Ramon Donadeu Caballero

GENERACIÓ PROCEDURAL DE PLANETES

TREBALL DE FI DE GRAU

dirigit per Jordi Duch Gavaldà

Grau d'Enginyeria Informàtica

UNIVERSITAT ROVIRA I VIRGILI

Tarragona
2021

Resum.

La generació procedural és el procés pel qual a partir d'un conjunt d'algoritmes, es pot generar informació de manera pseudoaleatori. Aquesta informació pot ser tant terreny, ciutats, textures, objectes... Aquesta tecnologia ofereix una gran descàrrega de feina per a programadors i dissenyadors de videojocs, ja que permet generar milions d'iteracions diferents d'una manera molt senzilla.

L'objectiu de projecte és permetre a l'usuari introduir un conjunt de paràmetres en l'entorn de Unity per poder generar un planeta al seu gust.

En aquest document analitzarem diferents formes de generar terreny realista en un planeta, s'explicarà com funcionen els algoritmes de soroll i tot el procés de creació que s'ha dut a terme, des de la creació de l'esfera, passant per la creació de muntanyes fins a la coloració del planeta amb diferents biomes.

Resumen.

La generación procedural es el proceso por el cual a partir de un conjunto de algoritmos, se puede generar información de manera pseudoaleatoria. Esta información puede ser terreno, ciudades, texturas, objetos... Esta tecnología ofrece una gran descarga de Trabajo para programadores y diseñadores de videojuegos, ya que permite generar millones de iteraciones diferentes de una forma muy sencilla.

El objetivo del proyecto es permitir al usuario introducir un conjunto de parámetros en un entorno de Unity para poder generar un planeta a su gusto.

En este documento analizaremos las diferentes formes para generar terreno realista en un planeta, se explicará como funcionan los algoritmos de ruido y todo el proceso de creación que se ha llevado a cabo, des de la creación de la esfera, pasando por la creación de montañas hasta la coloración del planeta con diferentes biomas.

Abstract.

Procedural generation is the process by which, from a set of algorithms, information can be generated in a pseudo-random way. This information can be terrain, cities, textures, objects... This technology offers a great discharge of work for programmers and game designers, as it allows to generate millions of different iterations in a very simple way.

The aim of the project is to allow the user to enter a set of parameters in a Unity environment to generate a planet to their liking.

In this document we will analyze the different ways to generate realistic terrain on a planet, we will explain how the noise algorithms work and the whole process of creation that has been carried out, from the creation of the sphere, through the creation of mountains to the coloring of the planet with different biomes.

Índex

1. INTRODUCCIÓ ... 3
1.1 MOTIVACIÓ ... 3
1.2 OBJECTIUS ... 3
1.3 ESTRUCTURA DE LA MEMÒRIA ... 4
2. RECERCA PRÈVIA ... 5
2.1 GENERACIÓ PROCEDURAL ... 5
2.2 SOROLLS ... 5
2.2.1 Perlin Noise ... 6
2.2.2 Simplex Noise ... 8
2.2.3 Ridged Noise ... 9
2.2.4 Fractal Noise ... 11
2.3 DIAGRAMA DE VORONOI ... 12
2.4 MESH ... 13
2.5 CREACIÓ D'UNA ESFERA ... 14
2.5.1 Esfera a Partir d'un Cub ... 14
2.5.2 Esfera UV ... 15
2.5.3 Esfera a Partir d'un Icosaedre ... 16
2.5.4 Esfera a Partir d'un Octaedre ... 17
3. IMPLEMENTACIÓ ... 19
3.1. PRIMER PROTOTIP ... 20
3.2 CREACIÓ DE L'ESFERA ... 25
3.4 GENERACIÓ DE COLORS ... 30
3.5 INTERFÍCIE ... 36
3.6 PROBLEMES DURANT EL DESENVOLUPAMENT DEL PROGRAMA FINAL. ... 38
4. RESULTATS ... 40
5. TREBALL FUTUR ... 44
6. CONCLUSIONS ... 45
7. REFERÈNCIES ... 46

Índex de figures

FIGURA 1. SOROLL BLANC. 6
FIGURA 2. SOROLLS DE PERLIN 6
FIGURA 3. GRAELLA PER CALCULAR ELS GRADIENTS. 7
FIGURA 4. RESULTAT DEL PRODUCTE ESCALAR A CADA PUNT DE LA QUADRÍCULA 7
FIGURA 5. RESULTAT DE LA SUMA DELS PRODUCTES ESCALARS DELS GRADIENTS. 8
FIGURA 6. ARTEFACTES GENERATS D'UN FRACTAL NOISE A PARTIR D'UN PERLIN NOISE 8
FIGURA 7. SÍMPLEX. 9
FIGURA 8. RESULTAT DEL RIDGE NOISE. 9
FIGURA 9. GRÀFIC RESULTANT DE LA FUNCIÓ COS(X). 10
FIGURA 10. GRÀFIC RESULTANT DE LA FUNCIÓ: 1-|COS(X)|. 10
FIGURA 11. GRÀFIC RESULTANT DE LA FUNCIÓ: (1-|COS(X)|)^2. 11
FIGURA 12. EXEMPLE DELS DIFERENTS OCTAUS D'UN FRACTAL NOISE. 11
FIGURA 13. RESULTAT DEL FRACTAL NOISE 12
FIGURA 14. DIAGRAMA DE VORONOI 12
FIGURA 15. RESULTAT DE QUATRE RELAXACIONS DE LLOYD SOBRE UN DIAGRAMA DE VORONOI. 13
FIGURA 16. ESFIRIFICACIÓ D'UN CLUB. 14
FIGURA 17. VISTA FRONTAL D'UN CUB ESFERIFICAT. 15
FIGURA 18. ESFERA UV. 16
FIGURA 19. CREACIÓ D'UN ICOSAEDRE A PARTIR DE TRES PLANS PERPENDICULARS 16
FIGURA 20. GRÀFIC QUE MOSTRA EL NOMBRE DE TRIANGLES PER ITERACIÓ. 17
FIGURA 21. GRÀFIC QUE MOSTRA EL NOMBRE DE TRIANGLES PER CARA PER NOMBRE DE VÈRTEX PER ARESTA. 18
FIGURA 22. DIAGRAMA DE GANTT DEL PRIMER PROTOTIP. 19
FIGURA 23. DIAGRAMA DE GANTT DEL PROTOTIP FINAL 19
FIGURA 24. FRAGMENT D'UN DIAGRAMA DE VORONOI DE 8000 POLIGONS. 20
FIGURA 25. ESTRUCTURA GENERADA PER L'ALGORITME DE VORONOI. 21
FIGURA 26. TRIANGULACIÓ D'UN POLÍGON. 21
FIGURA 27. TRIANGULACIÓ INCORRECTA D'UN POLÍGON 22
FIGURA 28. ALGORITME DE GIFT WRAPING. 22
FIGURA 29. DIAGRAMA DE VORONOI AMB ELS POLÍGONS TRIANGULATS. 23
FIGURA 30. RESULTAT DEL PRIMER PROTOTIP AMB 30000 POLÍGONS 24
FIGURA 31. GRÀFIC QUE MOSTRA EL TEMPS DE GENERACIÓ DE CADA PART DE L'ALGORITME RESPECTE AL NOMBRE DE POLÍGONS SELECCIONATS. 24
FIGURA 32. PROCÉS DE COM ES CREA LA TRIANGULACIÓ DE LES CARES DE L'OCTAEDRE. 28
FIGURA 33. TEXTURA D'UN PLANETA 34
FIGURA 34. INTERFÍCIE DEL SHADER, PRIMERA PART. 35
FIGURA 35. INTERFÍCIE DEL SHADER, SEGONA PART. 35
FIGURA 36. INTERFÍCIE DE GENERACIÓ DEL PLANETA. 36
FIGURA 37. INTERFÍCIE PER MODIFICAR ELS VALORS DE GENERACIÓ D'ALTURA. 37
FIGURA 38. PRIMERA PART DE LA INTERFÍCIE PER MODIFICAR ELS COLORS DEL PLANETA. 37
FIGURA 39. EDITOR DEL COLOR DEL BIOMA. 38
FIGURA 40. SEGONA PART DE LA INTERFICIE PER MODIFICAR ELS COLORS DEL PLANETA. 38
FIGURA 41. PLANETA AMB TEXTURA COMPRIMIDA. 39
FIGURA 42. TEXTURA COMPRIMIDA 39
FIGURA 43. PLANETA D'EXEMPLE 1. 40
FIGURA 44. OPCIONS DEL PLANETA 1. 41
FIGURA 45. PLANETA D'EXEMPLE 2. 41
FIGURA 46. OPCIONS DEL PLANETA 2 42
FIGURA 47. PLANETA D'EXEMPLE 3. 42
FIGURA 48. OPCIONS DEL PLANETA 3 43

1. Introducció

En aquest document parlaré del procés d'aprenentatge i creació d'un generador de planetes procedural amb opcions de personalització introduïdes per l'usuari. Explicaré diferents formes de generar terreny i tot el procés que he dut a terme.

La generació procedural és el procés pel qual un ordinador genera informació a partir d'un algoritme de manera pseudoaleatòria. Aquesta tecnologia és molt utilitzada en el món dels videojocs i cada cop de manera més ambiciosa, ja que permet milions de generacions diferents, ja sigui de terreny, de textures, animacions… Ja que permet alliberar una gran quantitat de feina als dissenyadors de videojocs.

En aquest projecte em centraré en la generació de terreny a partir d'algoritmes de soroll coherents, que són algoritmes que generen una imatge aleatòria amb valors d'entre 1 i 0 de manera que els punts adjacents tenen una relació directa, és a dir, dos punts adjacents tindran valors semblants per crear una imatge amb transicions suaus.

Tot i que els algoritmes poden ser completament aleatoris, en aquest cas, es permetrà certa participació de l'usuari, per tal que pugui configurar com vulgui cada un dels planetes, així com la quantitat d'elevacions, la seva mida, o també el color dels planetes i la seva distribució.

