea -1 este folosită în cazul în care nu avem deloc triunghiuri de la un punct anume. Această valoare semnalează faptul că triunghiurile s-au terminat. Pot fi maxim 5 astfel de triunghiuri, deci un element din acest tabel va fi alcătuit mereu din 15 numere (3 laturi * 5 triunghiuri) + o valoare -1 pentru oprire. În acest exemplu am avut un singur triunghi, și anume, triunghiul dat de vârfurile de pe laturile 2, 3, 11.
Putem căuta acum după variabila *cubeindex* acest element in tabelul de triunghiuri.

Rezultatul acestui algoritm în implementarea mea este acesta:

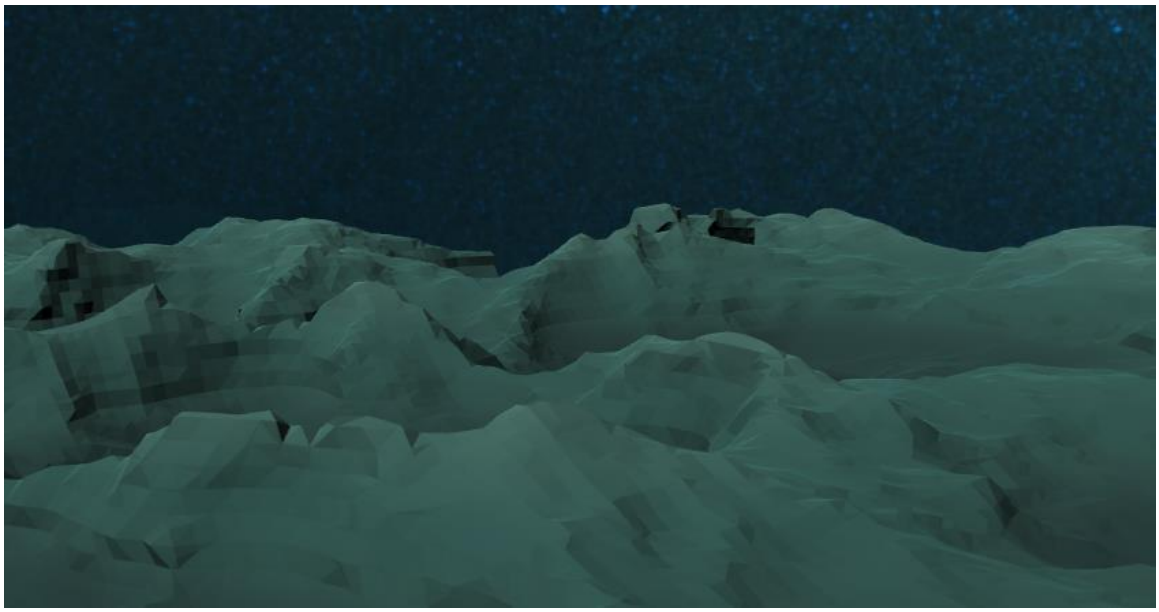


Figure 6 Mediu generat

Acest rezultat, folosind Marching Cubes, este unul mult mai bun și mai realist decât un rezultat pe care l-am obținut inițial, cand am încercat generarea suprafeței folosind un simplu zgomot perlin.

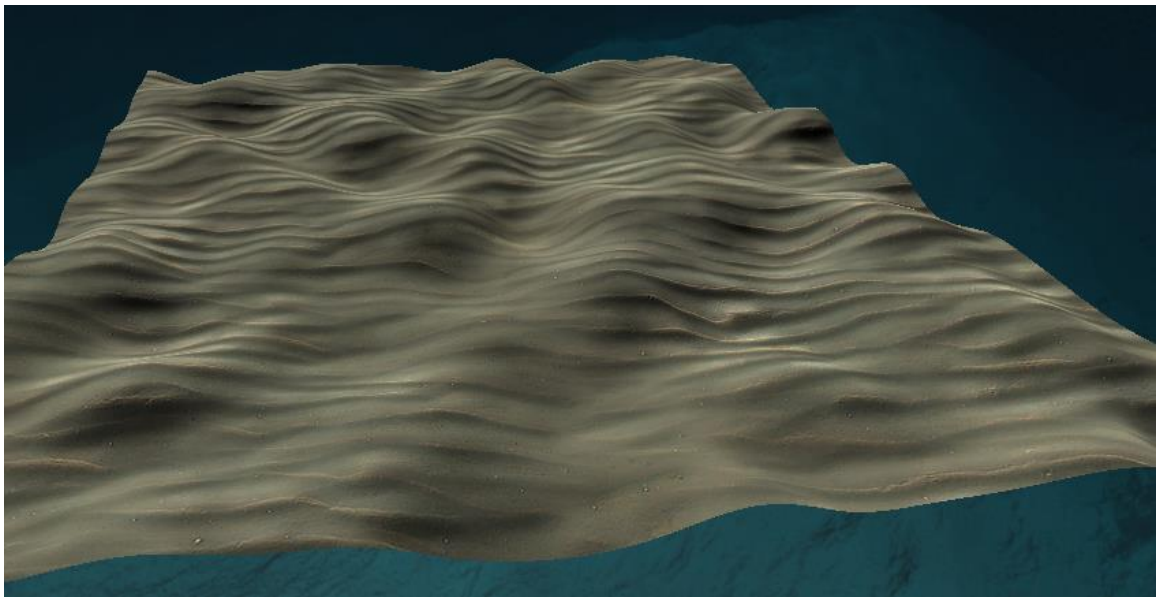


Figure 7 Rezultat folosind zgomot perlin

4.2. Comportamentul realist al peștilor

4.2.1. Informații generale

Acest comportament simulează prin intermediul algoritmului de boizi. Algoritmul de Boizi simulează viața artificială. Acesta a fost dezvoltat de către Craig Reynolds în anul 1986. Principala intentie a acestui algoritm este simularea efectului de cârd pe care îl au păsările, sau, în cazul de față, efectul de banc de pești. Numele de *Boid* provine de la versiunea pe scurt a unui obiect *Bird-oid*, adică, un obiect care se comportă ca și o pasăre.

La fel ca și algoritmul anterior, și acest algoritm are aplicabilități în mai multe domenii. De exemplu, este folosit în jocuri si proiecte grafice pentru a simula păsări, animale marine, turme de animale terestre, etc. Alte exemple ar fi ‘swarm robotics’ și animație.

4.2.2. Algoritmul propriu zis

Algoritmul funcționează ca și majoritatea algoritmilor de simulare a vieții artificiale. Astfel, complexitatea acestui algoritm apare doar atunci cand apare interacțiune între agenți individuali care urmează niște reguli simple. Un astfel de algoritm, putem spune că are un comportament *emergent*.

Un boid trebuie să urmeze trei reguli simple:

- **separare** – să își modifice direcția, astfel încât să nu se lovească de alți boizi vecini

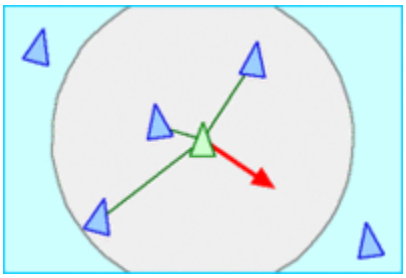


Figure 8 https://en.wikipedia.org/wiki/Boids

- **aliniere** – să se orienteze și sa urmeze direcția dată de media direcțiilor boizilor vecini

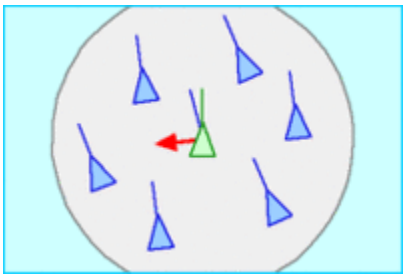


Figure 9 https://en.wikipedia.org/wiki/Boids

- **coeziune** – să încerce sa meargă spre poziția centrului de masă al boizilor vecini

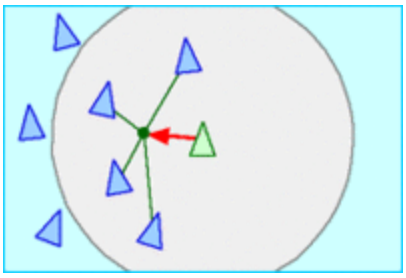


Figure 10 https://en.wikipedia.org/wiki/Boids

Pentru realizarea funcționalității acestui algoritm, dacă ar fi să luăm în calcul doar domeniul bidimensional, fiecare boid ar trebui să arunce niște raze în jurul lui și să folosească aceste raze pentru detecția boizilor alăturați. Odată ce boidul detectează boizii de lângă el, algoritmul devine simplu deoarece boidul trebuie doar sa-și ajusteze direcția în funcție de vecinii lui, urmând cele trei reguli: separare, aliniere și coeziune.

Pentru generarea razelor într-un mediu bidimensional, soluția este una foarte simplă. Se alege un cerc în jurul boidului și se trag raze la distanțe egale pe cercul cu boidul in centru. Se poate alege și zona pe care o vede boidul (field of view).

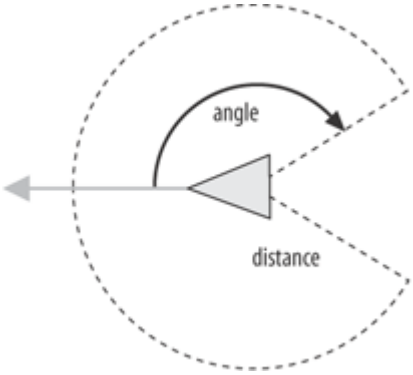


Figure 11 https://www.gamecareerguide.com/features/312/book_excerpt_killer_game_.php

În schimb, pentru spațiul 3D lucrurile devin puțin mai complicate. Nu mai generăm doar un cerc. Trebuie să generam o sferă în jurul boidului. Problema este că avem nevoie de o distribuție egală a punctelor pe sferă. Pentru a obține o astfel de distribuție, am ales să mă folosesc de *rația de aur* care este regăsită și în lumea reală, de exemplu, la distribuția semințelor de floarea soarelui.

Figure 12 Distributia semintelor de floarea soarelui

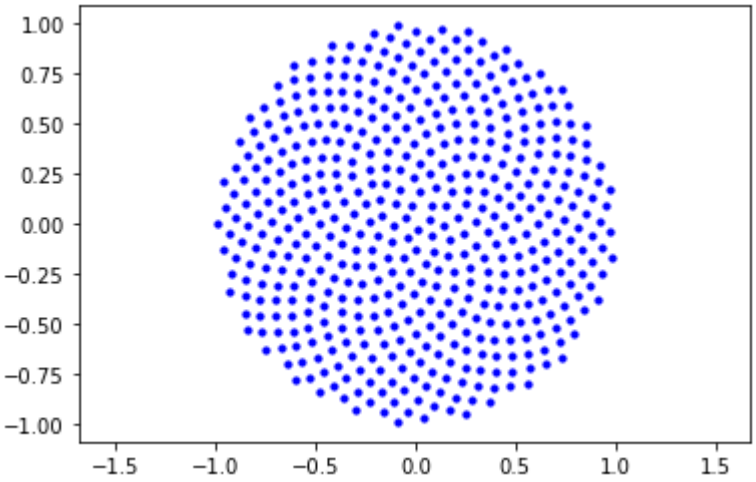


Figure 13 Rația de aur aplicată in 2D

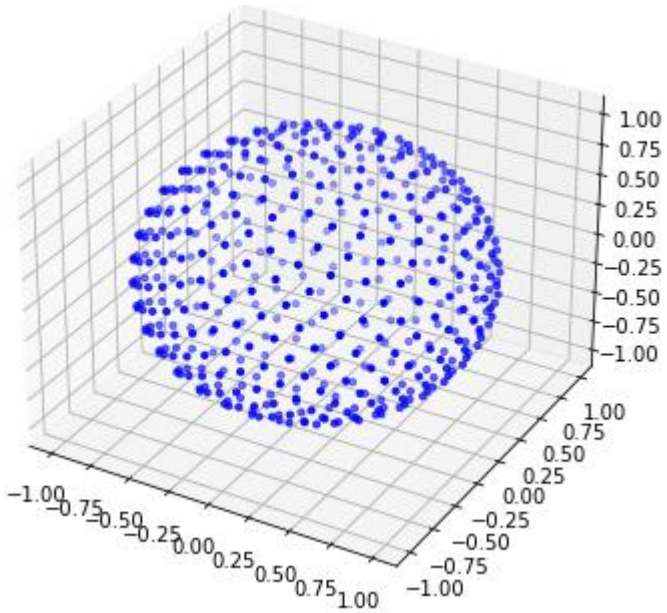


Figure 14 Rația de aur aplicată in 3D

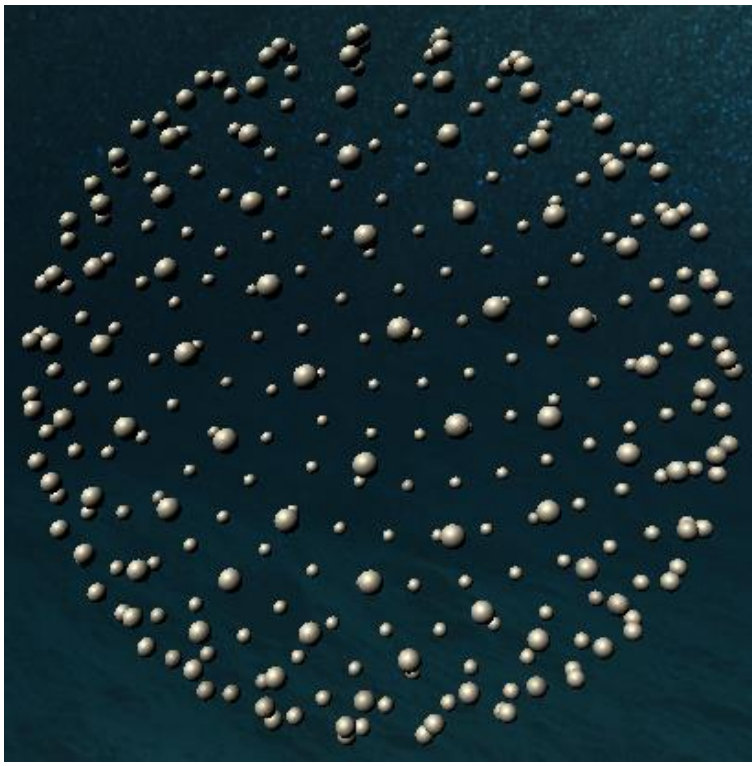


Figure 15 Sfera generată în Unity

Această soluție a fost cel mai ușor de implementat. Există mai multe modalități de generare a punctelor pe o sferă. Trei dintre acestea sunt: simularea electronilor, Hypercube Rejection și Aproximare Spirală. Simularea electronilor se refera la, dupa cum spune numele, o simulare a unor electroni (puncte) pe o sferă care se resping. Respingându-se, aceștia vor tinde să stea la o distanță maximă unul față de celălalt, iar distribuția acestora ajunge să fie egală pe sferă. Hypercube Rejection este o metodă simplă prin care se distribuie egal niște puncte într-un cub, iar apoi se elimină toate punctele care nu fac parte din sferă. Aproximarea Spirală este tragerea unei spirale în jurul sferei. Atunci toată problema se reduce la alegerea unor puncte echidistante pe o linie.

Odată ce avem aceste puncte generate, aplicăm aceleași reguli adăugând la ele o dimensiune în plus, selectăm un field-of-view, iar comportamentul boizilor va fi corect.

Împreună cu capitolul următor trebuie sa reprezinte aproximativ 60% din total.

Scopul acestui capitol este de a explica principiile funcţionale ale aplicaţiei implementate. Aici se va descrie soluţia propusă dintr-un punct de vedere teoretic - explicaţi şi demonstraţi proprietăţile şi valoarea teoretică:

- algoritm utilizat sau propus,
- protocoale utilizate,
- modele abstracte,
- explicaţii/argumentări logice ale soluţiei alese,

- structura logică şi funcţională a aplicaţiei.

NU SE FAC referiri la implementarea propriu-zisă.

NU SE PUN descrieri de tehnologii preluate cu copy-paste din alte surse sau lucruri care nu ţin strict de proiectul propriu-zis (materiale de umplutură).