1.1 Motivació

Des de fa temps tinc com a passatemps l'activitat de crear mons, amb els seus continents, ciutats i històries, i sempre havia volgut crear un programa que em permetés fer això d'una manera més senzilla. És per aquesta raó que vaig proposar aquest treball, tot i que finalment es va encaminar més en la direcció d'un generador de terreny.

Com he comentat a l'apartat anterior, en el món dels videojocs, en el que també hi soc aficionat, s'utilitza la generació aleatòria de terreny, i és probable que en un futur, això sigui una cosa normal. Amb els avanços que s'estan fent en el camp de la intel·ligència artificial, no seria una bogeria pensar què en un futur, un joc fos capaç de crear per a cada usuari un món sencer amb les seves ciutats, muntanyes, rius i amb l'ajuda de la intel·ligència artificial es poguessin generar històries diferents per cada jugador, i que aquestes històries tinguessin en compte el comportament del jugador. Així es podrien generar pràcticament jocs infinits i cada persona tindria una experiència única basada en com es comporten dins aquest món.

1.2 Objectius

Els objectius d'aquest treball en un principi eren poder generar un planeta aleatòriament i permetre a l'usuari fer canvis en el terreny al seu gust. Els objectius principals que em vaig proposar van ser generar terreny realista i donar color, ja que per crear aquest planeta vaig haver d'utilitzar un entorn gràfic que mai havia fet servir, aquest programa és anomenat Unity, cosa que finalment m'ha comportat problemes a causa del meu desconeixement bàsic de com s'ha de programar amb objectes i textures, conceptes que per mi eren nous i en els quals he hagut d'invertir moltes hores d'aprenentatge.

1.3 Estructura de la Memòria

Aquest document es divideix en cinc parts, la primera de totes inclou la recerca prèvia al desenvolupament del programa, en ella s'hi explica quines tecnologies s'usaran i com funcionen junt amb comparacions entre diferents implementacions plantejades.

A continuació es troba la implementació del projecte, primer es pot trobar l'explicació del primer prototip que vaig dur a terme i tot seguit el projecte final, on s'explicarà l'estructura dels fitxers del programa i on també es detallarà el funcionament en concret d'algunes funcions.

Tot seguit es podran trobar tres exemples de planetes generats amb aquest programa junt amb el paràmetres utilitzats en cada un d'ells, com a demostració dels resultats que es poden obtenir.

En quart lloc, hi ha un petit apartat on s'explica possibles modificacions i millores futures per al programa, que, tot i que havent assolit l'objectiu plantejat, és possible millorar-lo per tal que sigui més atractiu.

Per acabar, les conclusions, on explicaré com ha sigut tot el procés del projecte junt amb els coneixements i resultats obtinguts durant tot el desenvolupament.

2. Recerca Prèvia

2.1 Generació Procedural

La generació procedural és el procés pel qual es crea informació a partir d'un algoritme junt amb un conjunt de regles i valors introduïts per l'humà.

Aquesta tecnologia pot ser utilitzada per obtenir objectius diferents, per exemple, es poden crear textures, models tridimensionals d'objectes, o com en aquest cas, terreny aleatori.

L'ús de la generació procedural és molt utilitzada en l'entorn dels videojocs, ja que dona lloc a la generació aleatòria i alhora, automatitzada d'escenaris, amb això podem aconseguir dues coses. Primer de tot, alliberar una gran quantitat de feina als dissenyadors, que no han de crear diferents escenaris, zones o mapes, sinó que tenen un algoritme que genera aquests recursos automàtics, de manera, que si fos necessari generar un món massa gran per a un grup de treballadors, podrien usar aquesta tecnologia per obtenir una base sobre on començar a treballar i estalviar-se una gran quantitat de feina.

Aquest és un recurs molt utilitzat en un tipus de joc anomenat *Roguelike* o *Rogelite.* En aquests jocs, el jugador du a terme petites partides de poca durada on va investigant un mapa generat proceduralment, un cop, per qualsevol raó, el jugador perdi la partida, tornarà a començar una nova des del principi, però amb la diferència que cada cop, el mapa generat de nou, per tant, cada partida és diferent de l'anterior. O també s'utilitza en jocs de supervivència o de *sandbox* [1] en els que cada partida es genera un mapa pràcticament infinit i aleatori a partir d'una *seed* [2].

D'aquesta manera, un sol joc pot tindre iteracions de mapes pràcticament infinites i per una altra banda, totes aquestes iteracions han sigut creades per l'ordinador, de manera que els programadors no les han hagut de fer una a una, cosa que comportaria un volum de feina molt gran.

Tenint en compte això, es pot trobar un altre avantatge d'aquesta tecnologia, la reducció d'ús de memòria.

Ja que els mapes es generen automàticament, no es requereix de l'espai de memòria necessari per guardar tots aquests mapes diferents, simplement es crearan aleatòriament quan es necessitin.

2.2 Sorolls

Dins el procés de generació procedural, és podem utilitzar diferents algoritmes o mètodes, per aconseguir l'objectiu proposat. En aquest cas, s'han de generar diferents altures i punts per obtenir un terreny realista, és per això que utilitzaré sorolls.

---

[1] Joc en el que l'usuari no te cap meta en concret i pot fer el que vulgui.

[2] *Seed*, o en català *Llavor,* és un valor del qual a partir es genera un resultat procedural, i amb cada llavor sempre es donarà el mateix resultat.

El soroll, *noise* en angles, és un conjunt de valors aleatoris que pot donar lloc a un so, una imatge, o un objecte tridimensional, si s'interpreten els valors com a una, dues o tres dimensions respectivament.

Un soroll que tothom coneix vindria a ser el soroll blanc, que és aquell que veiem fa uns anys quan enceníem la televisió i no estava sintonitzada correctament, de manera que es veia un conjunt de punts en blanc i negre per tota la pantalla junt amb el soroll desagradable que emetien els altaveus. Aquests són dos exemples de soroll, un és visual i l'altre auditiu.

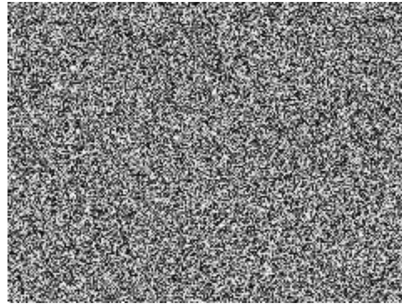


Figura 1. Soroll Blanc.

Existeixen diferents tipus de sorolls aleatoris, cada un anomenat sobre un color, blanc, lila, marró… Però per generar un mapa d'altures necessitem sorolls que tinguin una correlació entre els diferents punts adjacents, de manera que són sorolls pseudoaleatoris.

2.2.1 Perlin Noise

El Perlin Noise va ser inventat per Ken Perlin l'any 1985 quan buscava una forma de generar imatges generades per ordinador.

Figura 2. Sorolls de Perlin

Per generar un soroll de Perlin utilitzem un algoritme en el qual se li introdueixen unes coordenades i la funció retorna un valor entre 1 i -1, l'algoritme consta de tres passos:

<u>Definició de la graella:</u>

Es genera una graella d'espais de forma quadriculada, en dues, tres o quatre dimensions on les arestes tenen una longitud unitària. A partir de les interseccions que es generen, és a dir, els vèrtexs de tots els quadrats, es generarà un gradient pseudoaleatori, aquests gradients van des del valor 1 al -1.

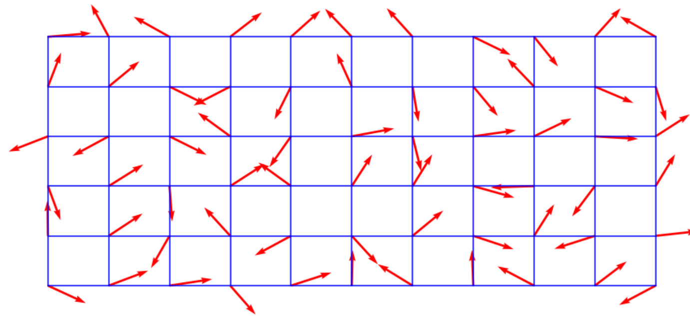


Figura 3. Graella per calcular els gradients.

Producte escalar:

A continuació, a partir de les coordenades que han sigut introduïdes a l'algoritme es calcularà un vector des de les coordenades cap als vèrtexs del requadre en el qual es troba aquesta coordenada i es farà el producte escalar entre cada un d'aquests vectors amb el gradient corresponent a cada vèrtex.

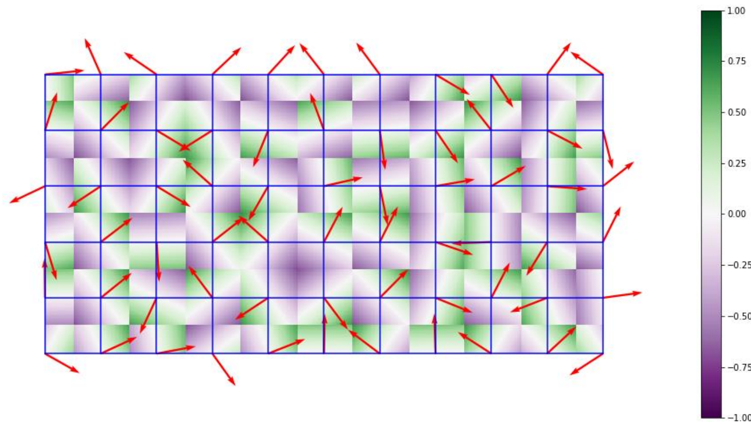


Figura 4. Resultat del producte escalar a cada punt de la quadrícula.

Com es pot veure a la imatge, si tracem una línia perpendicular de cada un dels vectors, podem observar com aquest nou vector està situat sobre una zona blanca, això és pel fet que dos vectors perpendiculars tenen un producte escalar 0. Si els dos vectors tenen la mateixa direcció, obtindrem un valor d'1 i si van en direccions oposades, tindrà valor -1.

Interpolació:

El pròxim pas és, a partir dels valors que hem obtingut, fer la mitjana entre tots ells per obtenir el valor final entre 1 i -1 que donarà un mapa d'altures coherent.

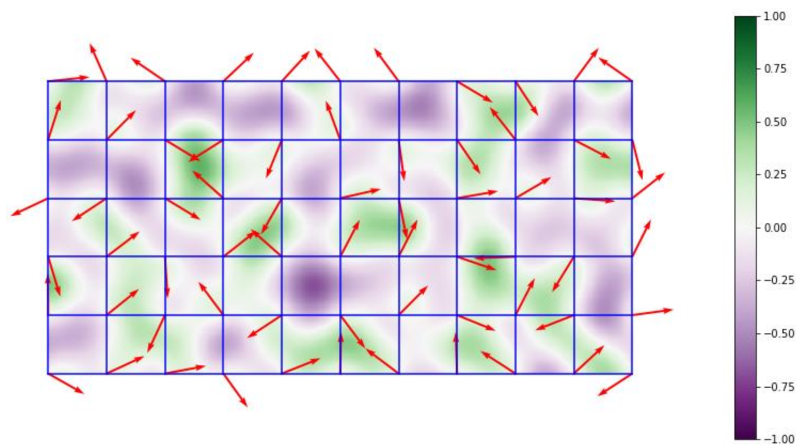


Figura 5. Resultat de la suma dels productes escalars dels gradients.

Un problema que s'adreçarà en un futur, és la generació d'artefactes en els mapes generats per aquest soroll. Els artefactes són anomalies que es presenten en una representació digital d'una imatge. En aquest cas causada per com es calculen els gradients dels vèrtexs anteriorment explicats.

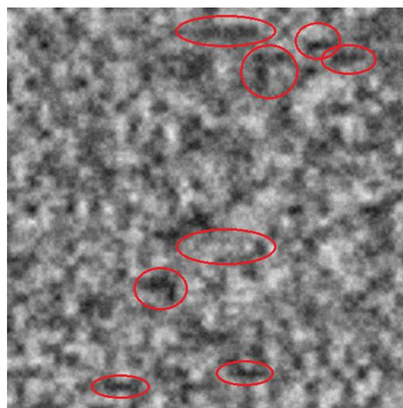


Figura 6. Artefactes generats d'un Fractal Noise a partir d'un Perlin Noise

Com es pot veure en aquesta imatge, hi ha encerclades unes zones on es pot veure clarament zones estranyament rectes quan estan sent calculades per un algoritme aleatori.

2.2.2 Simplex Noise

Aquest soroll és una modificació del Perlin Noise fet també pel mateix creador, però aquest algoritme està optimitzat.

Per començar, tots els càlculs dels gradients que s'han comentat a l'apartat anterior es fan a partir d'una estructura de triangles en comptes de quadrats, això és degut al fet que, els triangles són una manera òptima d'omplir un espai. Aquest algoritme rep el seu nom dels símplex, que és aquesta estructura triangular amb la que es calcula aquest soroll.

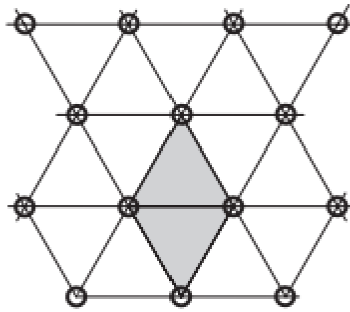


Figura 7. Símplex.

Pel fet d'utilitzar aquesta estructura ens estalviem una operació quan hem d'interpolar tots els resultats, ja que tindrem només tres vectors en comptes de quatre. S'han fet estudis en els quals, utilitzar Perlin Noise té un cost computacional de O(N*2^N) i en canvi Símplex Noise té un cost de O(N^2).

A banda d'això, també es va modificar perquè funciones amb espais de quatre i cinc dimensions, també s'eliminen artefactes generats per l'algoritme antic.

2.2.3 Ridged Noise

Amb els sorolls anteriors es poden crear terrenys bastant realistes, tot i això, les muntanyes acostumen a ser estructures més complexes, amb el Simplex Noise podem obtenir diferents nivells d'altura, però quan es genera un punt alt, acostuma a tindre un pendent molt suavitzat, en canvi les muntanyes tenen pendents majors com més a prop del cim ens trobem, aquí és on entra la utilització del Ridge Noise.

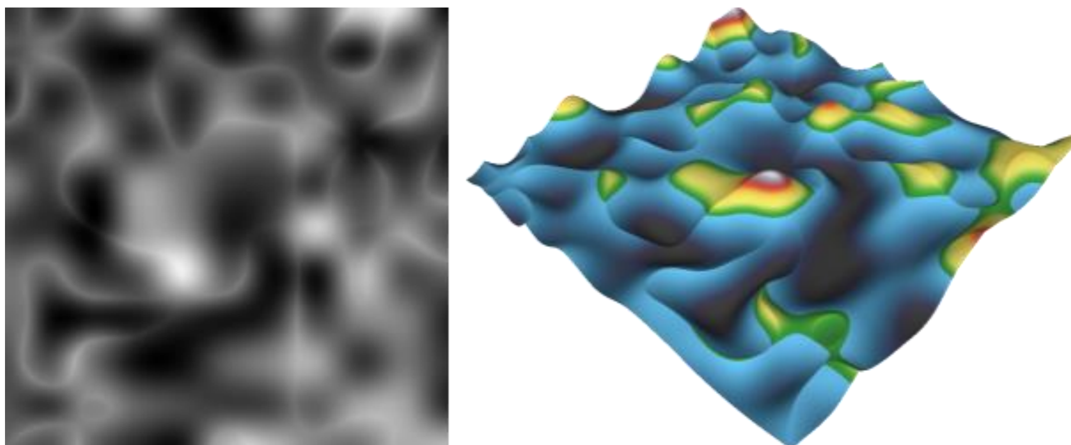


Figura 8. Resultat del Ridge Noise.

El Ridge Noise, en català soroll de cim, rep el seu nom per la forma que genera de cims de muntanyes com es pot veure a les imatges anteriors.

Aquests càlculs es faran sobre els diferents punts que s'obtindran dels algoritmes anteriors, però per fer una explicació més visual, en comptes de calcular punts aleatoris utilitzarem la funció del cosinus.

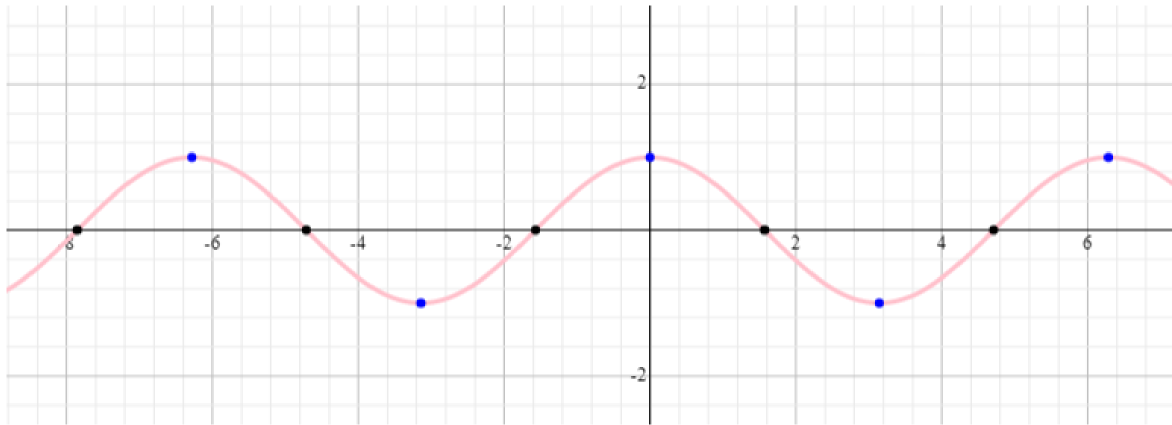


Figura 9. Gràfic resultant de la funció cos(x).

Un cop l'algoritme de soroll, en aquest cas el cosinus, ha retornat un valor per al punt que s'ha introduït es calcularà el valor absolut i es restarà aquest valor a 1 de manera que quedarà la següent forma.

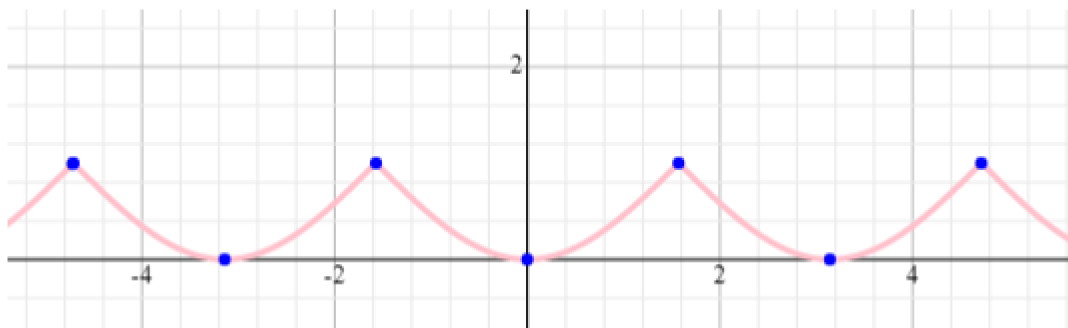


Figura 10. Gràfic resultant de la funció: 1-|cos(x)|.

Com es pot observar, en aquest cas es genera una forma més semblant a una muntanya, però tot i això, per donar més èmfasi a les cimes de les muntanyes, s'elevarà al quadrat el resultat de l'operació anterior, fent així que els punts més baixos ho siguin encara mes.

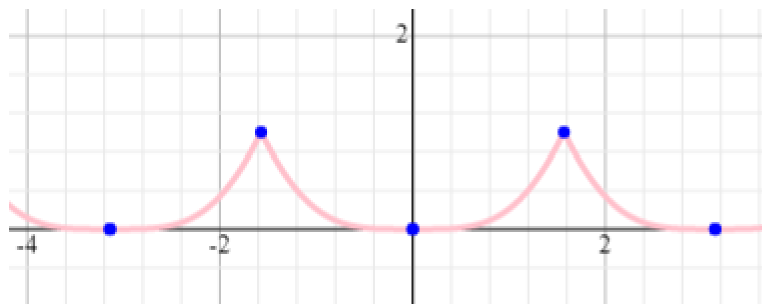


Figura 11. Gràfic resultant de la funció: (1-|cos(x)|)^2.