Capitolul 5. Proiectare de Detaliu si Implementare

Cum am facut implementare +
De ce Unity? +
Descrise shadere +
In general toate detaliile +

5.1. De ce am ales Unity?

5.1.1. Introducere

Unity este una dinte cele mai bune platforme pentru crearea conținutului interactiv în timp real. Unele dintre cele mai importante facilități ale acestui *Game Engine* sunt: Oferirea suportului pentru mai multe platforme, 2D-3D-VR-AR, costul, calitatea graficii, magazinul de bunuri, comunitatea puternică, managerul de pachete. Aceste exemple sunt minimale, așa că o să intru în detaliu în următoarele subcapitole.

5.1.2. Costul

Există patru tipuri de planuri ce pot fi alese pentru Unity: **Personal, Plus, Pro, Enterprise**. Ultimele trei variante trebuie plătite, dar prima variantă, anume, Unity Personal, este una mai mult decât suficientă pentru orice creator independent și e, de asemenea, gratuită. Această versiune pune la dispoziție un procentaj foarte mare din funcționalități, lipsind doar accesul la codul sursă al engine-ului și capacitatea creării unei licențe de server. Acest plan poate fi folosit pentru proiecte care au produs până 100000$ în ultimele 12 luni.

5.1.3. Managerul de pachete

Managerul de pachete ajută la instalarea rapidă de module sau bunuri oferite în mod oficial sau de la comunitate. Pachetele sunt extrem de variate și de multe ori se poate găsi aproape orice este nevoie. Managerul de pachete oferă posibilitatea alegerii versiunii unui pachet, accesarea documentației acestuia și multe altele.

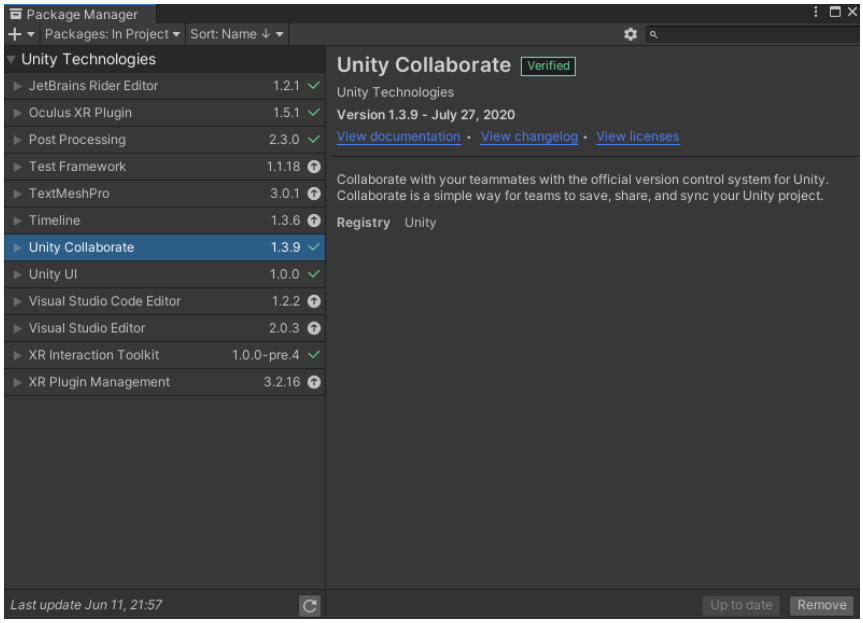


Figure 16 Managerul de pachete

5.1.4. 2D-3D-VR-AR

Unity pune la dispoziție posibilitatea de dezvoltare în orice mediu dorim. Se pot crea lumi 2D, lumi 3D, realitate virtuală și realitate augmentată. Unity pune la dispoziție foarte multe elemente ajutătoare pentru a realiza astfel de proiecte.

De exemplu, pentru mediul 2D și 3D există collidere care pot fi utilizate pentru a realiza interacțiunea ușoară între obiecte. Există posibilitatea de a adăuga elemente de fizică obiectelor precum gravitația, accelerația, viteza, etc. Texturile pot fi foarte ușor aplicate pe un obiect prin intermediul shaderelor. Unity oferă o interfață foarte prietenoasă pentru shadere, și anume, materialele. Un material poate fi atașat unui obiect, iar acel material va avea un shader. Materialul este cel care trimite parametri din interfață către shader. Shaderul va schimba apoi aspectul obiectului simplu 3D, fie aplicând o textură, fie deformând-o, etc.

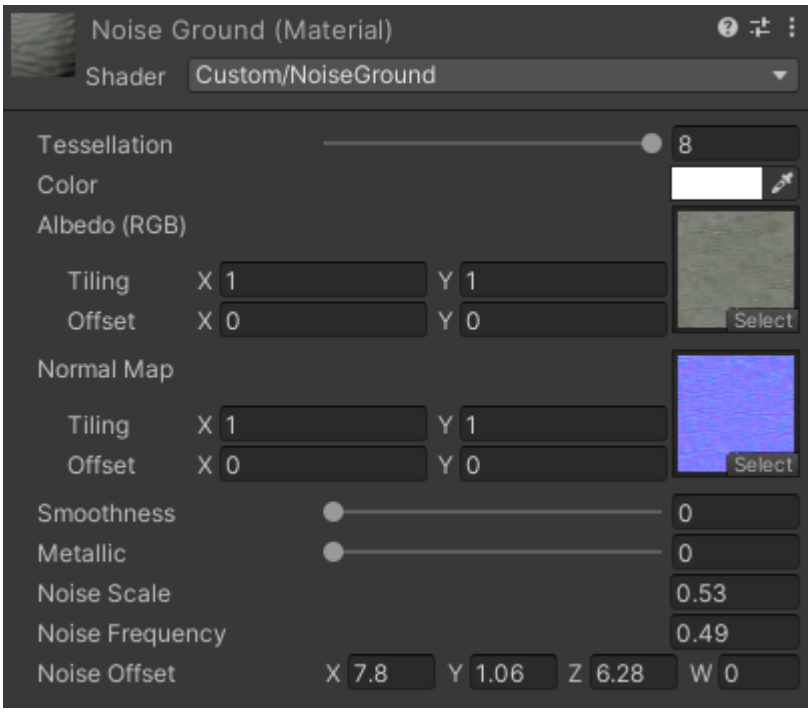


Figure 17 Interfața unui material

Ba mai mult, versiunile mai noi de Unity oferă posibilitatea de adăugare a shaderelor grafice. Un shader grafic face același lucru ca și un shader normal, doar că oferă o interfață grafică a variabilelor și elementelor care în mod normal ar trebui să fie cod.

```
void surf (Input IN, inout SurfaceOutputStandard o)
{
    // Albedo comes from a texture tinted by color
    fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
    o.Albedo = c.rgb;

    // Caustics sampling
    fixed2 uv = IN.uv_MainTex * _Caustics1_ST.xy + _Caustics1_ST.zw;
    uv += _Caustics1_Speed * _Time.y;

    fixed2 uv2 = IN.uv_MainTex * _Caustics2_ST.xy + _Caustics2_ST.zw;
    uv2 += _Caustics2_Speed * _Time.y;

    fixed s = _SplitRGB;
    fixed r = tex2D(_CausticsTex, uv + fixed2(+s, +s)).r;
    fixed g = tex2D(_CausticsTex, uv + fixed2(+s, -s)).g;
    fixed b = tex2D(_CausticsTex, uv + fixed2(-s, -s)).b;

    fixed3 caustics1 = fixed3(r, g, b);

    r = tex2D(_CausticsTex, uv2 + fixed2(+s, +s)).r;
    g = tex2D(_CausticsTex, uv2 + fixed2(+s, -s)).g;
    b = tex2D(_CausticsTex, uv2 + fixed2(-s, -s)).b;

    fixed3 caustics2 = fixed3(r, g, b);
```

Figure 18 Parte dintr-un shader normal

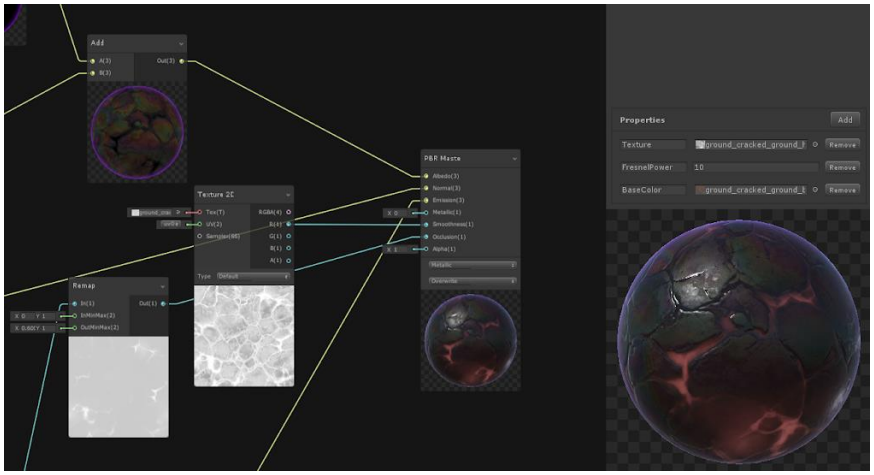


Figure 19 Parte dintr-un shader graph

5.1.5. VR și AR

Pentru a începe dezvoltarea în realitate virtuală sau augmentată, managerul de pachete ne stă la dispoziție cu module foarte avansate ce se ocupă de asta. Aceste module permit implementarea cu ușurință a funcționalităților de bază pentru VR. De exemplu, ne este foarte ușor să creem o zonă în care poate acționa un utilizator de VR și o foarte sugestivă interfață pentru controlarea manetelor și butoanelor de pe acestea.

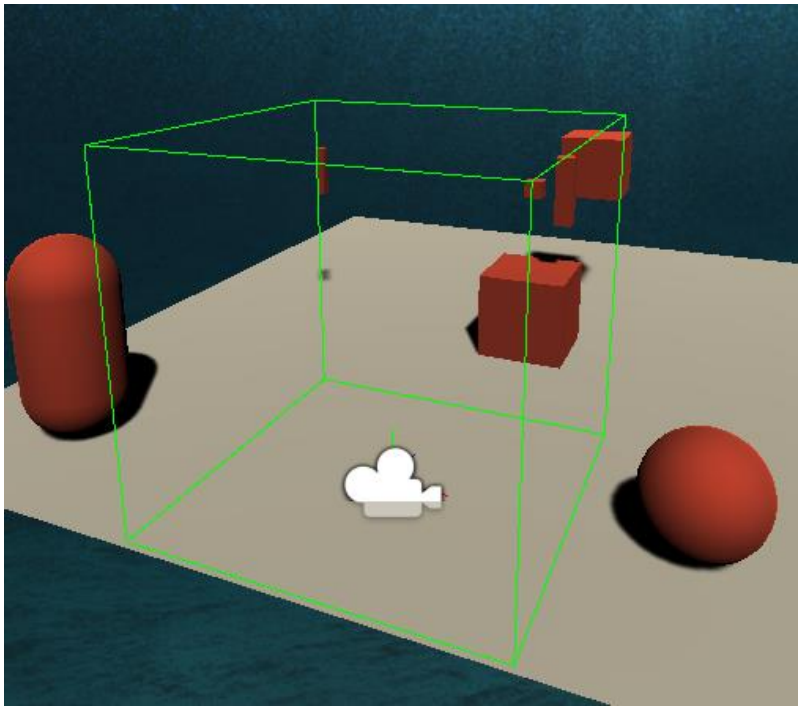


Figure 20 Zona în care poate acționa un jucător

5.1.6. Suportul pe mai multe platforme

Una dinte cele mai bune elemente oferite de către Unity este exportarea aceleiași aplicații pentru diferite sisteme de operare. De exemplu, putem crea o aplicație care să

funcționeze atât pe Windows, cât și pe Mac, Linux, console (PS, Xbox, etc.) sau chiar sisteme de operare pentru telefoane, cum ar fi Android. Exportarea unei aplicații pentru telefon în locul unei aplicații pentru calculator se poate face printr-o simplă apăsare de buton, pe când, în alte medii ar fi nevoie de zile sau săptămâni întregi de dezvoltare pentru asta.

Figure 21 Selectarea platformei de exportare

Un alt tip de compatibilitate este oferit pentru VR. În majoritatea cazurilor contează ce tip de headset este folosit. În cazul Unity, poate fi pur și simplu selectată compatibilitatea dorită.

Figure 22 Selectarea compatibilității VR

Împreună cu capitolul precedent reprezintă aproximativ 60% din total.

Scopul acestui capitol este de a documenta aplicaţia dezvoltată în aşa fel încât dezvoltarea şi întreţinerea ulterioară să fie posibilă. Cititorul trebuie să identifice funcţiile principale ale aplicaţiei din ceea ce este scris aici.

Capitolul ar trebui sa conţină (nu se rezumă neapărat la):

- schema generală aplicaţiei,
- descriere a fiecărei componente implementate, la nivel de modul,
- diagrame de clase, clase importante şi metode ale claselor importante.

5.2. Shadere

O parte foarte importantă pentru sugerarea realismului este utilizarea de shadere. Evident, am putea avea o scenă care să conțină doar elementele 3D care ar urma în mod corect execuția algoritmilor, dar diferența ar fi evidentă. Peștii nu ar fi deloc animați, ar fi doar obiecte statice care ar mișca dintr-o direcție în alta. Efectul subacvatic nu ar exista. Fără acel efect, un jucător ar putea considera că scena își are loc pe lună sau pe o planetă îndepărtată, mai ales în stadiul acesta al aplicației în care exista o lipsă a vegetației. Să nu mai vorbim de cel mai important lucru, culoarea. Scena nu ar avea culoare fără ajutorul acestora.

5.2.1. Shaderul pentru animația peștilor

Algoritmul a fost inspirat de https://www.bitshiftprogrammer.com/2018/01/how-to-animate-fish-swimming-with.html .

Acest shader transformă un simplu model de pește într-unul care, după anumiți parametri dați, simulează înotul.

Figure 23 Modelul normal al unui pește

Figure 24 Animația pe care o face shaderul

Pe lângă această animație shaderul oferă si posibilitatea schimbării culorii peștilor după un parametru de culoare.

Figure 25 Pești de diferite culori

Cea mai importantă parte din cadrul acestui shader este modificarea pozițiilor vârfurilor în funcție de timp. În acest mod, putem decide pe fiecare axă cât de mult să oscileze peștele. Dacă vrem ca peștele să își miște coada, va trebui să creștem frecvența sau amplitudinea pe axa Z. Dacă vrem ca peștele să înoate sub forma unor arcuri, putem crește amplitudinea pe axa Y. Daca vrem ca peștele să își schimbe direcția în stânga și în dreapta, crestem amplitudinea sau frecvența pe axa X. De asemenea, putem selecta, în funcție de modelul pe care îl avem, vârful de la care începe coada. Astfel peștele va mișca pe axa Z doar coada, simulând înotul.

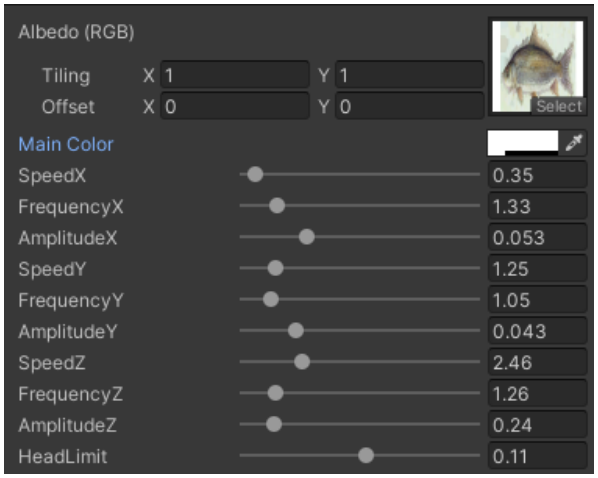


Figure 26 Materialul pentru controlul animației peștelui

5.2.2. Shader pentru efect subacvatic

Acest shader distorsionează elementele apropiate din cameră pentru a sugera mișcarea apei. Distanța de distorsionare poate fi schimbată, iar nivelul și frecvența distorsionării, la fel.

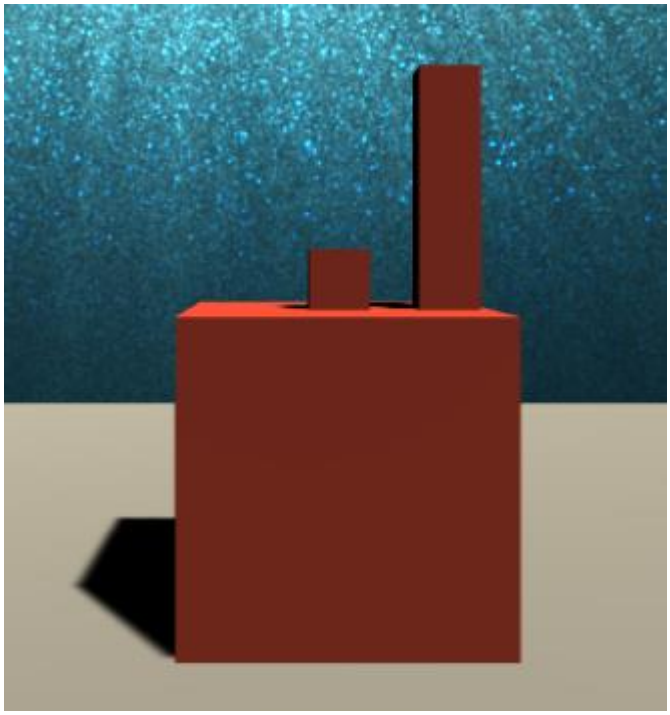


Figure 27 Imagine fără filtru subacvatic dat de shader

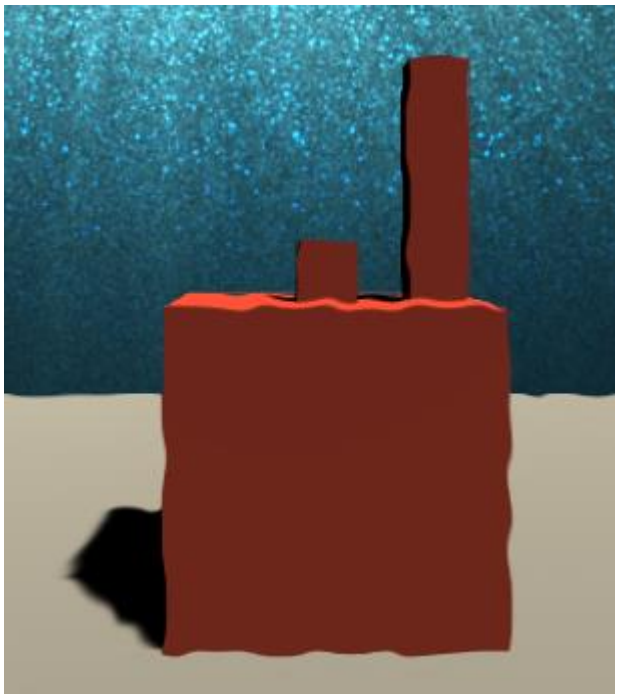


Figure 28 Imagine cu shader activ

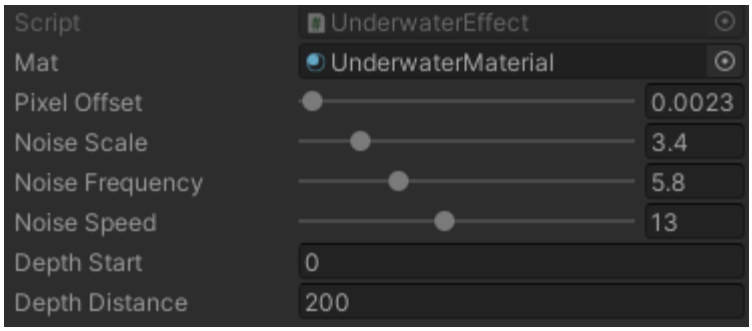


Figure 29 Parametri algoritmului

Algoritmul pentru generarea acestui comportament este urmatorul:

```
float3 spos = float3(i.scrPos.x, i.scrPos.y, 0) * _NoiseFrequency;
spos.z += _Time.x * _NoiseSpeed;
float noise = (_NoiseScale * ((snoise(spos) + 1) / 2));
```

Pentru poziția ecranului (screen position) se generează zgomot dupa frecvența _NoiseFrequency. Axa Z este animată prin adăugarea factorului de timp înmulțit cu viteza pe care o selectam prin _NoiseSpeed. Obținem zgomotul final prin scalarea zgomotului perlin generat cu factorul _NoiseScale dat din interfața materialului. Funcția *snoise* este o funcție din librăria noiseSimplex care ajută la generarea zgomotului perlin.

Următorul pas este generarea zgomotului doar pe aria selectată de noi.

```
float depthValue = Linear01Depth(tex2Dproj(_CameraDepthTexture, UNITY_PROJ_COORD(i.scrPos)).r) * _ProjectionParams.z;
depthValue = 1 - saturate((depthValue - _DepthStart) / _DepthDistance);
```

Pentru a obține adâncimea camerei ne folosim de parametri de proiecție din câmpul predefinit _ProjectionParams. Linear01Depth este o funcție predefinită de Unity care ne dă o valoare precisă între 0 și 1 a texturii date, adică, în cazul de față, o valoare pentru textura de adâncime a camerei la poziția curentă a ecranului. Scădem din 1 valoarea obținută pentru a obține un interval de la poziția de început pana la poziția de final, în apropierea camerei.

Ultimul pas al acestui shader este shiftarea pixelilor în direcțiile generate:

```
// Shift every pixel in direction generated
float4 noiseToDirection = float4(cos(noise * M_PI * 2), sin(noise * M_PI * 2), 0, 0);
fixed4 col = tex2Dproj(_MainTex, i.scrPos + (normalize(noiseToDirection) * _PixelOffset * depthValue));
return col;
```

Pentru a obține o scenă realistă folosim valori sinusoidale (sin si cos). Pe textura principală dată de _MainTex proiectăm zgomotul generat în funcție de offsetul ales prin _PixelOffset și valoarea depthValue calculată înainte.

5.2.3. Shaderul de ceață

Acest shader adaugă un efect de ceață camerei. Acest efect face ca lucrurile aflate la îndepărtare să fie ascunse în fundal, iar lucrurile aflate în apropiere să se vadă clar. Implementarea acestui shader este similară cu implementarea shaderului de distorsionare a imaginii, care dă senzația de tărâm subacvatic.

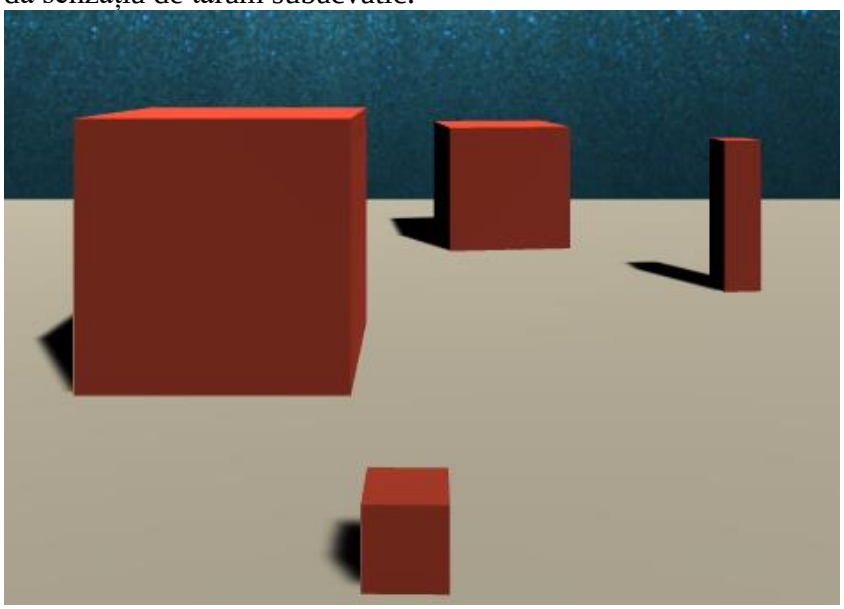


Figure 30 Scena fără ceață

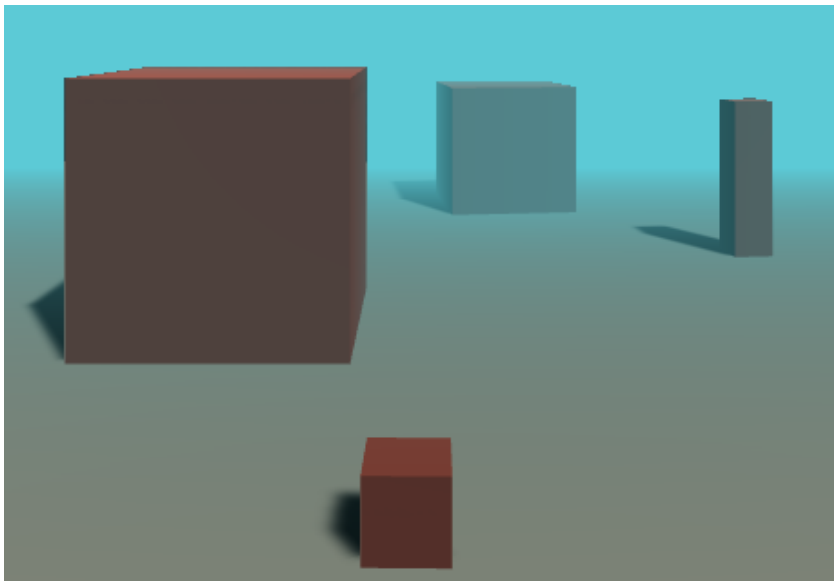


Figure 31 Scena cu ceață

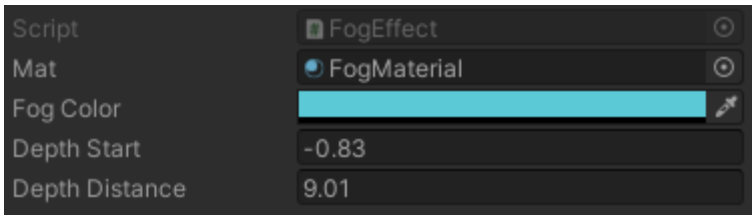


Figure 32 Interfața scriptului

La fel ca și la algoritmul anterior, trebuie să obținem adâncimea camerei și să selectăm o parte din ea pe care să aplicăm shaderul.

```
float depthValue = Linear01Depth(tex2Dproj(_CameraDepthTexture, UNITY_PROJ_COORD(i.scrPos)).r) * _ProjectionParams.z;
depthValue = saturate((depthValue - _DepthStart) / _DepthDistance);
```

De această dată, însă, nu mai scădem 1 din depthValue. Acum nu mai vrem, ca la distorsie, ca efectul să aibă loc în apropierea camerei. În schimb, vrem ca efectul să aibă loc în depărtarea camerei.

```
fixed4 fogColor = _FogColor * depthValue;
fixed4 col = tex2Dproj(_MainTex, i.scrPos);
return lerp(col, fogColor, depthValue);
```

Dupa obținerea adâncimii preluăm culoarea ceții prin variabila _FogColor. Înmulțim culoarea cu adâncimea. Textura principală este dată de _MainTex și poziția ecranului (scrPos). Pentru a obține efectul final, interpolăm textura inițiala cu ceața, având ca interval de interpolare valoarea depthValue calculată. Interpolarea se face cu ajutorul funcției predefinite de către Unity, lerp, a cărei nume provine de la prescurtarea *l(inear int)erp(olation)*. Funcția face exact ce îi spune numele, adică o interpolare liniară.

5.2.4. Shaderul pentru terrain (suprafața generată)

Pentru acest shader am încercat două implementări. În una dintre implementări, am folosit un script ce generează un gradient de culoare. Acest gradient este aplicat în funcție de înălțimea suprafeței, astfel simulând mai bine dispersia apei la adâncimi mai mari (acolo fiind mai întunecat). Gradientul acesta ar putea fi folosit în mai multe situații, de exemplu, pentru generarea pământului de diferite altitudini (cu zăpadă, apă, etc).

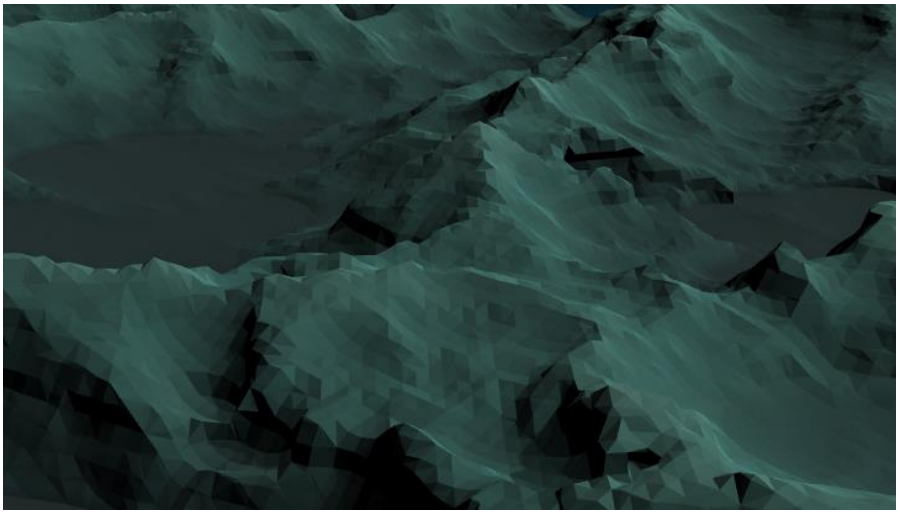


Figure 33 Efectul shaderului care aplică un gradient albastru

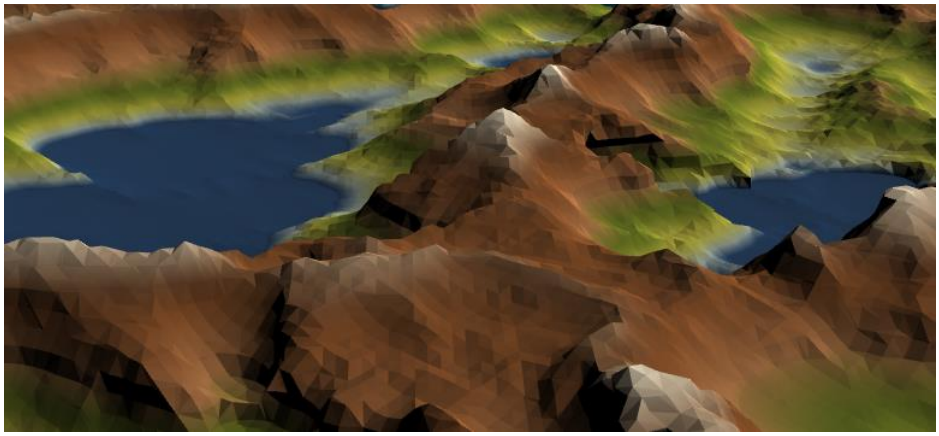


Figure 34 Același efect cu un gradient diferit

Problema cu această implementare este că nu include și posibilitatea adăugării unei texturi.

Funcționarea shaderului este dată de următorul algoritm.

```
float h = smoothstep(-boundsY/2, boundsY/2, IN.worldPos.y + IN.worldNormal.y * normalOffsetWeight);
float3 tex = tex2D(gradientramp, float2(h,.5));
o.Albedo = tex;
```

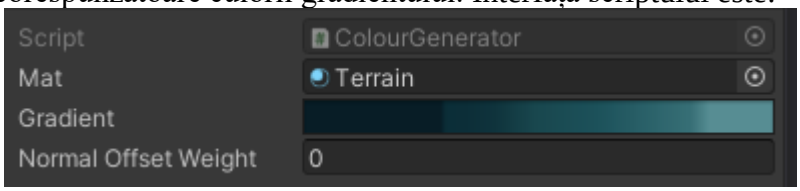
Funcția smoothstep realizează un gradient(o interpolare între primul și cel de-al doilea parametru) de-a lungul verticalei unei părți de teren. BoundsY, normalOffsetWeight sunt trimise de la algoritmul ce generează gradientul și cea din urmă semnalează cum să se facă interpolarea.

După generarea elementului de interpolare pe înălțime, se generează textura folosind gradientramp. Gradientramp este o textură gradient generată într-un script de generare a culorilor. Părțile importante din script sunt următoarele:

```
void RefreshTexture () {
    if (gradient != null) {
        Color[] colours = new Color[tex.width];
        for (int i = 0; i < resolution; i++) {
            Color gradientCol = gradient.Evaluate (i / (resolution - 1f));
            colours[i] = gradientCol;
        }

        tex.SetPixels (colours);
        tex.Apply ();
    }
}
```

Această funcție parcurge variabila gradient setată de noi în interfață și setează culoarea texturii corespunzătoare culorii gradientului. Interfața scriptului este:



În această funcție se generează și boundsY.

```
void Update () {
    Initialize ();
    RefreshTexture ();

    MeshGenerator m = FindObjectOfType<MeshGenerator> ();

    float boundsY = m.boundsSize * m.numChunks.y;

    mat.SetFloat ("boundsY", boundsY);
    mat.SetFloat ("normalOffsetWeight", normalOffsetWeight);

    mat.SetTexture ("gradientramp", tex);
}
```

Practic, boundsY seteaza înălțimea pe care se întinde gradientul. Dacă am avea mai multe *chunk-uri* verticale, trecerea de la o cluloare la alta ar fi mai lină.