2.2.4 Fractal Noise

Aquest tipus de soroll no utilitza un algoritme específic, sinó que, com també el Ridge Noise, utilitza els sorolls generats pel Simplex Noise i el Perlin Noise i genera un resultat diferent. En aquest cas, el que s'aconsegueix és donar detall al soroll generat, ja que, com hem vist en imatges anteriors, és un soroll amb uns canvis suaus, però en un terreny tenim més desnivells petits.

En aquest cas, l'usuari té més opcions de personalització que comentaré a continuació:

- Amplitud: Permet augmentar l'alçada dels punts retornats per l'algoritme de soroll
- Persistència: Aquest valor permet controlar com varia el valor de l'amplitud depenent del octau en el qual es trobi l'algoritme.
- Freqüència: Serveix per disminuir l'espai entre diferents cims, bàsicament estem fent que l'algoritme rebi com a input valors més allunyats, per tant semblaria que el resultat del soroll estigues disminuint el zoom de la imatge resultant.
- Lacunaritat: Igual que la persistència, permet controlar com es modifica la freqüència.
- Octaus: Aquest paràmetre és el que permet que tots els anteriors tinguin un efecte en el resultat final. Per cada octau que hagi introduït l'usuari, es repetirà el càlcul de cada un dels punts que s'introduiran a l'algoritme amb tots els paràmetres introduïts anteriorment.

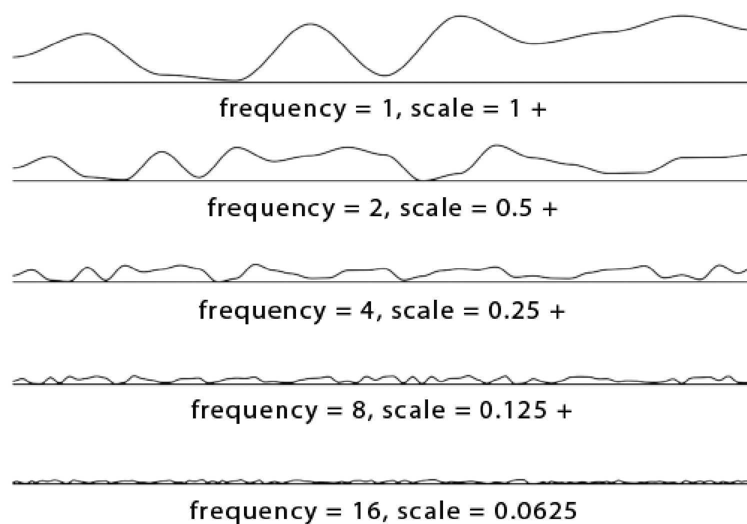


Figura 12. Exemple dels diferents octaus d'un Fractal Noise.

Com es pot veure en aquesta imatge, tenim cinc octaus, per tant es repetirà el procés cinc cops. Es comença amb un resultat suavitzat sense gaire detall, a la següent iteració s'augmenta la freqüència, per tant hi ha més quantitat de desnivells, però la seva altura és més baixa, ja que l'amplitud ha baixat, a la imatge l'amplitud és anomenada *scale*.

En les primeres iteracions obtenim la forma general del terreny, mentre cap al final obtindrem petits detalls que es veuran reflectits als resultats anteriors.

Se sumen tots els valors de cada iteració per obtenir la posició d'un sol punt, i aquest procés es repeteix per tots els punts a calcular per obtenir el següent resultat:

Figura 13. Resultat del Fractal Noise.

2.3 Diagrama de Voronoi

Una altra forma de generar terreny, és a partir d'un diagrama de Voronoi, és una construcció geomètrica que es genera a partir d'un conjunt de punts aleatoris, a partir d'aquests punts es generaran unes àrees que seran delimitades a partir de la distància euclidiana entre els punts adjacents. És a dir, a partir d'un conjunt de punts es buscarà la distància al voltant d'un punt en concret comparat amb la resta de punts del seu voltant, per crear una àrea on només hi haurà un sol punt. Per tant, el punt més proper de qualsevol lloc d'una àrea sempre serà el punt que estigui dins d'aquesta.

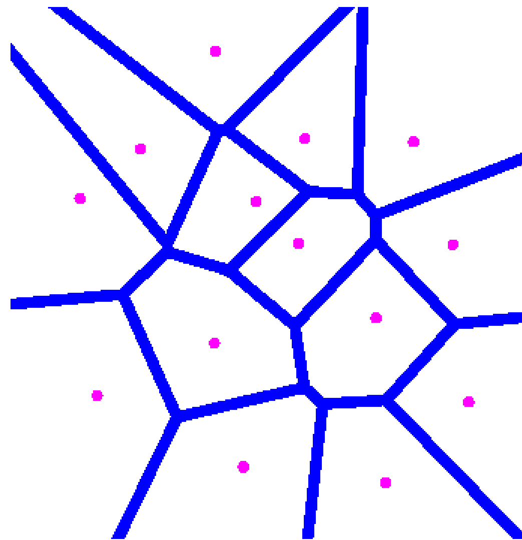


Figura 14. Diagrama de Voronoi.

Aquest algoritme s'utilitza en molts àmbits, per exemple, serviria per calcular quin és el supermercat, hospital, estació de transport públic més propera, depenent de la informació que busquem, marcaríem els punts en un mapa i generaríem el diagrama, i per saber quin és el punt més proper que busquem sol ens hem de fixar en l'àrea en la qual ens trobem.

Per exemple, aquest diagrama va ser utilitzat durant una epidèmia de colera l'any 1854 per trobar la relació entre les morts causades per la colera i la proximitat de diferents fonts d'aigua per on es transmetia aquesta malaltia.

L'avantatge d'aquest mètode és que es generaran formes irregulars aleatòries, ja que es generarà el diagrama a partir d'un conjunt aleatori de punts. De manera que amb donar altures individuals a cada una de les àrees ja obtindríem un mapa d'altures per zones.

Per altra banda hi ha el problema que s'hauria de desenvolupar un algoritme de generació d'altures per donar una alçada diferent a cada àrea, ja que si es volgués utilitzar un algoritme de soroll, podria donar resultats estranys, ja que, si entre dos punts diferents ens fixéssim en el mapa de soroll, podria donar-se el cas que hi hagués una vall o una muntanya per exemple, però els dos punts que estem analitzant estiguin a una altura semblant, per tant estaríem perdent informació i detall al nostre mapa.

També s'ha de tindre en compte que utilitzar un diagrama de Voronoi tal com es genera, podria donar formes geomètriques estranyes per a un terreny, és per això que s'ha d'utilitzar amb un algoritme anomenat, Algoritme de Lloyd.

Aquest algoritme crea un espai semblant entre tots els punts que s'han d'analitzar, de manera que les àrees resultants de l'algoritme de Voronoi després d'usar Lloyd seran més regulars.

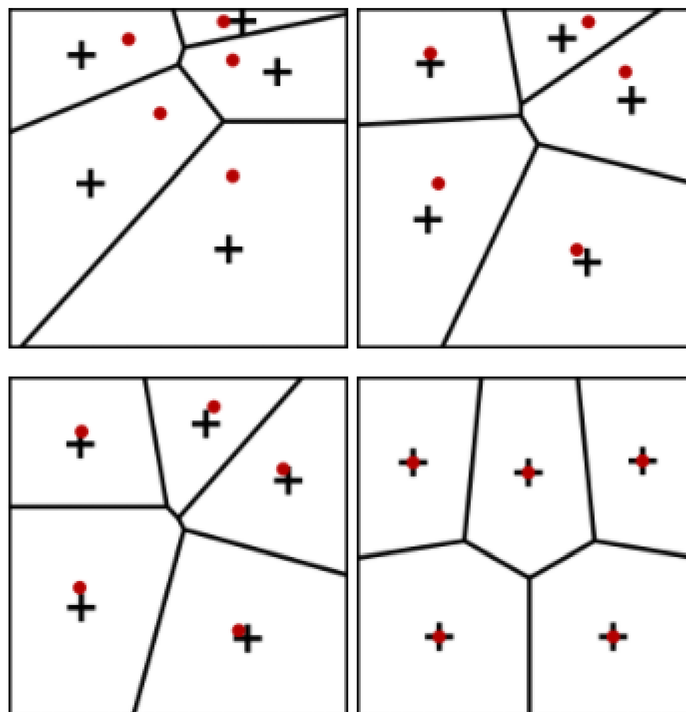


Figura 15. Resultat de quatre relaxacions de Lloyd sobre un Diagrama de Voronoi.

2.4 Mesh

Per representar el mapa tridimensional utilitzaré una mesh, o malla en català. Aquest element de Unity permet crear objectes a partir de la posició i unió de diferents vèrtexs en l'espai. És la manera en la qual es creen la gran majoria d'objectes d'un videojoc.

Una mesh té molts valors que es poden utilitzar per crear un objecte, però sol es comentaran els valors bàsics que són útils per entendre el funcionament del projecte.

Primer de tot, tenen una llista de vèrtex, aquesta llista és una llista del tipus *Vector3*, que en Unity és un objecte que conté tres coordenades. Aquesta llista conte tots els vèrtexs que s'han de representar per obtenir l'objecte desitjat.

En segon lloc, hi ha una altra llista de triangles, aquesta llista ha de contenir les unions entre els diferents vèrtexs. S'han d'escriure els índexs de cada vèrtex de tres en tres per tal que Unity interpreti correctament com s'ha de representar el triangle i també s'ha de tindre en compte que tots els vèrtexs s'han d'introduir en el mateix sentit, ja que la mesh només serà visible des del costat on els vèrtexs estiguin ordenats de manera horària.

2.5 Creació d'una Esfera

2.5.1 Esfera a Partir d'un Cub

El primer mètode parteix de la formació d'un cub, aquest cub està format per sis quadrats independents per formar el polígon, hi ha l'opció de generar un cub sencer sense els costats independents, cada opció té els seus avantatges i inconvenients.

En els dos casos el quadrat es divideix en diferents subquadres formant una graella i després aquests quadrats en triangles per poder crear la mesh. Tot seguit cada vèrtex se li afegeix el radi de l'esfera que es vol generar.

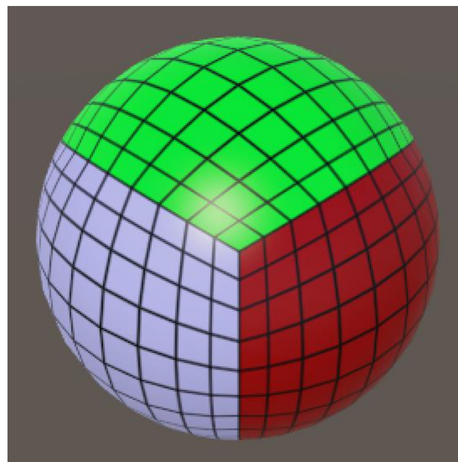


Figura 16. Esferificació d'un club.

El cub amb els costats independents és una bona opció en cas que es vulgui un bon rendiment si es volen fer modificacions en temps d'execució, ja que en ser cada costat independent es pot adaptar el detall de l'esfera depenent de la zona que aparegui en pantalla. Això és degut al fet que, si ens trobem en un dels costats de l'esfera, aquell costat pot tindre una resolució molt alta, mentre la resta de cares del cub tenen una baixa resolució. Ens referim a resolució com al nombre de polígons generats. Per tant, mentre s'estan visualitzant una gran quantitat de triangles a la zona del jugador, les altres zones que no són visibles tindran pocs triangles i per tant no utilitzaran recursos addicionals del programa.

Però aquest mètode té un inconvenient, ja que els diferents costats no comparteixen el vèrtex que conecta les diferents cares, en els vèrtexs que unirien els costats del cub es poden crear petits espais entre ells i per tant s'ha de tindre en compte alguna solució per resoldre-ho.

En canvi, si s'utilitza el cub sencer, aquest inconvenient no apareixerà, però per altra banda, no podem modificar la resolució del terreny depenent de la posició visible al programa.

Deixant de banda els dos mètodes per crear l'esfera, hi ha un inconvenient general amb aquest mètode. Aquest problema és que tots els quadrats que es generen no són regulars, és a dir, depenent de la posició del quadrat sobre cada pla del cub inicial, cada un tindrà una deformació més gran com més a prop de les arestes del cub es trobi.

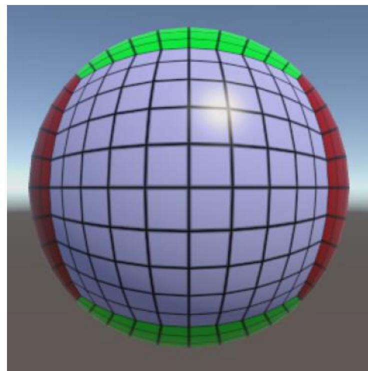


Figura 17. Vista frontal d'un cub esferificat.

Com es pot veure a aquesta imatge, els quadrats del centre mantenen la seva forma original, però els quadrats que són a l'aresta del cub són més prims, inclús al lloc d'unió de tres cares del cub, podem veure com el quadrat resultant és encara més petit que la resta. Això a altes resolucions no serà gaire notable, però és un inconvenient que s'ha de tindre en compte.

2.5.2 Esfera UV

Aquesta esfera rep el seu nom pel mapatge UV, que és el procés de transformar un objecte tridimensional a un pla de dues dimensions. Un exemple clar, que a més dona a entendre com funciona aquesta esfera és la forma en què es projecta el nostre planeta en un mapa.

Igual que el nostre planeta, aquesta forma comporta una deformació, ja que una esfera no es pot transformar fàcilment en una imatge bidimensional, és per això, que els mapes de la Terra són incorrectes, ja que les zones més a prop dels pols del planeta estan eixamplades per omplir l'espai que queda entre les separacions del mapa en 2D.

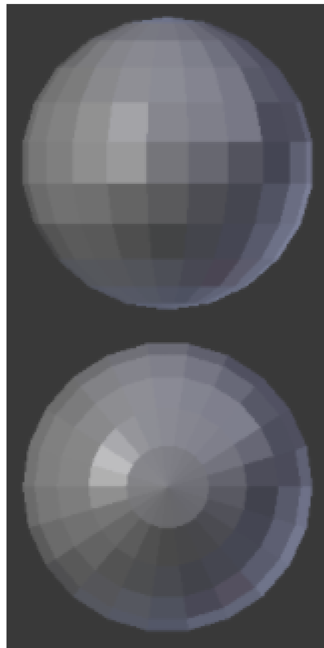


Figura 18. Esfera UV.

Com es pot veure a la imatge, la part superior de l'esfera està formada per quadrats que semblen triangles, en canvi al centre de l'esfera hi ha formes quadrades. Per aquesta raó, per aquest projecte, no és una bona opció usar aquest mètode.

2.5.3 Esfera a Partir d'un Icosaedre

En aquesta aproximació per crear una esfera tenim un nou component, els triangles. En els mètodes anteriors, l'estructura principal eren quadrats, i com hem vist, per tal d'esferificar un quadrat, aquest s'ha de deformar, això no succeeix amb els triangles.

Per crear un icosaedre es parteix de tres rectangles, cada un situat en una direcció diferent. Tot seguit, s'uneixen els punts dels diferents plans entre ells per crear el poliedre.

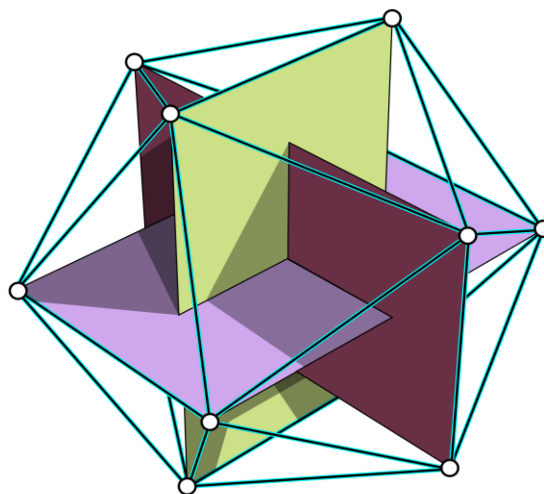


Figura 19. Creació d'un icosaedre a partir de tres plans perpendiculars.

Un cop s'ha obtingut l'icosaedre es farà una iteració en la qual, a dins de cada triangle, es generarà un triangle unint els punts centrals de les arestes del triangle exterior, de manera que apareixeran 4 triangles nous per cara.

Amb aquest mètode apareixen dos problemes, el primer és que hi ha poc control sobre el detall de l'esfera, ja que hi ha un salt molt gran del nombre de triangles en vuit iteracions, per exemple de la setena a la vuitena iteració hi ha un increment de quasi 50.000 a 250.000 triangles a tota la figura.

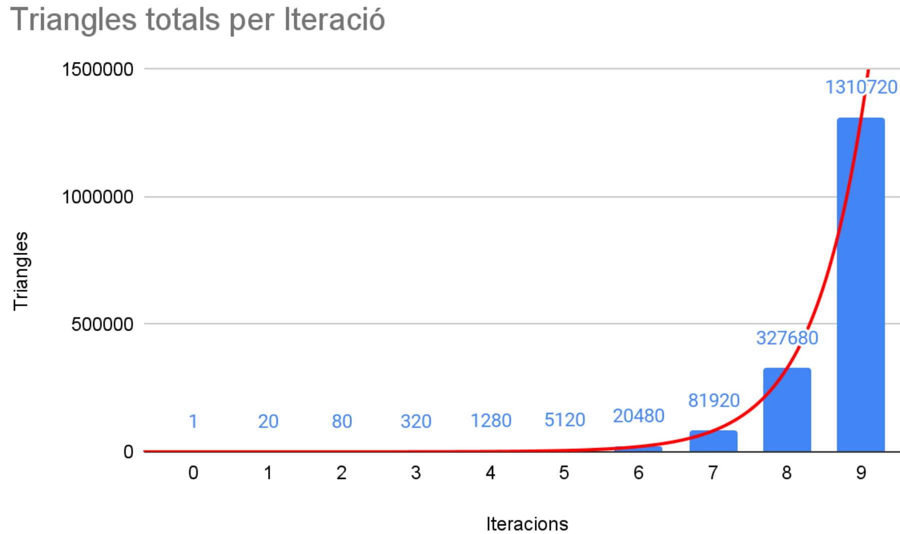


Figura 20. Gràfic que mostra el nombre de triangles per iteració.

Alhora, s'ha de tindre en compte que el càlcul dels triangles és una funció recursiva, ja que per cada un dels 20 triangles inicials de l'icosaedre se'n generen quatre, calcular 80 triangles no és un gran problema, però després aquests vuitanta es multipliquen per quatre, i els triangles resultants es tornen a multiplicar.

$$O=20*(4^n) \qquad (1)$$

2.5.4 Esfera a Partir d'un Octaedre

Aquest mètode és una millora de l'anterior, també es podria fer amb un icosaedre, però amb un octaedre es facilita molt la feina, ja que hi ha només vuit cares sobre les quals treballar en comptes de vint. En aquest cas, no es generarà un triangle a cada cara, sinó que s'indicarà quantes divisions a cada aresta es volen generar. De manera que el nombre de triangles serà el nombre de divisions al quadrat.