A doua implementare constă în maparea unei texturi pe fiecare triunghi care alcătuiește suprafața generată. Această soluție nu va apărea în varianta finală deoarece rezultatul nu este destul de bun. Pentru un rezultat bun pe o zonă generată automat, ar trebui să implementez o proiectare triplanară ca textura să arate bine.

În această variantă se aplică o culoare, două texturi și o animație de caustice (lumini care se mișcă datorită refracției date de mișcarea apei de la suprafață).

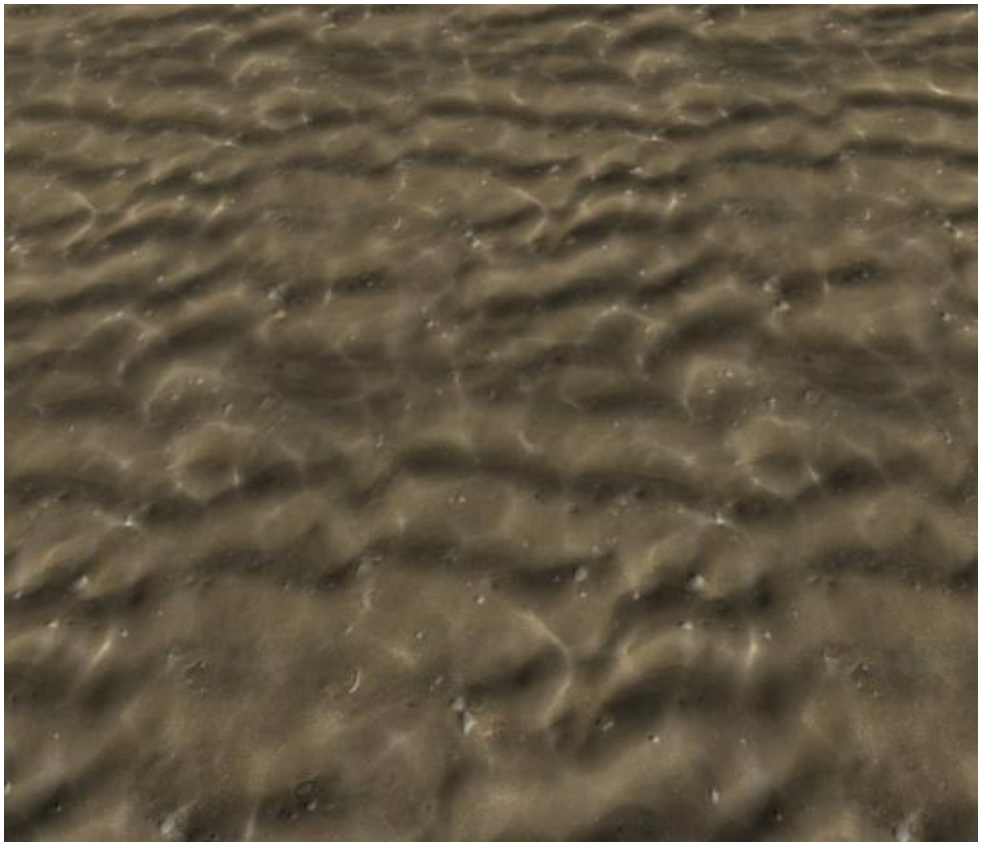


Figure 35 Textură aplicată cu animație de caustice

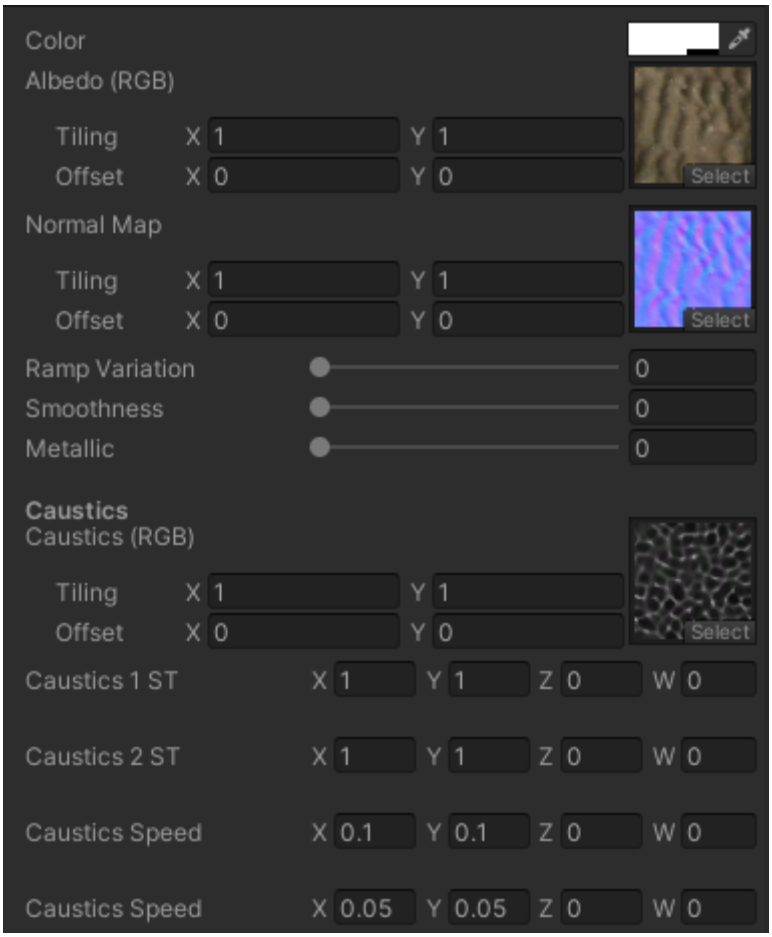


Figure 36 Interfața shaderului

Implementarea nu se schimbă cu mult. În loc să generăm o textură în funcție de gradient, aplicam doar textura a cărei valoare o înmulțim cu culoarea selectată prin parametrul _Color.

```
float3 tex = tex2D(_MainTex, IN.uv_MainTex) * _Color;
o.Albedo = tex.rgb;
```

După aplicarea texturii, trebuie să mapăm și o textură pentru normale pe care am dat-o ca parametru.

```
o.Normal = UnpackNormal(tex2D(_NormalMap, IN.uv_MainTex));
```

În momentul generării nu am ținut cont de generarea uv-urilor. UV-urile sunt informații pe care le primește textura care îi spun cum sa mapeze imaginea primita pe un interval între 0 și 1. Partea cea mai grea a fost găsirea unei reguli pentru generarea acestui UV.

```
var uv = new Vector2[vertices.Length];

for (int i = 0; i < numTris; i++)
{
    for (int j = 0; j < 3; j++)
    {
        meshTriangles[i * 3 + j] = i * 3 + j;
        vertices[i * 3 + j] = tris[i][j];

    }

    if(tris[i][1].z > tris[i][2].z)
    {
        uv[i * 3] = new Vector2(1, 0);
        uv[i * 3 + 1] = new Vector2(0, 1);
        uv[i * 3 + 2] = new Vector2(1, 1);
    } else
    {
        uv[i * 3] = new Vector2(1, 0);
        uv[i * 3 + 1] = new Vector2(0, 0);
        uv[i * 3 + 2] = new Vector2(0, 1);
    }


}
mesh.vertices = vertices;
mesh.triangles = meshTriangles;
mesh.uv = uv;
```

Soluția mea a fost maparea în funcție de orientarea triunghiurilor. Daca triunghiul avea vârful din mijloc în stanga vârfului următor mapam o anumită jumătate a texturii. În caz contrar, mapam restul texturii.

Ultima parte din acest shader este aplicarea causticelor. Această parte a fost inspirată din acest tutorial: https://www.alanzucconi.com/2019/09/13/believable-caustics-reflections/

Pentru a da impresia convingătoare de caustici se folosește o textură care este placabilă, adică o textură care poate fi repetată în toate direcțiile. Această textură va fi aplicată de doua ori pe suprafață, de fiecare dată folosind diferite viteze și dimensiuni. Aceste doua animații se vor amesteca folosind funcția *min* și, în final, se va face o dispersie a canalelor RGB pentru a simula dispersia luminii.

```
fixed2 uv = IN.uv_MainTex * _Caustics1_ST.xy + _Caustics1_ST.zw;
uv += _Caustics1_Speed * _Time.y;

fixed2 uv2 = IN.uv_MainTex * _Caustics2_ST.xy + _Caustics2_ST.zw;
uv2 += _Caustics2_Speed * _Time.y;

fixed s = _SplitRGB;
fixed r = tex2D(_CausticsTex, uv + fixed2(+s, +s)).r;
fixed g = tex2D(_CausticsTex, uv + fixed2(+s, -s)).g;
fixed b = tex2D(_CausticsTex, uv + fixed2(-s, -s)).b;

fixed3 caustics1 = fixed3(r, g, b);

r = tex2D(_CausticsTex, uv2 + fixed2(+s, +s)).r;
g = tex2D(_CausticsTex, uv2 + fixed2(+s, -s)).g;
b = tex2D(_CausticsTex, uv2 + fixed2(-s, -s)).b;

fixed3 caustics2 = fixed3(r, g, b);

// Add
o.Albedo.rgb += min(caustics1, caustics2);
```

Figure 37 Adăugarea causticelor

În codul de mai sus, _Caustics1_ST și _Caustics2_ST sunt cele două texturi. După cum am precizat și înainte, uv este folosit pentru a ști cum să mapăm textura. Astfel, luăm uv-ul inițial (IN.uv_MainTex) și matricile alcătuite din axele xy/zw pentru fiecare textură de caustice. Pentru animație adăugăm în ecuație și timpul. Avem o variabilă de direcție care, în funcție de valoare va îndepărta culorile pe cele două axe (+, + pentru roșu, +, - pentru verde și -, - pentru albastru). În final, se combină cele trei canale de culoare și se aplică funcția de *min* între cele două texturi.

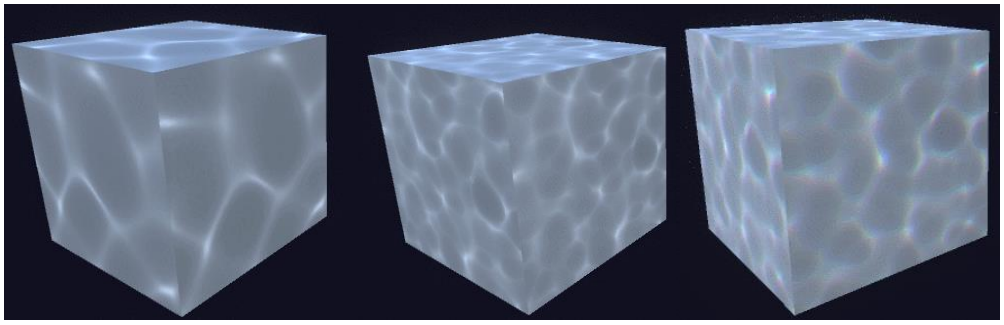


Figure 38 Rezultate fără textură dublă, cu textură dublă, cu canale rgb modificate

5.3. Implementarea Algoritmilor

5.3.1. Comportamentul Boizilor

Pentru început, un boid este un obiect al cărui copil este un model 3D pentru un pește, de exemplu. La inițializare se va atașa modelul și poziția modelului copil.

```
0 references
void Awake () {
    material = transform.GetComponentInChildren<MeshRenderer> ().material;
    cachedTransform = transform;
}
```

Algoritmul funcționează la nivel de grup de pești, dar fiecare boid în parte va trebui să implementeze următoarele funcții:

```
public void Initialize (BoidSettings settings, Transform target) {
    this.target = target;
    this.settings = settings;

    position = cachedTransform.position;
    forward = cachedTransform.forward;

    float startSpeed = settings.minSpeed;
    velocity = forward * startSpeed;
}
```

Figure 39 Inițializarea boidului cu datele primite

Pe lângă faptul că boidul primește o poziție random de la un spawner, primește ținta către care acesta va trebui să avanseze (*target*) și setări de la un fișier de setări care este separat pentru a reduce aglomerarea codului. Setările sunt atașate unui așa numit *ScriptableObject* care poate fi echivalat cu un fișier header din limbajul C.

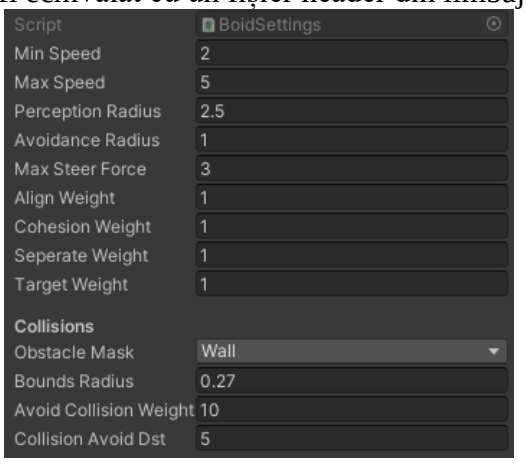


Figure 40 Interfața setărilor

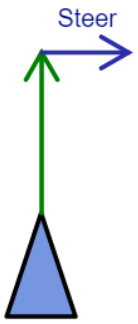
Aceste setări pot varia de la viteză, la proprietăți ale algoritmului (aliniere, coeziune, separare) și chiar și coliziuni care să facă boizii să evite anumite obstacole selectate.

Un boid trebuie să poată urma *target*-ul bancului de pești. Trebuie să generăm un vector accelerație către direcția țintei.

```
Vector3 FollowTarget()
{
    if (target != null)
    {
        Vector3 offset = target.position - position;
        return SteerTowards(offset) * settings.targetWeight;
    }
    return Vector3.zero;
}

5 references
Vector3 SteerTowards (Vector3 vector) {
    Vector3 v = vector.normalized * settings.maxSpeed - velocity;
    return Vector3.ClampMagnitude (v, settings.maxSteerForce);
}
```

Obținem vectorul dat de diferența dintre poziția țintei și poziția curentă, apoi schimbăm direcția către acel vector în funcție de *targetWeight*. Schimbarea direcției (*SteerTowards*) se face cu un vector în funcție de viteză. Cu cât velocitatea boidului e mai mare, cu atât se va întoarce mai puțin boidul. Ne asigurăm, de asemenea, că nu depășim *maxSteerForce*.



O dată ce boidul este orientat spre punctul țintă, trebuie să țină cont și de vecinii acestuia. Dacă avem vecini, putem genera un centru de coeziune, și putem să adunăm vectorului nostru de accelerație ce ne dă direcția și valorile vectorilor de aliniere, coeziune și separare.

```
Vector3 AdaptToNeighbours(Vector3 acc)
{
    if (neighbourCount != 0)
    {
        centreOfCohesion /= neighbourCount;

        Vector3 offsetToCentre = (centreOfCohesion - position);
        Vector3 alignmentForce = SteerTowards(alignment) * settings.alignWeight;
        Vector3 cohesionForce = SteerTowards(offsetToCentre) * settings.cohesionWeight;
        Vector3 seperationForce = SteerTowards(separation) * settings.seperateWeight;

        acc += alignmentForce;
        acc += cohesionForce;
        acc += seperationForce;
    }

    return acc;
}
```

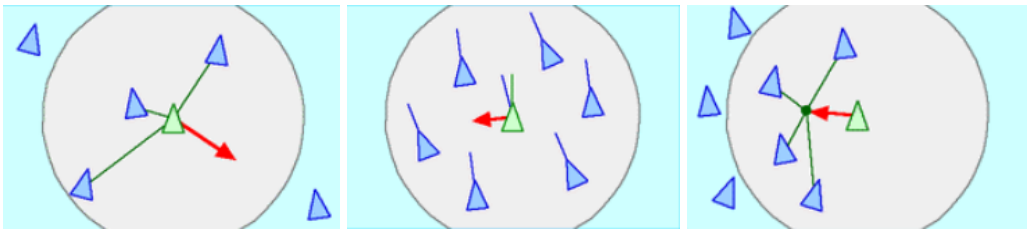


Figure 41 Separare, aliniere și coeziune

După realizarea acestor pași trebuie să ne asigurăm că boidul va evita o viitoare coliziune. Verificăm dacă urmează să se lovească de ceva, iar în acel caz se modifică din nou vectorul de accelerație.

```
Vector3 AvoidCollisions(Vector3 acc)
{
    if (WillCollide ()) {
        Vector3 collisionAvoidDir = ObstacleRays ();
        Vector3 collisionAvoidForce = SteerTowards (collisionAvoidDir) * settings.avoidCollisionWeight;
        acc += collisionAvoidForce;
    }
    return acc;
}

1 reference
bool WillCollide()
{
    if (Physics.SphereCast(position, settings.boundsRadius, forward, out _, settings.collisionAvoidDst, settings.obstacleMask))
    {
        return true;
    }
    return false;
}
```

WillCollide verifică printr-o rază înainte, în funcție de parametri dați, dacă boidul se va lovi de ceva. În acest caz, se obțin razele din jurul boidului care nu ating obstacolul. Se schimbă orientarea boidului în altă direcție față de obstacol.