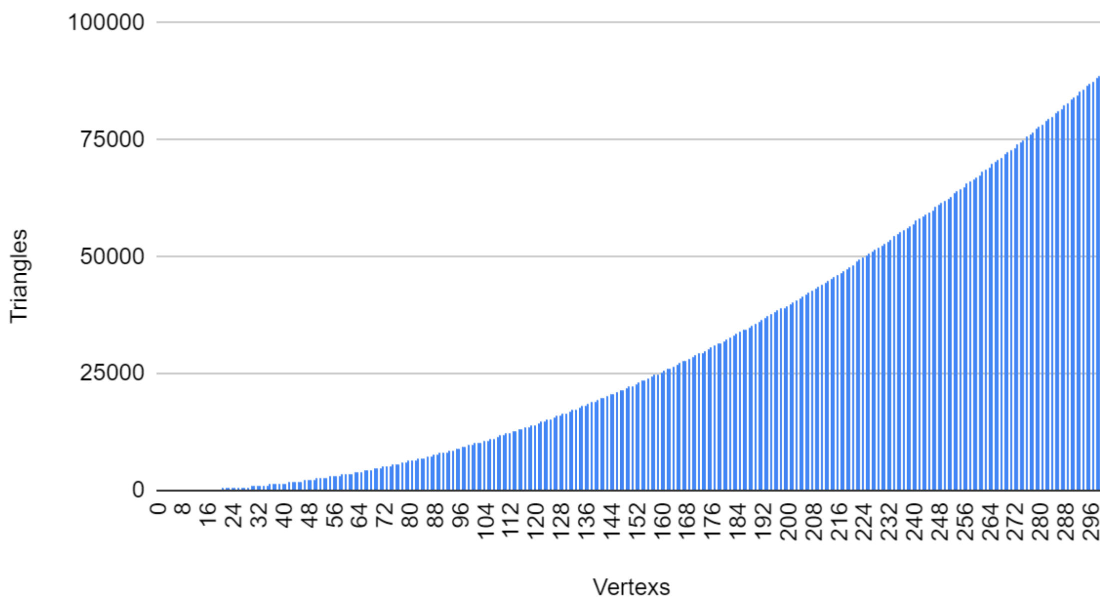


Figura 21. Gràfic que mostra el nombre de triangles per cara per nombre de vèrtex per aresta.

Com es pot observar, amb aquest mètode s'obté un augment més uniforme quan augmentem el detall de l'esfera, per una altra banda, el temps d'execució també baixa considerablement, ja que per calcular tots els triangles, es divideixen les tres arestes de la cara que estem tractant en el nombre de separacions definides per l'usuari, i tot seguit es generen un conjunt de punts ordenats a l'interior de la cara per crear triangles regulars per tota l'àrea. El nombre de triangles és causada per la successió dels quadrats dels nombres enters:

$$\text{Núm. Triangles} = \text{Núm. separacions} \wedge 2 \qquad (2)$$

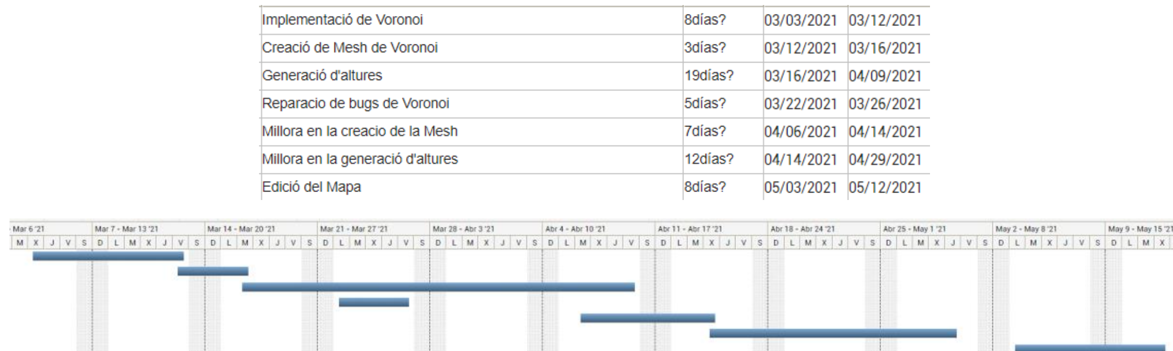
Per tant el cost computacional d'aquest algoritme vindria a ser:

$$O = 8 * N \wedge 2 \qquad (3)$$

Com a mostra de la comparació, per generar una esfera a partir de l'icosaedre l'última iteració que tenia un temps de creació raonable era la vuitena i es generaven 327.680 triangles, en canvi amb l'octaedre, el punt límit se situava a prop de les 300 divisions per costat, generant 720.000 triangles.

3. Implementació

El procés d'implementació ha estat format per dues parts principals, la primera amb el primer prototip va durar des de març fins a finals de maig. Va ser un procediment molt llarg perquè necessitava buscar molta informació de com fer el que necessitava, ja que no tenia coneixements suficients, a més, el programa, com comentaré més endavant, tardava molta estona en generar un mapa realista, cosa que em comportava una pèrdua de temps molt gran. A mitjans de maig, veient que no tenia un bon resultat i s'apropava la data d'entrega del treball vaig decidir posposar l'entrega al setembre.

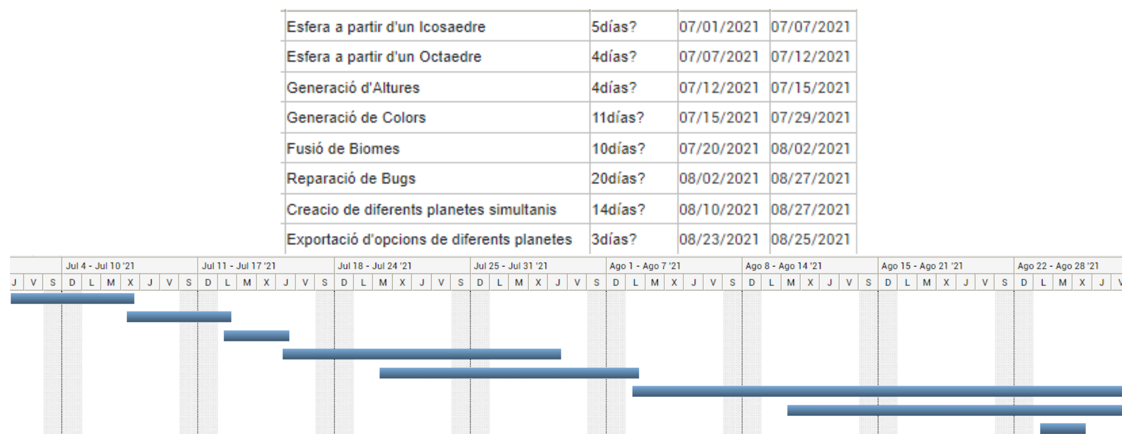


Implementació de Voronoi	8días?	03/03/2021	03/12/2021
Creació de Mesh de Voronoi	3días?	03/12/2021	03/16/2021
Generació d'altures	19días?	03/16/2021	04/09/2021
Reparacio de bugs de Voronoi	5días?	03/22/2021	03/26/2021
Millora en la creacio de la Mesh	7días?	04/06/2021	04/14/2021
Millora en la generació d'altures	12días?	04/14/2021	04/29/2021
Edició del Mapa	8días?	05/03/2021	05/12/2021

Figura 22. Diagrama de Gantt del primer prototip.

Un cop passats els exàmens de final de curs vaig tornar a posar-me amb el primer prototip, per intentar arreglar-lo, durant les dues últimes setmanes de juny vaig buscar formes de millorar tant el resultat, com l'eficiència, però no va ser possible, i em vaig decidir per buscar una alternativa.

Per tant, al juliol vaig començar el treball de zero amb un objectiu diferent, en aquest cas en comptes d'un mapa, generar un planeta, durant aquests dies vaig estar treballant la gran part del temps del qual disposava per aconseguir un bon resultat. Durant el mes de juliol vaig aconseguir tenir una generació d'un planeta satisfactori, generant muntanyes i biomes. Durant agost vaig acabar d'implementar alguna funcionalitat dels biomes junt amb una gran quantitat de temps per arreglar alguns errors, que tot i ser mínims i, al final, ser senzills d'arreglar, van comportar molt temps per trobar quin era el problema. Aquests inconvenients van vindre donats del fet de permetre generar diferents planetes simultanis.



Esfera a partir d'un Icosaedre	5días?	07/01/2021	07/07/2021
Esfera a partir d'un Octaedre	4días?	07/07/2021	07/12/2021
Generació d'Altures	4días?	07/12/2021	07/15/2021
Generació de Colors	11días?	07/15/2021	07/29/2021
Fusió de Biomes	10días?	07/20/2021	08/02/2021
Reparació de Bugs	20días?	08/02/2021	08/27/2021
Creacio de diferents planetes simultanis	14días?	08/10/2021	08/27/2021
Exportació d'opcions de diferents planetes	3días?	08/23/2021	08/25/2021

Figura 23. Diagrama de Gantt del prototip final

3.1. Primer Prototip

Tot i que aquest primer prototip no va acabar sent part del programa final, explicaré per sobre el procediment que vaig seguir per crear-lo junt amb els problemes que em va comportar i perquè vaig descartar aquesta idea.

Després de buscar informació sobre com es poden generar mapes o terreny aleatoris em vaig decantar per utilitzar un Diagrama de Voronoi, semblava una opció més atractiva, ja que, en la majoria de generacions de terreny s'utilitza Perlin.

Per tant vaig començar amb el desenvolupament d'aquest programa, un dels grans inconvenients que m'he trobat durant tot el transcurs del treball és la inexperiència amb Unity, que m'ha provocat moltes hores de buscar solucions específiques a problemes que, probablement, per a un usuari de Unity habitual, serien solucions trivials i no tindrien cap dificultat per resoldre.

El primer pas que vaig dur a terme en aquest prototip va ser aconseguir un algoritme de Voronoi amb relaxació de Lloyd per tal d'obtenir un mapa de polígons sobre el que començar a treballar.

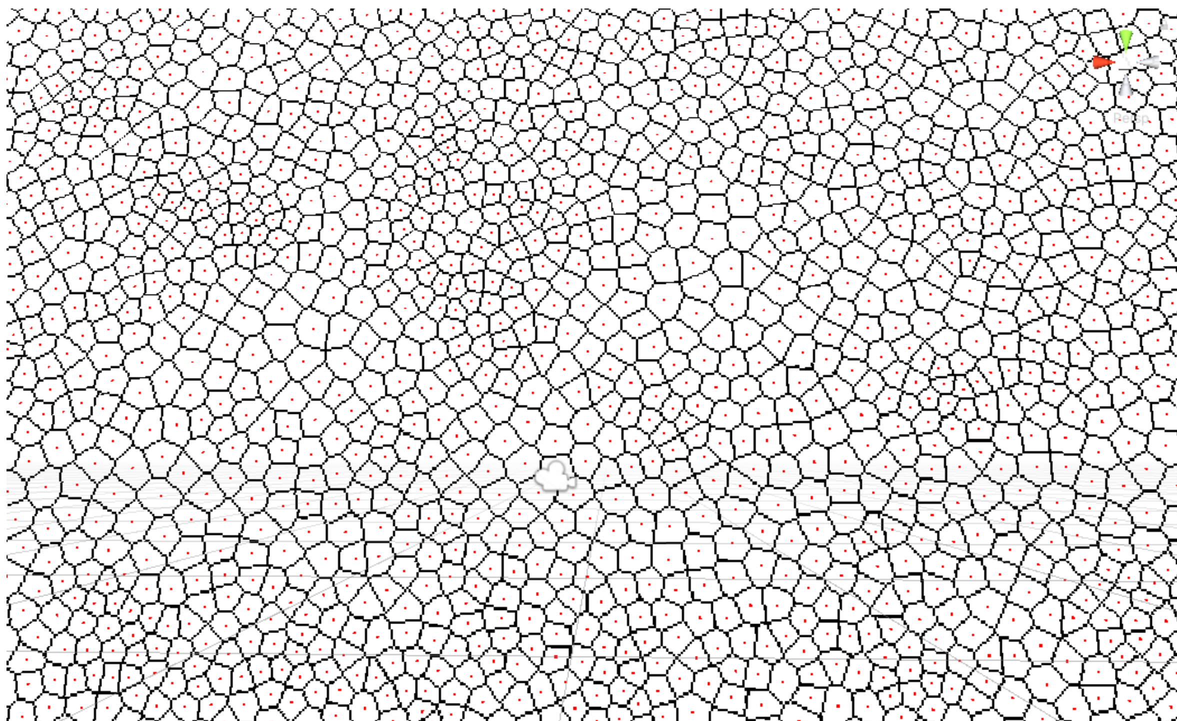


Figura 24. Fragment d'un diagrama de Voronoi de 8000 poligons.

Amb aquest algoritme, es podia introduir el nombre de polígons que volia que es generessin, entenent polígon com a cada un dels punts aleatoris que es crearien al pla. Junt amb les mides de l'objecte que es crea, en aquest cas és una imatge de 1000 píxels de costat.

L'algoritme que utilitzava, retornava una estructura de dades amb la informació de tot el diagrama de Voronoi.

voronoi	{csDelaunay.Voronoi}
Edges	Count = 2981
PlotBounds	{Rectf}
SitesIndexedByLocation	Count = 1000
edges	Count = 2981
plotBounds	{Rectf}
sites	{csDelaunay.SiteList}
sitesIndexedByLocation	Count = 1000
triangles	Count = 0
weigthDistributor	{System.Random}

Figura 25. Estructura generada per l'algoritme de Voronoi.

En aquesta estructura es podia trobar, totes les arestes del diagrama, on s’emmagatzemava també, a quins dos polígons separava aquesta aresta junt amb quins dos vèrtexs formaven la mateixa aresta, cada vèrtex emmagatzemant també la seva posició al pla.

Tot i tindre tota aquesta informació, per tal que el programa pogués crear una malla per representar tridimensionalment aquest mapa, era necessari triangular tots els polígons del diagrama. En aquest moment és quan començaven a aparèixer els problemes d’aquest prototip.

Primer de tot havia d’anar recorrent cada un dels polígons generats emmagatzemant tots els seus vèrtexs en una llista, mentre comprovava que cada vèrtex no fos ja a la llista, junt amb això, també feia una llista de tots els polígons, que contenia una llista dels índexs dels vèrtexs que formaven aquest mateix polígon. D’aquesta manera en buscar un polígon, tindria ràpidament accés a la informació dels seus vèrtexs.

Això amb l’estructura generada a partir del diagrama de Voronoi no es podia fer, ja que els vèrtexs s’emmagatzemaven sense cap relació, era simplement la informació de la seva posició, i necessitava tindre tots els vèrtexs en ordre per crear la mesh.

Quan s’havia aconseguit tota aquesta informació era el moment de començar a crear els triangles per a cada polígon. La forma en la qual s’aconseguia això era, agafant de tres en tres vèrtexs per formar un triangle, i així fins a triangular tota la figura.

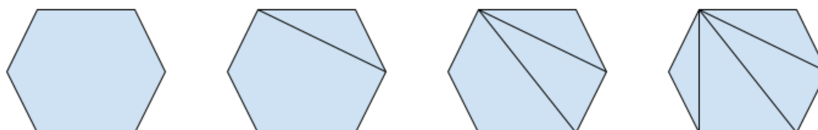


Figura 26. Triangulació d'un polígon.

Però aquí va aparèixer un problema, quan agafava tots els vèrtexs de cada un dels polígons, tal com es genera el diagrama de Voronoi, els vèrtexs no estaven ordenats, per tant es generaven formes no coherents.

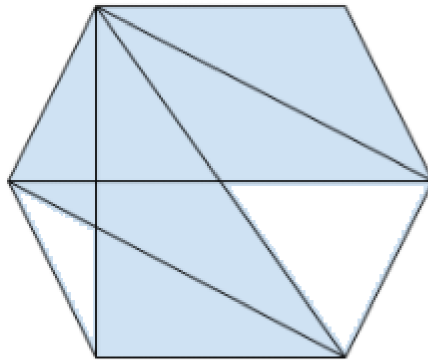


Figura 27. Triangulació incorrecta d'un polígon.

Per ordenar-los vaig utilitzar un algoritme anomenat *Gift Wrapping*, que consisteix a rodejar un conjunt de punts, de manera que es crea una àrea formada pels punts més externs.

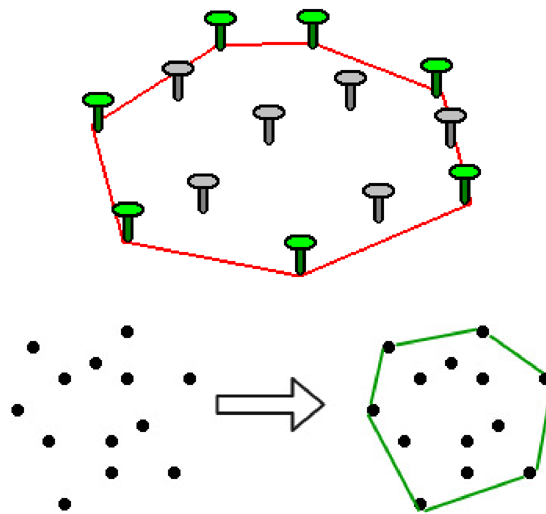


Figura 28. Algoritme de Gift Wraping.

Aquest algoritme analitza tots els punts d'un conjunt per conèixer el recorregut necessari per tindre tots els punts encerclats per una línia. El primer que es fa és, trobar el punt més a l'esquerra de tots els punts i a partir d'allí busca el següent punt més a l'exterior, un cop trobat, repeteix el procés des del nou punt que s'ha trobat i això es repeteix fins a tornar al punt inicial.

En aquest cas, no hi havia vèrtexs interns, però aquest algoritme servirà per obtenir l'ordre en què els vèrtexs estan situats i per tant, es podria generar correctament la triangulació dels polígons.

Un cop fet tot això ja es pot generar la mesh i s'obtindrà el següent resultat.

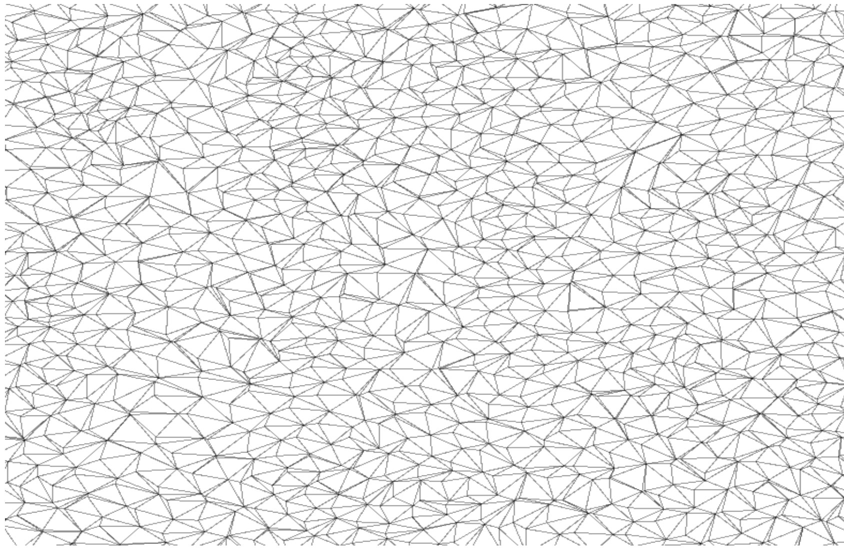


Figura 29. Diagrama de Voronoi amb els polígons triangulats.

Si analitzem tots els càlculs que s'han fet per obtenir aquesta imatge, es pot veure com el cost computacional del programa ha augmentat molt. S'han de recórrer tots els vèrtexs de tots els polígons, i després dins de cada grup de vèrtexs calcular el seu ordre i per últim calcular els triangles de cada un dels polígons. I encara queda un últim pas per poder començar a generar altures.