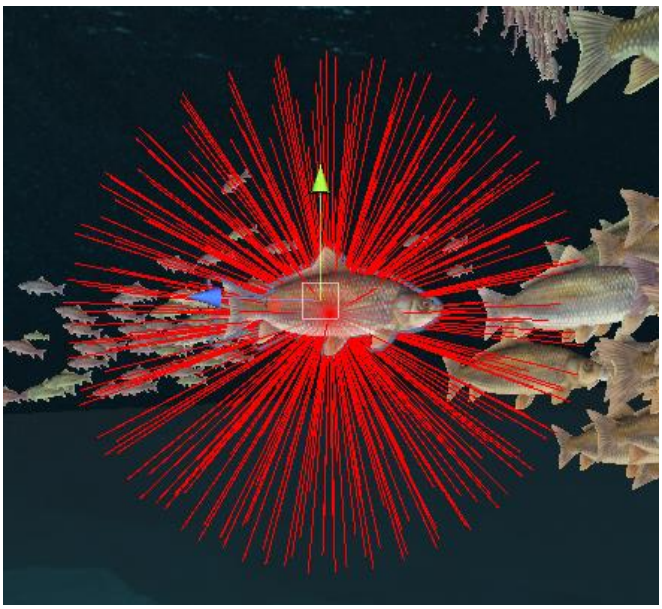


Figure 42 Razele prin care verifică un boid coliziunile

```
Vector3 ObstacleRays () {
    Vector3[] rays = BoidHelper.directions;

    for (int i = 0; i < rays.Length; i++) {
        Vector3 dir = cachedTransform.TransformDirection (rays[i]);
        Ray ray = new Ray (position, dir);

        if (!Physics.SphereCast (ray, settings.boundsRadius, settings.collisionAvoidDst, settings.obstacleMask)) {
            return dir;
        }
    }

    return forward;
}
```

După cum am precizat la descrierea generală a algoritmului, ca să generăm aceste raze în jurul boidului, va trebui să avem o distribuție bună a punctelor pe o sferă. Aceste raze se vor genera în scriptul *BoidHelper*. Pentru fiecare rază se verifică dacă se poate trage fără coliziuni și se salvează când se găsește una. Boidul se va orienta către acea rază.

```
public static class BoidHelper {

    const int directionCount = 300;
    public static readonly Vector3[] directions;

    0 references
    static BoidHelper () {
        directions = new Vector3[directionCount];

        float goldenRatio = (1 + Mathf.Sqrt(5)) / 2;
        float angleIncrement = Mathf.PI * 2 * goldenRatio;

        for (int i = 0; i < directionCount; i++) {

            // Get current fragment
            float t =  i / (float) directionCount;
            float inclination = Mathf.Acos (1 - 2 * t);

            // Distance on sphere relative to first point.
            float angularDistance = angleIncrement * i;

            float x = Mathf.Sin (inclination) * Mathf.Cos (angularDistance);
            float y = Mathf.Sin (inclination) * Mathf.Sin (angularDistance);
            float z = Mathf.Cos (inclination);
            directions[i] = new Vector3 (x, y, z);
        }
    }
}
```

BoidHelper se folosește de proprietățile rației de aur pentru a genera pozițiile x, y, z ale punctelor de pe sferă.

```
velocity += acceleration * Time.deltaTime;

Vector3 dir = velocity / velocity.magnitude;
float speed = Mathf.Clamp (velocity.magnitude, settings.minSpeed, settings.maxSpeed);

velocity = dir * speed;
```

Acum că am obținut accelerația, putem să schimbăm vectorul de velocitate. După schimbare, trebuie totuși să ne asigurăm că ne aflăm în intervalul *minSpeed-maxSpeed*. Reducem, deci, modulul vectorului la acel interval prin funcția *Clamp*, obținem vectorul unitate de direcție și reconstruim *velocity*.

```
cachedTransform.position += velocity * Time.deltaTime;
cachedTransform.forward = dir;

position = cachedTransform.position;
forward = dir;
```

Putem, în sfârșit, schimba poziția boidului în mod corect. Această funcționare trebuie să fie aplicată fiecărui boid în parte. Pentru asta am ales să creez un script manager de boizi. În primul rând, managerul inițializează fiecare boid cu setările date de noi. Mai apoi, se atribuie poziția și orientarea fiecărui boid.

```
for (int i = 0; i < boidCount; i++) {
    boidData[i].direction = boids[i].forward;
    boidData[i].position = boids[i].position;
}
```

Pentru procesarea rapidă a boizilor folosim un *ComputeShader* pentru a putea lucra în paralel.

```
var boidBuffer = new ComputeBuffer (boidCount, BoidData.Size);
boidBuffer.SetData (boidData);

compute.SetBuffer (0, "boids", boidBuffer);
compute.SetInt ("boidCount", boidCount);
compute.SetFloat ("viewRadius", settings.perceptionRadius);
compute.SetFloat ("avoidRadius", settings.avoidanceRadius);

// Count of thread Groups (total boids / size of a group)
int threadGroups = Mathf.CeilToInt (boidCount / (float) threadGroupSize);
compute.Dispatch (0, threadGroups, 1, 1);
```

Se trimite mai departe o structură de date pentru fiecare boid în parte, conținând informațiile acestuia. Pe deasupra, se trimite numărul de boizi, setări și se realizează calculul pe grupurile de threaduri.

```
for (int i = 0; i < boidCount; i++) {
    boids[i].alignment = boidData[i].alignment;
    boids[i].centreOfCohesion = boidData[i].centreOfCohesion;
    boids[i].separation = boidData[i].separation;
    boids[i].neighbourCount = boidData[i].boidCount;

    boids[i].UpdateBoid ();
}
```

Rezultatele se obțin și se atașează la boizi.

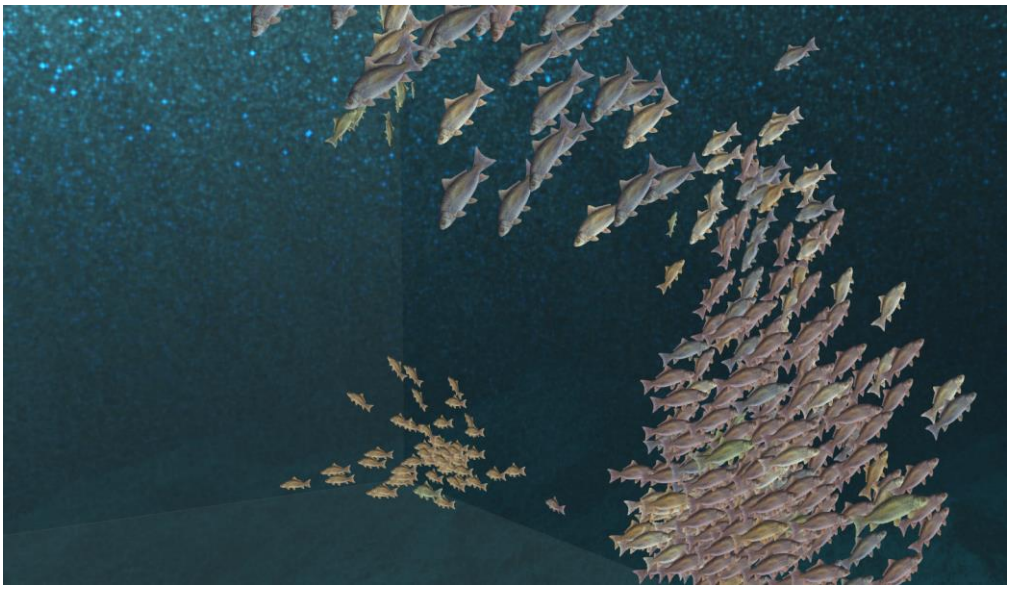


Figure 43 Rezultat final

5.3.2. Generarea terenului

Generarea terenului se face cu ajutorul algoritmului *marching-cubes*. Pentru a aplica acest algoritm eficient am împărțit zonele în *chunk*-uri. Astfel, vom genera doar bucăți din teren în direcția spre care ne uităm ca să nu generăm prea multe, inutil.

Figure 44 Chunk example

O mare parte din codul destinat generării pământului nu este algoritmul propriu zis. Cea mai mare parte din cod ajută la selectarea chunk-urilor care vor fi generate. Aplicația rulează în mod simplu la nivel înalt și prezintă două situații: o situație în care chunk-urile sunt fixate, nu vor mai apărea altele, nu vor mai dispărea din cele existente. O să folosesc această versiune pentru scena în care se va modifica automat terenul atunci când sunt modificați parametri. Așa se asigură o rulare fără pierderea cadrelor pe secundă, modificările aplicându-se la un număr limitat de chunk-uri. Cealaltă versiune este folosită pentru generarea de chunk-uri pe măsură ce jucătorul se va mișca în lume. Variabila *fixedMapSize* ne spune dacă suntem în primul sau cel de-al doilea caz.

Abia după aceste părți vom putea merge mai departe la generarea zgomotului și construirea acestuia folosind marching cubes.

```
public void Run ()
{
    CreateBuffers();

    if(!fixedMapSize && Application.isPlaying)
    {
        InitUnfixedChunks();
    }
    else
    {
        InitFixedChunks();
    }

    if (!Application.isPlaying)
    {
        ReleaseBuffers();
    }
}
```

Figure 45 Simpla rulare a aplicației

Cum vrem ca algoritmul să fie eficient, generăm bufferele necesare pentru a putea fi trimise la *ComputeShader* mai departe. Pentru bucata curentă de pământ, trebuie să generăm un număr de cuburi pe care să aplicăm algoritmul pentru generare.

```
// Buffer precision (basically how many voxels will be in a cube + nr of triangles)
int cubeCountPerAxis = pointCountPerAxis - 1;
int cubeCount = cubeCountPerAxis * cubeCountPerAxis * cubeCountPerAxis;

int pointCount = pointCountPerAxis * pointCountPerAxis * pointCountPerAxis;

// There can be maximum 5 triangles per cube.
int maxTriangleCount = cubeCount * 5;
```

```
// public ComputeBuffer(int count, int stride, ComputeBufferType type);
// A buffer of maxTriangleCount triangles of sizeof(int) * 3 * 3 size that can be appended
//  - Size is made by 3 points represented in 3D space by 3 int coordinates
triangleBuffer = new ComputeBuffer(maxTriangleCount, sizeof(int) * 9, ComputeBufferType.Append);
// Position + density (3 coords + 1 float)
pointsBuffer = new ComputeBuffer(pointCount, sizeof(int) * 4);
// Value that determines which triangles to be selected from the tables
// (int value which tells what edges are below the isosurface)
triCountBuffer = new ComputeBuffer(1, sizeof(int), ComputeBufferType.Raw);
```

Din *pointCountPerAxis* putem genera numărul de cuburi și să-l trimitem mai departe la ComputeShader. De asemenea trimitem și numărul maxim de triunghiuri. Pot fi maxim 5 triunghiuri într-un cub, deci numărul de triunghiuri va fi de 5 ori mai mare decât numărul de cuburi, în cel mai nefericit caz.

Generarea se va face în funcție de poziția camerei de vizualizare. Pentru asta trebuie să salvăm chunk-urile din față.

```
Vector3 viewerPos = viewer.position;
Vector3 positionInChunk = viewerPos / boundsSize;
Vector3Int viewerCoord = new Vector3Int(Mathf.RoundToInt(positionInChunk.x),
                                        Mathf.RoundToInt(positionInChunk.y),
                                        Mathf.RoundToInt(positionInChunk.z));

int maxChunksInView = Mathf.CeilToInt(viewDistance / boundsSize);
```

Pentru situația în care numărul de chunk-uri este fixat, tot timpul vom avea o lista cu chunk-urile anterioare. Ca intrare vom da numărul de chunk-uri pe cele trei axe. Parcurgem fiecare coordonată, iar dacă deja exista chunk-ul. Dacă există, îl folosim pe acela, dacă nu, adăugăm în lista de chunk-uri unul nou.

Pe fiecare chunk trimitem mai departe parametri către compute shader care vor fi procesați de algoritm.

```
for (int x = 0; x < numChunks.x; x++)
{
    for (int y = 0; y < numChunks.y; y++)
    {
        for (int z = 0; z < numChunks.z; z++)
        {
            Vector3Int coord = new Vector3Int(x, y, z);
            bool chunkExists = false;

            for (int i = 0; i < oldChunks.Count && !chunkExists; i++)
            {
                if (oldChunks[i].coord == coord)
                {
                    chunks.Add(oldChunks[i]);
                    oldChunks.RemoveAt(i);
                    chunkExists = true;
                }
            }

            if (!chunkExists)
            {
                var newChunk = CreateChunk(coord);
                chunks.Add(newChunk);
            }

            chunks[count].SetUp(mat, generateColliders);
            UpdateChunkMesh(chunks[count]);
            count++;
```

Figure 46 Cazul în care avem număr static de chunk-uri

În cealaltă situație, pentru a evicientiza procesul, se implementează un proces de reciclare. Practic, vom seta o rază la o distanță dată. Orice depășește lungimea acesteia va fi trimis spre reciclare. Chunk-urile trimise spre reciclare vor fi refolosite pentru desenarea celor relevante din interiorul razei. Evităm, deci, generarea prea multor chunk-uri și supraîncărcarea memoriei.

```
int maxChunksInView = Mathf.CeilToInt(viewDistance / boundsSize);
float diagonalViewDistance = viewDistance * viewDistance;

Vector3 cameraPos = camera.position;
Vector3 positionInChunk = cameraPos / boundsSize;
Vector3Int cameraIntPos = new Vector3Int(Mathf.RoundToInt(positionInChunk.x),
                                         Mathf.RoundToInt(positionInChunk.y),
                                         Mathf.RoundToInt(positionInChunk.z));

RecycleOutOfDistance(diagonalViewDistance);
```

Figure 47 Reciclarea chunk-urilor aflate înafara ariei vizuale

```
if (IsVisibleFrom(bounds, Camera.main))
{
    Chunk chunk = null;
    if (recycleableChunks.Count > 0)
    {
        chunk = recycleableChunks.Dequeue();
    }
    else
    {
        chunk = CreateChunk(coord);
        chunk.SetUp(mat, generateColliders);
    }

    chunk.coord = coord;
    existingChunks.Add(coord, chunk);
    chunks.Add(chunk);
    UpdateChunkMesh(chunk);
}
```

Figure 48 Folosirea chunk-urilor reciclabile în cazul în care există

După ce decidem dacă chunk-urile o să fie reciclate sau create, aplicăm, la fel ca anterior, algoritmul pe *ComputeShader*. Pentru aplicarea algoritmului trebuie să trimitem mai mulți parametri mai departe pe care îi vom aplica în următoarea formulă:

```
-(pos.y + floorOffset) + noise * noiseWeight + (pos.y % params.x) * params.y;
```

Formula poate fi împărțită în trei părți:

- poziția terenului în chunk

```
-(pos.y + floorOffset) + noise * noiseWeight + (pos.y % params.x) * params.y;
```

pos.y este poziția verticală de la care începe generarea terenului care se află la mijlocul înălțimii chunk-ului.
floorOffset este valoarea cu care modificăm acea poziție, ridicând astfel terenul în sus sau coborându-l în jos.

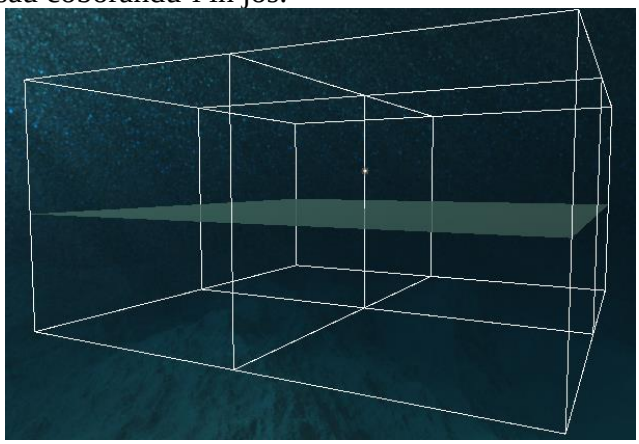


Figure 49 Poziția fără offset

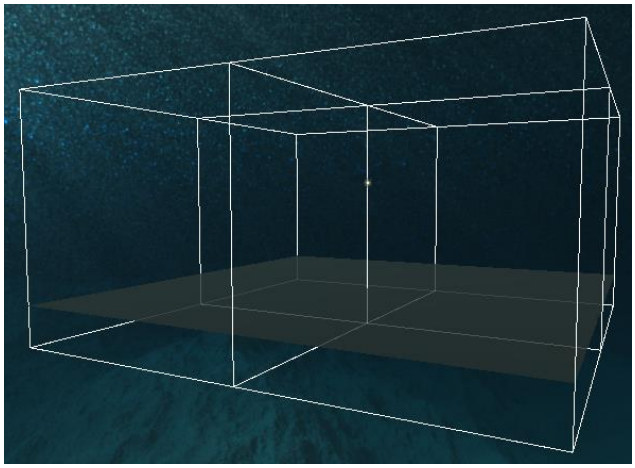


Figure 50 poziția cu offset

- zgomotul

```
-(pos.y + floorOffset) + noise * noiseWeight + (pos.y % params.x) * params.y;
```

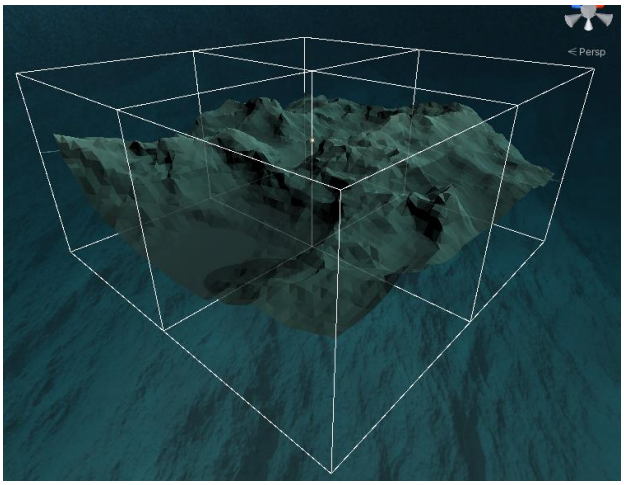


Figure 51 Zgomotul adăugat

Zgomotul este generat de mai mulți parametri.

```
for (int i = 0; i < octaves; i++) {
    float n = snoise((pos+offsetNoise) * frequency + offsets[i] + offset);
    float v = 1 - abs(n);
    v = v*v;
    v *= weight;
    weight = max(min(v*weightMultiplier,1),0);
    noise += v * amplitude;
    amplitude *= persistence;
    frequency *= lacunarity;
}
```

n reprezintă un zgomot perlin generat după niște valori de offset alese întâmplător.

weight reprezintă intensitatea zgomotului, cât de proeminent este acesta. Valoarea 1 a acestei variabile reprezintă o influență totală a zgomotului, iar valoarea 0 înseamnă lipsa influenței zgomotului.

amplitude este amplitudinea zgomotului, cât de mult poate varia zgomotul în înălțime.

frequency este frecvența zgomotului și, după cum îi spune numele, arată cât de des apare acest zgomot.

octaves poate fi considerată o variabilă de limită. Aceasta spune de câte ori se vor repeta operațiile de zgomot. La un moment dat, funcția de zgomot va converge la o anumită valoare și nu se mai modifica, deci creșterea numărului de octave ar crește doar complexitatea algoritmului fără rost. De aceea este bine să alegem un număr bun pentru această variabilă. Nu vrem să fie nici prea mic, deoarece ar reduce complexitatea zgomotului, dar nici prea mare, deoarece ar adăuga calcule inutile.

persistence și *lacunarity* dau cel mai bun rezultat atunci când se crește frecvența și se reduce amplitudinea. Persistența spune rata cu care crește (sau la valoare subunitară, scade) amplitudinea zgomotului. Lacunaritatea arată același lucru, doar pentru frecvență. De aceea, datorită modului în care arată pământul real (cu atât mai detaliat cu cât mai fragmentat) trebuie să alegem o valoare supraunitară pentru lacunaritate și una subunitară pentru persistență.

snoise nu este o variabilă, dar este o funcție de zgomot. Shader-ele nu au funcții de noise, iar acestea trebuie generate din texturi sau din calcule de către noi. Am preluat o funcție de zgomot realizată de Ian McEvan.

- influențele controlate

```
-(pos.y + floorOffset) + noise * noiseWeight + (pos.y % params.x) * params.y;
```

Se observă datorită calculului simplu ce parte a generării afectează această parte. Calculul nu este obligatoriu să fie la fel ca în această formulă. Se poate folosi orice formulă în funcție de valoarea dorită. În acest caz, valoarea parametrului *x* dat este folsită pentru a da efectul de canion (textura are înălțimi constante pe valori între multiplii de *x*). Variabila *y* controlează cât de tare se răspândește acest efect.

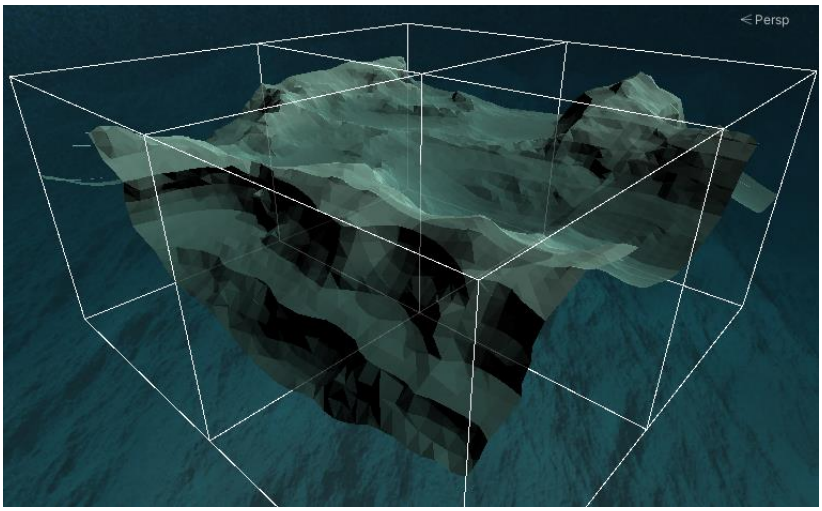


Figure 52 Efect de canion

După obținerea punctelor generate după formulă putem să le dăm mai departe în *ComputeShader*-ul ce aplică algoritmul pentru generarea triunghiurilor. În acest punct trebuie doar să aplicăm algoritmul pe fiecare cub.

```
float4 cubeCorners[8] = {
    points[indexArray(id.x, id.y, id.z)],
    points[indexArray(id.x + 1, id.y, id.z)],
    points[indexArray(id.x + 1, id.y, id.z + 1)],
    points[indexArray(id.x, id.y, id.z + 1)],
    points[indexArray(id.x, id.y + 1, id.z)],
    points[indexArray(id.x + 1, id.y + 1, id.z)],
    points[indexArray(id.x + 1, id.y + 1, id.z + 1)],
    points[indexArray(id.x, id.y + 1, id.z + 1)]
};
```

Inițial obținem colțurile cubului.

```
int cubeIndex = 0;
if (cubeCorners[0].w < isoLevel) cubeIndex |= 1;
if (cubeCorners[1].w < isoLevel) cubeIndex |= 2;
if (cubeCorners[2].w < isoLevel) cubeIndex |= 4;
if (cubeCorners[3].w < isoLevel) cubeIndex |= 8;
if (cubeCorners[4].w < isoLevel) cubeIndex |= 16;
if (cubeCorners[5].w < isoLevel) cubeIndex |= 32;
if (cubeCorners[6].w < isoLevel) cubeIndex |= 64;
if (cubeCorners[7].w < isoLevel) cubeIndex |= 128;
```

Al doilea pas este calcularea variabilei *cubeIndex* pe care am prezentat-o la descrierea algoritmului.

```
for (int i = 0; triangulation[cubeIndex][i] != -1; i +=3) {

    int a0 = cornerIndexAFromEdge[triangulation[cubeIndex][i]];
    int b0 = cornerIndexBFromEdge[triangulation[cubeIndex][i]];

    int a1 = cornerIndexAFromEdge[triangulation[cubeIndex][i+1]];
    int b1 = cornerIndexBFromEdge[triangulation[cubeIndex][i+1]];

    int a2 = cornerIndexAFromEdge[triangulation[cubeIndex][i+2]];
    int b2 = cornerIndexBFromEdge[triangulation[cubeIndex][i+2]];

    Triangle tri;
    tri.vertexA = interpolate(cubeCorners[a0], cubeCorners[b0]);
    tri.vertexB = interpolate(cubeCorners[a1], cubeCorners[b1]);
    tri.vertexC = interpolate(cubeCorners[a2], cubeCorners[b2]);
    triangles.Append(tri);
}
```

În final generăm triunghiurile în funcție de tablea de triangulare dată de dezvoltatorii algoritmului. Le adăugăm vectorului de triunghiuri.

Se poate, acum, construi mesh-ul din triunghiurile obținute.

```
Mesh mesh = chunk.mesh;
mesh.Clear();

var vertices = new Vector3[triCount * 3];
var meshTriangles = new int[triCount * 3];

for (int i = 0; i < triCount; i++)
{
    for (int j = 0; j < 3; j++)
    {
        meshTriangles[i * 3 + j] = i * 3 + j;
        vertices[i * 3 + j] = tris[i][j];
    }
}
mesh.vertices = vertices;
mesh.triangles = meshTriangles;

mesh.RecalculateNormals();
```

La mesh se adaugă vârfurile, triunghiurile și se recalculează normalele. S-ar putea genera și UV-ul pentru mesh, dar, nedorind să texturăm această suprafață, nu o să o creem în această versiune.

5.4. Alte adăugări pentru completarea funcționalității scenelor

5.4.1. Introducere

Deși tot ce mi-am propus este implementat, trebuie să mă asigur că acestea pot fi într-un tot unitar. Pentru asta am avut nevoie de utilități extra pentru a-mi ușura aplicarea algoritmilor, testarea, navigarea, etc.

5.4.2. Skybox

Pentru a oferi o tentă mai reală scenei, am adăugat un skybox. Pentru asta m-am folosit de magazinul de bunuri pentru a lua un skybox gratuit. Acesta a venit la pachet cu un cubemap și un material pe care este aplicat cubemap-ul. Aplicarea skybox-ului se face prin adăugarea acestuia în cadrul proprietăților de iluminare.

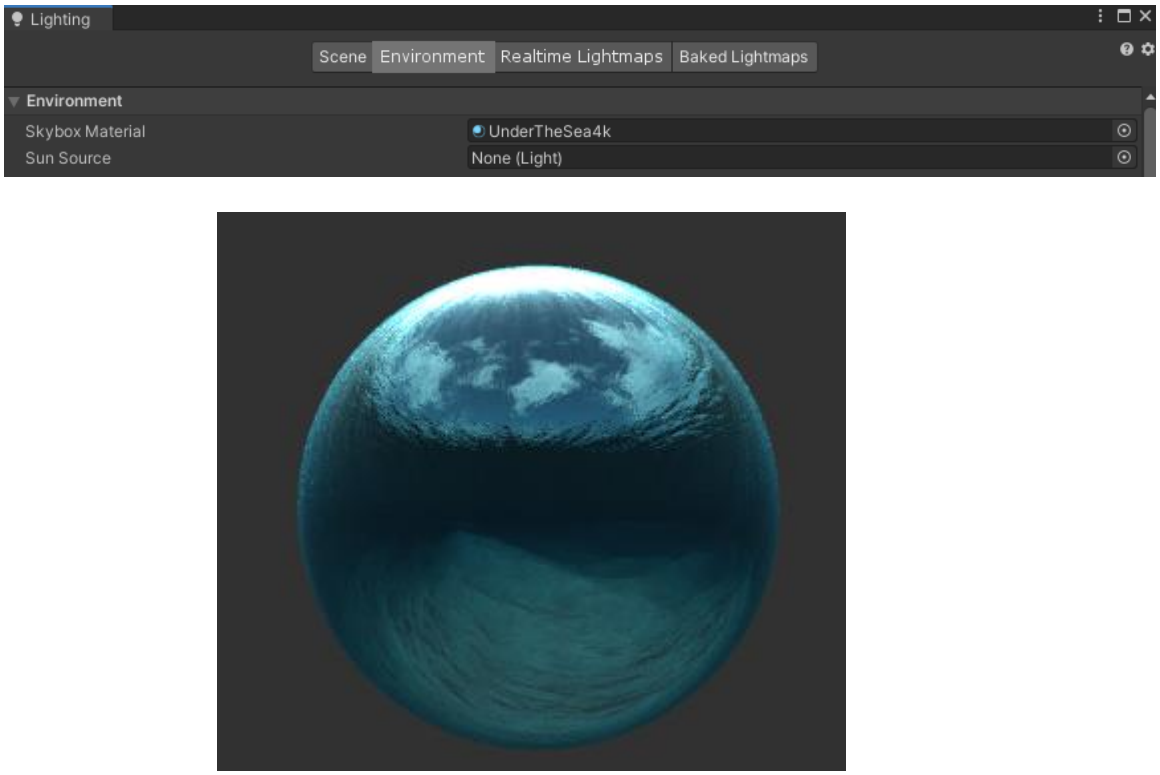


Figure Modelul pentru skybox

5.4.3. Mișcarea camerei

Cu toate că mișcarea se realizează prin intermediul VR-ului, nu a fost tot timpul așa. Pentru verificarea algoritmilor și pentru construirea și vizualizarea scenei a trebuit să îmi construiesc un algoritm pentru mișcarea camerei.

Funcționarea este dată de urmatoarele doua scripturi: unul care realizează mișcarea și celălalt care realizează rotația în cazul în care mouse-ul este apăsat.

```
void Update()
{
    if(Input.GetKey(KeyCode.W)) {
        transform.position = transform.position + Camera.main.transform.forward * speed * Time.deltaTime;
    }
    if(Input.GetKey(KeyCode.S))
    {
        transform.position = transform.position - Camera.main.transform.forward * speed * Time.deltaTime;
    }
    if (Input.GetKey(KeyCode.A))
    {
        transform.position = transform.position - Camera.main.transform.right * speed * Time.deltaTime;
    }
    if (Input.GetKey(KeyCode.D))
    {
        transform.position = transform.position + Camera.main.transform.right * speed * Time.deltaTime;
    }

}
```

Figure 53 Mișcarea pentru tastele WASD

Dacă tasta W sau S este apăsată, atunci camera va mișca în direcția vectorului *forward* pe axa pozitivă, respectiv negativă.

Dacă tasta A sau D este apăsată, atunci camera va mișca în direcția vectorului *right* pe axa pozitivă, respectiv negativă.

```
[Range(100f, 500f)]
public float mouseSensitivity = 100f;
public Transform controllerBody;

float xRotation = 0f;
// Start is called before the first frame update
0 references
void Start()
{
    Cursor.lockState = CursorLockMode.Locked;
}

// Update is called once per frame
0 references
void Update()
{
    float mouseX = 0f;
    float mouseY = 0f;
    if(Input.GetMouseButton(1))
    {
        mouseX = Input.GetAxis("Mouse X") * mouseSensitivity * Time.deltaTime;
        mouseY = Input.GetAxis("Mouse Y") * mouseSensitivity * Time.deltaTime;
    }


    xRotation -= mouseY;
    xRotation = Mathf.Clamp(xRotation, -90f, 90f);

    // Using localRotation instead of Rotate to be able to Clamp down to 180 deg.
    transform.localRotation = Quaternion.Euler(xRotation, 0f, 0f);
    controllerBody.Rotate(Vector3.up * mouseX);

}
```

Figure 54 Rotația camerei atunci când mouse-ul este apăsat

Am declarat mai multe variabile pentru controlul camerei. *mouseSensitivity* precizează viteza cu care se realizează rotația, iar *controllerBody* precizează părintele

obiectului cameră care își va schimba rotația în funcție de mouse. Execuția codului este simplă. La fiecare modificare a pozițiilor X și Y ale mouse-ului, dacă click dreapta este apăsat (*GetMouseButton(1)*) se modifică rotațiile *mouseX* și *mouseY*. Prima dintre ele este folosită pentru a roti de-a lungul axei Y, adică în stânga și dreapta. De aceea, mai la finalul codului înmulțim mouseX cu vectorul *up*. Cealaltă rotație este cea după axa X, adică în sus și în jos, dar pentru a evita comportamentul bizar al unei camere care s-ar da peste cap, a trebuit sa limitez rotația pe verticala la 90 grade în sus și 90 grade în jos.

5.4.4. Spawner-ul de boizi

Crearea manuala de boizi consumă mult timp și nu este eficientă. Pentru a crea boizi de diferite dimensiuni și culori a trebuit să îmi creez un *prefab* de spawner. Un prefab este un obiect pe care l-am salvat pentru a-l putea replica mai târziu.

Un spawner este un obiect sferic invizibil în timpul rulării aplicației. Acest spawner se ocupă de crearea mai multor pești. De asemenea, acesta dictează modul în care aceștia sunt creați.

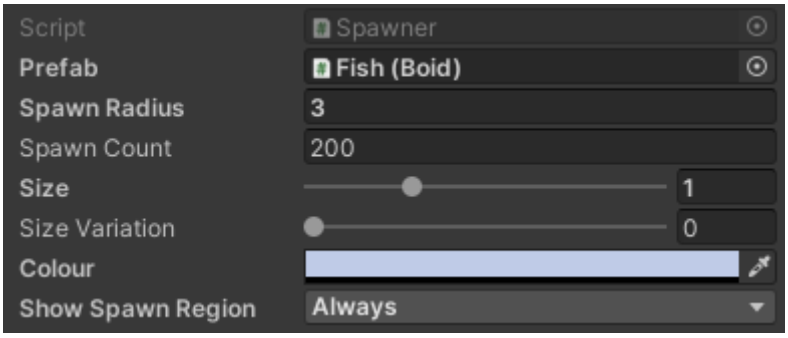


Figure 55 Parametri unui spawner

După cum se poate observa și în figură, un spawner poate decide ce tip de boid spawnează (în cazul în care am avea alte tipuri de pești, am putea spawna acele tipuri), la ce distanță de centru vor apărea peștii, câți pești vor apărea și ce dimensiune vor avea acei pești. La această dimensiune am adăugat și un element de variație. Dacă variația este 0, toți peștii vor avea aceași marime. Dacă, în schimb, variația este diferită de 0, peștii ce vor apărea de dimensiuni egale cu dimensiunea selectată și o creștere sau scădere în funcție de valoarea dată.

Culoarea are dublă utilizare. Aceasta schimbă culoarea indicatorului spawner-ului și culoarea peștilor.

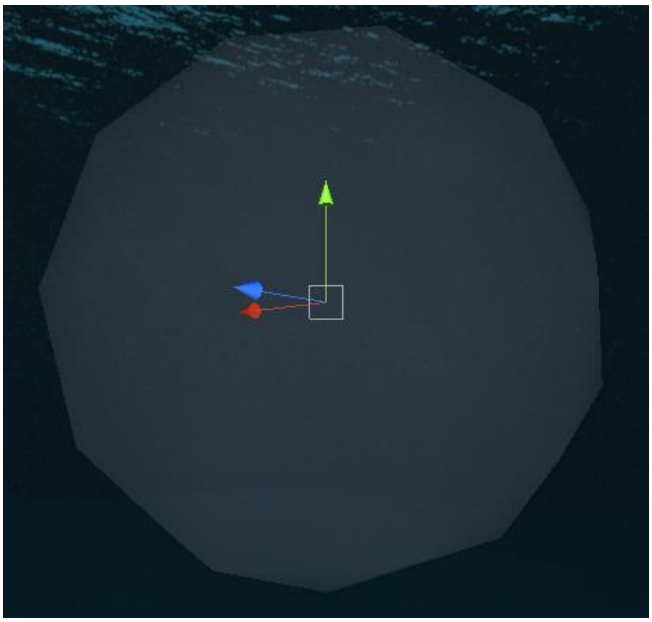


Figure 56 Un spawner în editor

```
void Awake () {
    for (int i = 0; i < spawnCount; i++) {
        Boid boid = Instantiate (prefab);

        // Position and orientation of the boid
        boid.transform.position = Random.insideUnitSphere * spawnRadius + transform.position;
        boid.transform.forward = Random.insideUnitSphere;

        float randomVal = Random.Range(size - sizeVariation, size + sizeVariation);
        boid.transform.localScale = new Vector3(randomVal, randomVal, randomVal);

        boid.SetColour (colour);
    }
}
```

Figure 57 Funcționarea spawner-ului

Pentru fiecare boid se face o instanțiere. Apoi, pentru varietate, se așează boidul într-un loc întâmplător în raza sferei. Se orientează, la fel, întâmplător în interiorul sferei și, dacă s-a dat o variație, se crește sau scade dimensiunea peștelui curent. În final se setează culoarea peștelui.

```
private void OnDrawGizmos () {
    if (showSpawnRegion == GizmoType.Always) {
        Gizmos.color = new Color(colour.r, colour.g, colour.b, 0.4f);
        Gizmos.DrawSphere(transform.position, spawnRadius);
    }
}
```

Figure 58 Desenarea indicatorului de spawner

Un gizmo este, practic, un indicator. Acesta nu va fi vizibil în timpul jocului, dar este foarte util în timpul editării sau testării. Dacă am selectat vizualizarea zonei de

spawn, generăm pentru Gizmo-ul curent culoarea dată, cu transparență și desenăm o sferă.

5.4.5. Perete de sticlă

Dacă vrem să simulăm un acvariu și dacă vrem ca boizii să nu părăsească o anumită zonă trebuie să avem delimitatori ai zonelor în care aceștia au acces. De aceea am realizat un acvariu de sticlă care realizează coliziuni cu aceștia pentru a ne asigura că nu vor trece prin sticlă peștii.

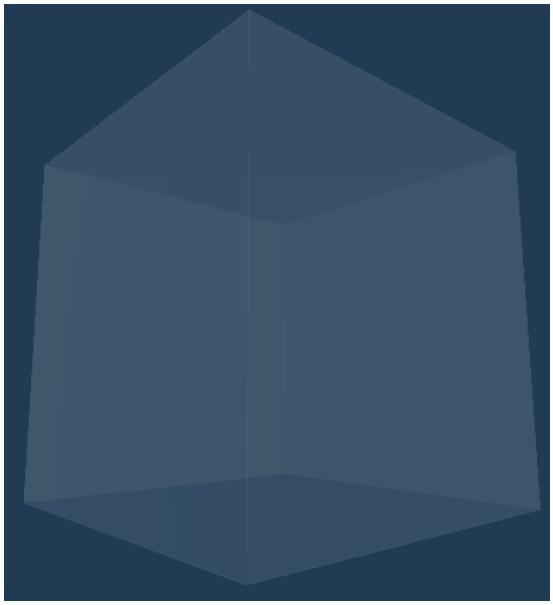


Figure 59 Perete de sticlă

Acest perete are un material standard care, în loc de efect *opaque* are un efect *fade* pentru a simula transparența sticlei.

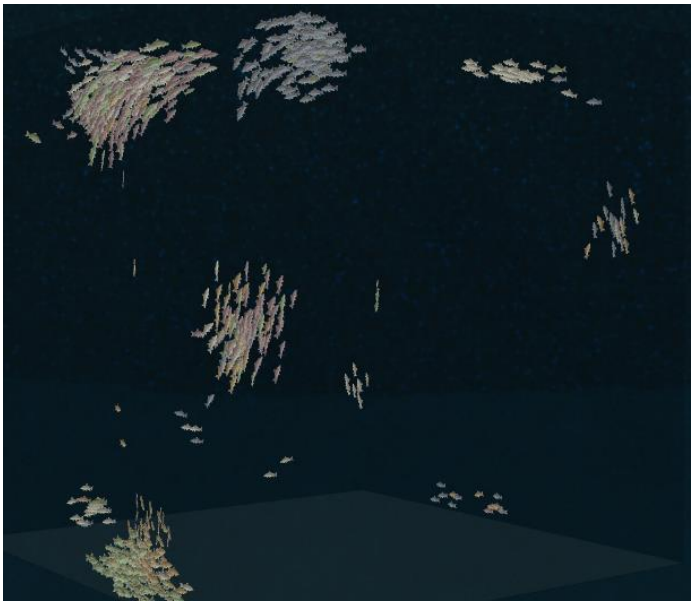


Figure 60 Exemplu de utilizare a corpului de sticlă

5.4.6. Animația mâinilor pentru VR

Dacă vrem ca acțiunile făcute de noi să corespundă cu acțiunile din joc, trebuie să sugerăm asta prin animații. Inițial am avut nevoie de un model pentru mâini. Steam oferă toate modelele care sunt folosite în SteamVR. Am preluat acel model împreună cu scheletul acestuia de animație.

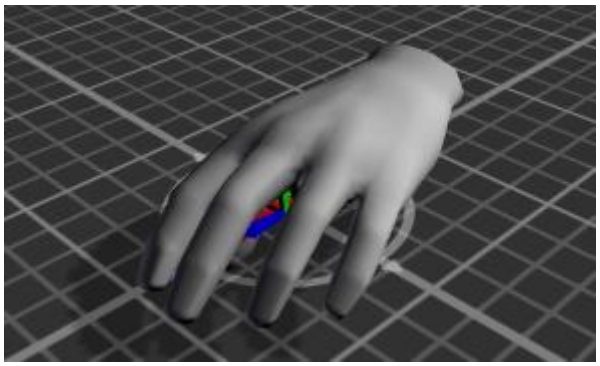


Figure 61 Modelul unei mâini

Pentru animație, Unity oferă un obiect *Animator*. Acest obiect ușurează munca animatorului și permite aplicarea animațiilor în funcție de un arbore de acțiuni.

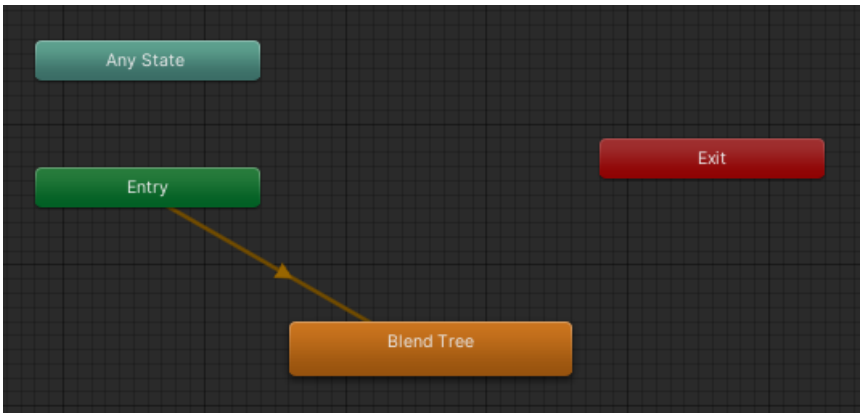


Figure 62 Arbore de acțiuni

Fiecare stare din acest arbore poate fi modificată. Animația fiind simplă în cazul meu, a dus la simpla schimbare a *Blend Tree*-ului.

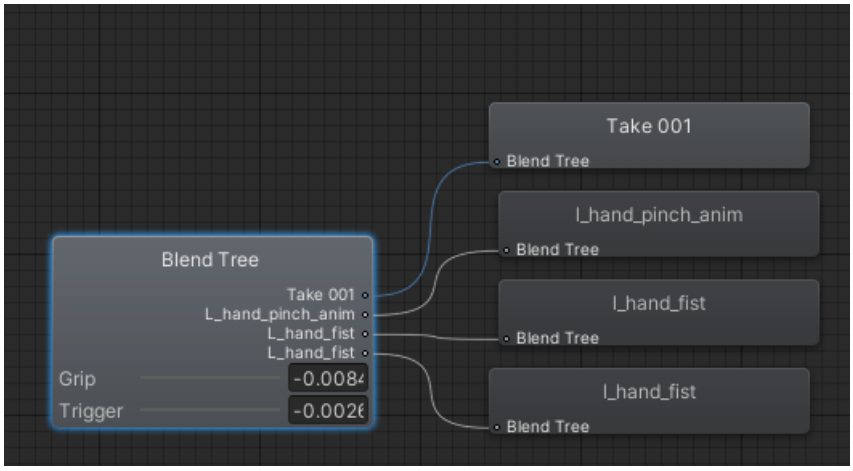


Figure 63 Animația blend tree-ului

Controlerele VR au butoane care nu au doar valori de 0 și 1. Aceste valori sunt de precizie cu virgulă și pot fi oriunde din intervalul 0-1. Din cauză că am adăugat 4 acțiuni s-au creat patru puncte de animație. Aniamtorul oferă fluiditate între acțiuni pentru ca animația să nu fie bruscă. Astfel, tot ce trebuiue să fac este să modific scheletul pentru fiecare stare. Se va realiza o mișcare liniară între cele patru stări în funcție de input-ul oferit.

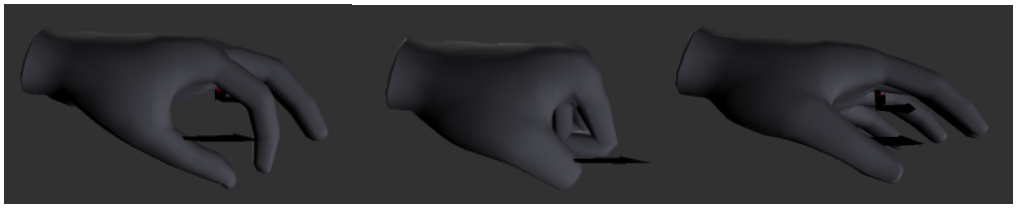


Figure 64 Pozițiile posibile ale mâinii

Momentan există doar trei poziții posibile, una dintre ele fiind utilizată de două ori ca înlocuitor pentru atunci când voi dori implementarea unei noi animații.

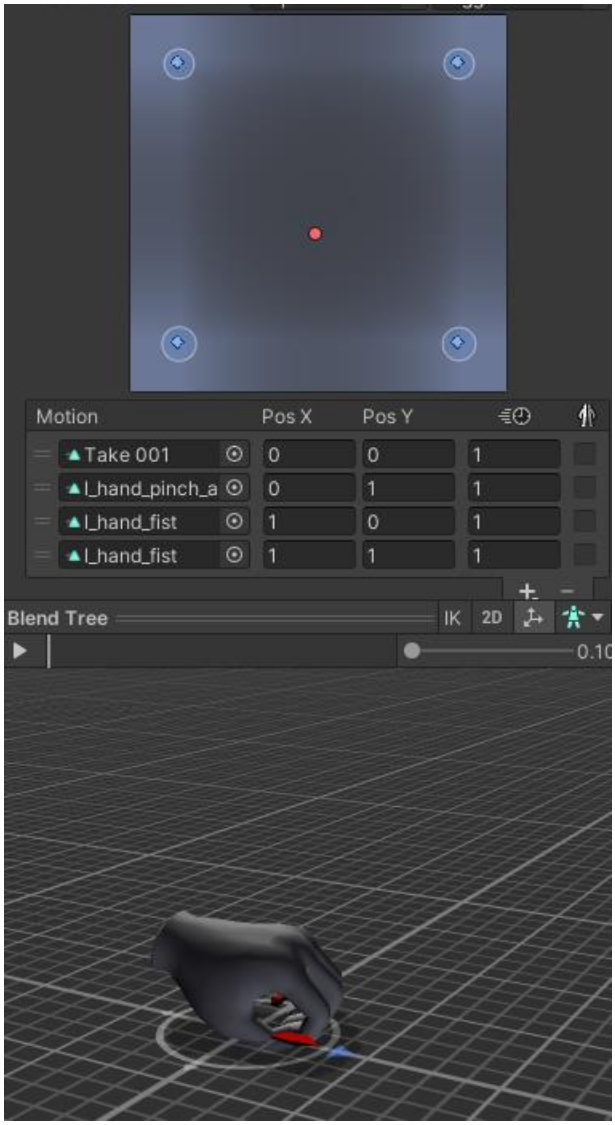


Figure 65 O animație intermediara la mijlocul suprafeței dată de cele patru puncte

Pentru preluarea input-ului oferim, în primul rând, în interfață modelul și caracteristicile controlerului. Caracteristicile vor fi *Controller* și *Right/Left* după caz.

Figure 66 Caracteristicile unei mâini

```
List<InputDevice> devices = new List<InputDevice>();
InputDevices.GetDevicesWithCharacteristics(controllerCharacteristics, devices);
```

```
if (devices.Count > 0)
{
    targetDevice = devices[0];

    spawnedHandModel = Instantiate(handModelPrefab, transform);
    handAnimator = spawnedHandModel.GetComponent<Animator>();
}
```

În cod, după preluarea acestora, luăm primul controler care se potrivește cu ele. Instanțiem modelul mâinii și îi atașăm animatorul.

```
void Update()
{
    if(!targetDevice.isValid)
    {
        TryInitialize();
    } else
    {
        spawnedHandModel.SetActive(true);
        UpdateAnimation();
    }
}
```

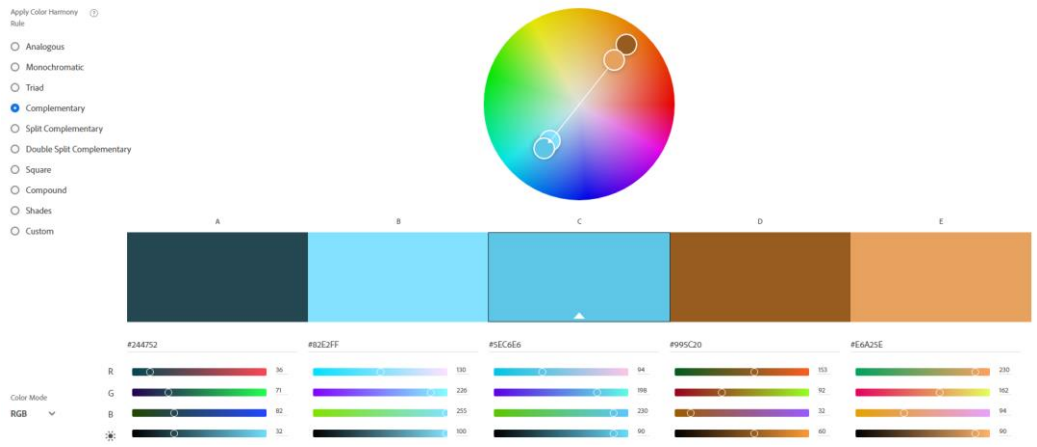
Daca inițializarea a reușit putem activa modelul și actualiza animația. Animația se activează dacă butonul de trigger sau cel de grip este apăsat.

```
void UpdateAnimation()
{
    // Try getting grip value. If unable, value is 0
    if (targetDevice.TryGetFeatureValue(CommonUsages.grip, out float gripValue))
    {
        handAnimator.SetFloat("Grip", gripValue);
    }
    else
    {
        handAnimator.SetFloat("Grip", 0);
    }

    // Try getting trigger value. If unable, value is 0
    if (targetDevice.TryGetFeatureValue(CommonUsages.trigger, out float triggerValue))
    {
        handAnimator.SetFloat("Trigger", triggerValue);
    }
    else
    {
        handAnimator.SetFloat("Trigger", 0);
    }
}
```

5.4.7. Meniul de navigație

Pentru meniu am ales culori complementare fundalului pentru a oferi o estetică bună scenei. Aceste culori au fost generate de Adobe Color în felul următor.



Pentru interfață trebuie să folosim un *canvas* în care vom atașa mai multe elemente de interfață. În cazul de față, la meniul principal vom avea 3 butoane care au text de tipul *TextMeshPro.* Această librărie oferă un mod mai avansat de a crea text cu rezoluție mai clară, gradient, texturi și multe alte facilități.

Figure 67 Meniul

Aceste trei butoane vor duce către una dintre cele trei scene de joc. *Explore World* este modul principal de joc. Acesta oferă posibilitatea de explorare a unei lumi subacvatice. *Analyse Marching Cubes* oferă posibilitatea controlului unei suprafețe generate în mod automat cu ajutorul VR-ului. *Analyse Boids* oferă posibilitatea controlului comportamentului boizilor.

Pentru controlul setărilor din ultimele două moduri de rulare a aplicației se folosesc intrările de la controlerul de VR. Am realizat un meniu care, prin intermediul unor scripturi realizează conexiunea cu proprietățile obiectului în cauză.

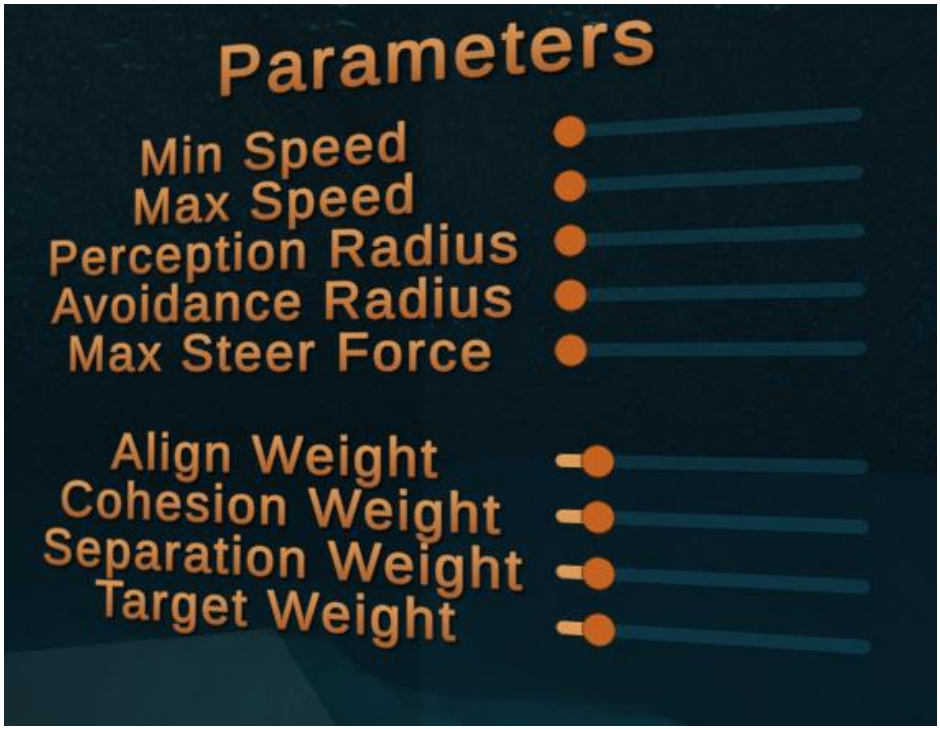


Figure 68 Exemplu de meniu de control

```
public class OnSliderValueChanged : MonoBehaviour
{
    private NoiseDensity nd;


    private void Start()
    {
        nd = GetComponent<NoiseDensity>();
    }


    private void UpdateMesh()
    {
        if (FindObjectOfType<MeshGenerator>())
        {
            FindObjectOfType<MeshGenerator>().RequestMeshUpdate();
        }
    }


    public void OnSeedValueChanged(float value)
    {
        nd.seed = Mathf.RoundToInt(value);
        UpdateMesh();
    }


    public void OnOctaveValueChanged(float value)
    {
        nd.numOctaves = Mathf.RoundToInt(value);
        UpdateMesh();
    }
```

Figure 69 Exemplu de cod pentru conectarea meniului

Conectarea se face prin intermediul unui eveniment de tipul *OnSliderValueChange.* Când valoarea slider-ului se schimbă se schimbă și valorile parametrilor algoritmilor, după anumite intervale predeterminate pentru a nu depăși anumite valori care ar duce la proasta funcționare a algoritmului.

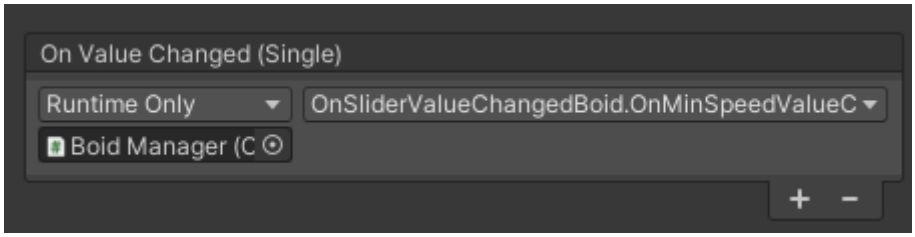


Figure 70 Interfața unui eveniment OnSliderValueChange

5.4.8. Controlul direcției peștilor

Toate aceste interfețe oferă posibilitatea utilizatorului să interacționeze cu aplicația. Am adăugat o nouă caracteristica, și anume, controlul direcției în care merg peștii. În meniul dedicat controlului parametrilor peștilor există și o opțiune pentru atașarea unei *"momeli"* la controler-ul stâng al utilizatorului. Persoana va putea mișca momeala, iar pești, în funcție de puterea țintei, se vor îndrepta către aceasta.

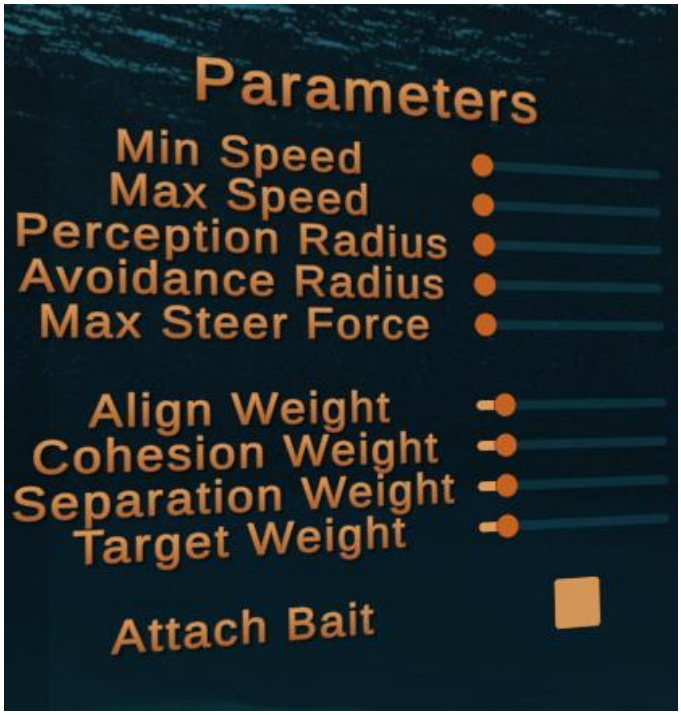


Figure 71 Meniul pentru controlul peștilor și atașarea momelii

Momeala este reprezentată de o sferă, dar aceasta ar putea fi reprezentată de orice.

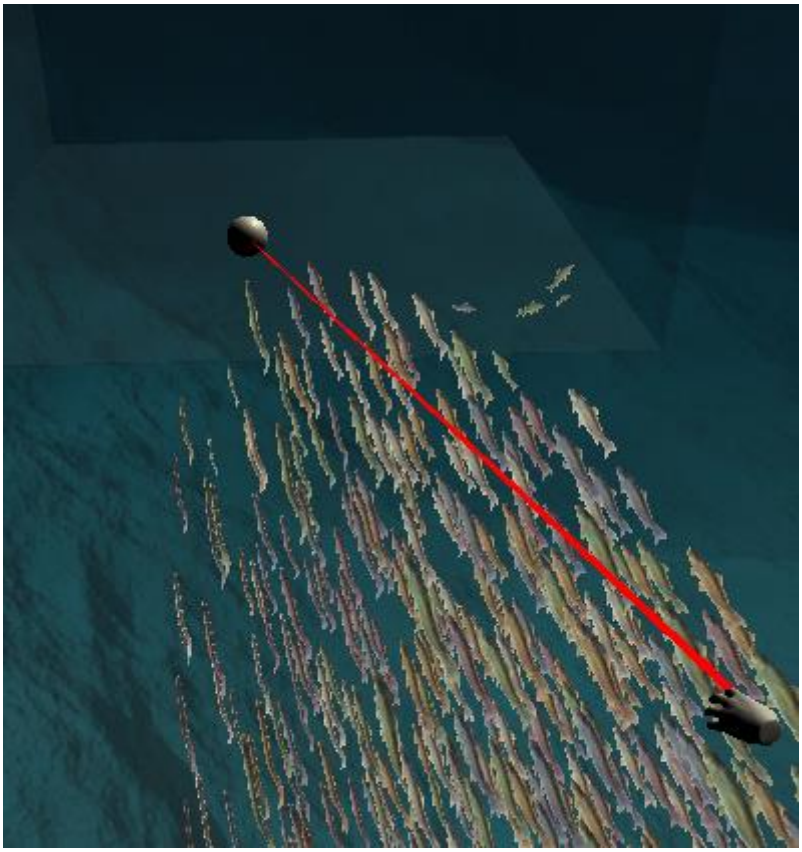


Figure 72 Momeala pentru pești

Pentru implementarea acestui *feature* a trebuit să modific codul pentru un boid și să creez un script care să controleze acest tip de comportament.

```
0 references
void Start()
{
    boids = FindObjectsOfType<Boid>();
    shape = Instantiate(shape, transform.position, Quaternion.identity, transform);
    enableTargeting = false;
}

0 references
void Update()
{
    if (enableTargeting && boids != null)
    {
        shape.transform.position = new Vector3(transform.position.x, transform.position.y, transform.position.z + radius);
        Vector3 pos = RotateAroundPivot(shape.transform.position, transform.position, transform.rotation.eulerAngles);
        shape.transform.position = pos;

        foreach (Boid b in boids)
        {
            b.targetFromVR = shape.transform;
            b.UpdateBoid();
        }
    } else
    {
        shape.transform.position = new Vector3(0, 0, 0);
    }

}
```

Se instanțiază forma dată (în cazul de față, forma unei sfere) și se caută toate obiectele de tip boizi. În cazul în care am bifat checkbox-ul (a cărei implementare s-a realizat la fel ca și implementarea celorlalte elemente de tip meniu), se atașează o bilă pe

post de momeală la o distanță dată față de utilizator. Bila se rotește în jurul persoanei în funcție de modul în care rotește aceasta controler-ul. De aceea trebuie să alegem ca pivot persoana și să rotim în jurul acesteia bila. Pentru fiecare boid se updatează destinația țintă.

Testare şi Validare

Iau tabel de la C3. Daca pot rula pe diverse calculatoare.

Aproximativ 5% din total.

Capitolul 6. Manual de Instalare si Utilizare

Print screen-uri utilizare + resurse hard/soft

În secţiunea de Instalare trebuie să detaliaţi resursele software şi hardware necesare pentru instalarea şi rularea aplicaţiei, precum şi o descriere pas cu pas a procesului de instalare. Instalarea aplicaţiei trebuie să fie posibilă pe baza a ceea ce se scrie aici.

În acest capitol, trebuie să descrieţi cum se utilizează aplicaţia din punct de vedere al utilizatorului, fără a menţiona aspecte tehnice interne. Folosiţi capturi ale ecranului şi explicaţii pas cu pas ale interacţiunii. Folosind acest manual, o persoană ar trebui să poată utiliza produsul vostru.

Capitolul 7. Concluzii

Dezvoltari ulterioare

Cca. 5% din total.

Capitolul ar trebui sa conţină (nu se rezumă neapărat la):

- un rezumat al contribuţiilor voastre
- analiză critică a rezultatelor obţinute
- descriere a posibilelor dezvoltări şi îmbunătăţiri ulterioare

Bibliografie

Capitol 1 + 2 + 3 + 4 in ordine alfabetica dupa autor

[1] A. Bak, S. Bouchafa, and D. Aubert, "Detection of independently moving objects through stereo vision and ego-motion extraction," in *IEEE Intelligent Vehicles Symposium (IV)*, San Diego, USA, 2010, pp. 863-870.

[2] A. Chambolle and T. Pock, "A First-Order Primal-Dual Algorithm for Convex Problems with Applications to Imaging," *Journal of Mathematical Imaging and Vision*, vol. 40, pp. 120-145, 2011.

[3] R. C. Gonzalez and R. E. Woods, *Digital Image Processing. Second Edition.*: Addison-Wesley Longman Publishing Co., Inc., 2001.

[4] Ajax Tutorial, http://www.tutorialspoint.com/ajax/.

Anexa 1 (dacă este necesar)

…
Secţiuni relevante din cod
…
Alte informaţii relevante (demonstraţii etc.)
…
Lucrări publicate (dacă există)
etc.