Per tal de generar les altures a la mesh s'ha de donar un valor a cada un dels vèrtexs depenent dels vèrtexs del seu voltant, per tal de relacionar tots els vèrtexs l'últim pas és crear una relació entre tots ells. Aquest procés també consumia una gran quantitat de temps, ja que s'havia de tornar a recórrer cada un dels polígons. Dins de cada polígon ja teníem ordenats els vèrtexs, per tant s'agafava un vèrtex de cada polígon i s'afegia a una nova llista on es relacionava amb els vèrtexs que tenia al seu voltant i així amb tots els vèrtexs del polígon. Això es repetia per cada polígon, si un vèrtex ja es trobava a la llista, només se li havia d'afegir els nous veïns del polígon actual.

A partir d'aquí quedaria donar una altura a un dels vèrtexs aleatòriament i amb un senzill algoritme que vaig crear, s'anava donant una altura menor als vèrtexs veïns, aquesta altura variava sobre l'altura del vèrtex pare, com més alt el vèrtex, el decrement era més gran. I aquest és el resultat.

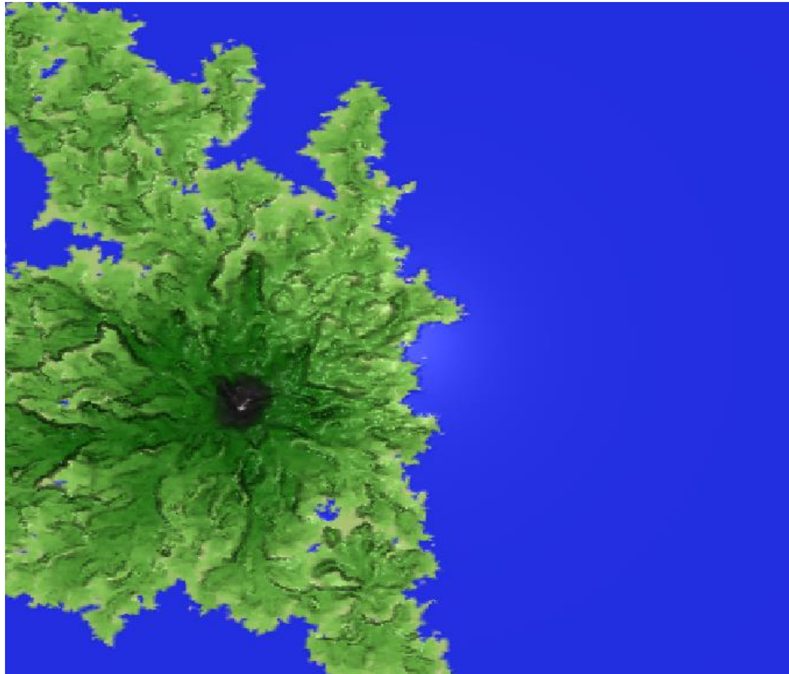


Figura 30. Resultat del primer prototip amb 30000 polígons.

No és un resultat dolent a simple vista, es generen formes de terreny interessants, però té uns quants problemes.

Si miràvem la imatge en tres dimensions, es veien diferències d'altura estranyes i poc realistes, un altre gran problema era el temps perquè es generés un mapa, aquest per exemple constava de 30.000 polígons i tardava a prop de mitja hora en generar-se. La idea principal del projecte era que hi hagués una possible interacció amb l'usuari per poder modificar valors i adaptar com volgués el terreny, cosa que amb aquest prototip no era viable.

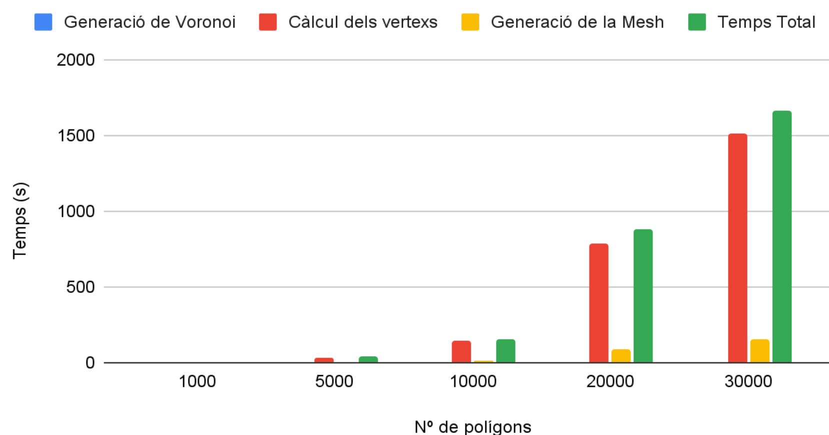


Figura 31. Gràfic que mostra el temps de generació de cada part de l'algoritme respecte al nombre de polígons seleccionats.

En aquest gràfic es mostra com varia el temps d'execució depenent del nombre de polígons que volem generar, com es pot observar, la part que té un cost computacional més alt, són els càlculs dels vèrtexs, obtenir els vèrtexs de cada polígon, ordenar aquests vèrtexs, trobar els veïns... I per obtenir un resultat realista, s'ha d'executar el programa per 15.000 polígons aproximadament.

El problema no era el fet que tardes molt, sinó que per aquesta mateixa raó, cada cop que feia canvis i volia veure el resultat d'aquests canvis, havia d'esperar una gran quantitat de temps i no era viable. Per altra banda, hi havia poca personalització, vaig crear un script que permetia modificar el terreny amb un pinzell, però no em va semblar suficient. Per aquestes raons vaig decidir canviar de mètode.

Després de descartar el primer prototip vaig decidir buscar altres opcions. Primer de tot vaig decidir generar les altures amb mapes de soroll, ja que, al cap i a la fi és la forma més utilitzada generalment, i la raó és la senzillesa i els bons resultats que dona aquest mètode. Alhora, vaig decidir que, en comptes de generar un mapa pla, fer una esfera per representar un planeta.

3.2 Creació de l'Esfera

Els processos necessaris per poder generar la mesh que formarà l'esfera del planeta estaran situats al fitxer *PlanetMesh.cs.* Aquests processos són, el càlcul de la posició de tots els vèrtexs, la triangulació de les cares i els càlculs de les UVs.

Dins aquest fitxer es poden trobar dues classes que s'utilitzaran per guardar informació mentre no es creï l'esfera:

Una classe *Edge,* que s'utilitzarà per guardar una llista de tots els vèrtexs situats a les arestes de l'octaedre.

Una classe *VertexList*, que contindrà una llista de tots els vèrtex que formen l'esfera.

En aquest fitxer cal destacar tres estructures que són necessàries per generar l'octaedre:

```
static readonly Vector3[] octVert = { Vector3.up, Vector3.left,
      Vector3.back , Vector3.right , Vector3.forward , Vector3.down };

static readonly int[] octEdges = { 0, 1, 0, 2, 0, 3, 0, 4, 1, 2, 2,  3,
      3, 4, 4, 1, 5, 1, 5, 2, 5, 3, 5, 4 };

static readonly int[] octTriangles = { 0, 1, 4, 1, 2, 5, 2, 3, 6, 3, 0,
      7, 8, 9, 4, 9, 10, 5, 10, 11, 6, 11, 8, 7 };
```

Codi 1. Estructura base d'un octaedre.

El primer vector està format per *Vector3*, que és una estructura de Unity que és en si un vector de tres posicions, utilitzat per establir informació de punts i vectors en un espai tridimensional. Conte les sis posicions que necessiten els vèrtexs per formar l'octaedre, Aquestes posicions són unitàries des del punt 0,0 de l'espai en el qual estem treballant, de manera que els valors tindran una coordenada en l'espai amb valor 1 i dues a la posició 0.

El segon vector d'enters fa referència al primer, on s'està indicant quins vèrtexs formen cada un dels dotze costats que formen l'octaedre. Cada dos valors del vector formen un costat.

El tercer fa referència al segon vector, aquest indica de tres en tres, quins dels costats del vector anterior formen una cara de l'octaedre.

A continuació es començarà a explicar les diferents funcions que formen aquesta classe, començarem parlant del constructor.

Al constructor s'establiran els valors de les variables necessàries junt amb el càlcul del número de vèrtex que es generaran per cara i el numero total de vèrtexs de tot l'icosaedre.

Per calcular el nombre de vèrtexs per cara vaig haver de buscar la forma de fer un càlcul per un nombre *n* de vèrtexs. Manualment vaig començar a calcular quants vèrtexs es generaven depenent del nombre de divisions en una sola aresta i vaig obtenir els següents resultats:

0 divisions → 3 vèrtexs

1 divisió → 6 vèrtexs

2 divisions → 10 vèrtexs

3 divisions → 16 vèrtexs

Aquesta successió és anomenada *Llei de formació de nombres triangulars* i és generada a partir d'aquesta fórmula:

$$\text{nº vèrtexs per cara} = (n * (n + 1)) / 2 \qquad (4)$$

Sent *n* el nombre de vèrtexs per aresta, amb aquesta informació es pot calcular el nombre total de vèrtex que hi haurà a tot l'icosaedre per tal de crear la llista on guardar la informació de cada un d'ells.

$$\text{nº total de vèrtexs} = \text{nº vèrtexs per cara} * 8 - (n) * 12 + 6; \qquad (5)$$

Aquesta fórmula sorgeix a partir de què, si tenim X vèrtex per cara, es poden multiplicar pel nombre de cares que tenim, amb aquest càlcul tenim duplicats tots els vèrtexs de les arestes, per eliminar aquesta duplicitat, multipliquem *n*, el nombre de vèrtexs per aresta, pel nombre d'arestes, és a dir dotze, i ho restem al resultat anterior, per últim sumem sis, que són els vèrtexs originals de l'octaedre.

Un cop tenim la informació necessària començarem a crear la mesh. Primer de tot, es genera un vector amb una mida igual al nombre de vèrtexs totals, on guardarem la posició de tots els vèrtexs perquè la mesh pugui representar-los. S'afegiran els sis vèrtexs inicials de l'octaedre i a partir d'aquí es començarà a crear la resta de vèrtexs.

```
Edge[] edges = new Edge[12];
for (int i = 0; i < octEdges.Length; i += 2)
{
    Vector3 startVertex = vertexList.items[octEdges[i]];
    Vector3 endVertex = vertexList.items[octEdges[i + 1]];

    int[] edgeVertexIndices = new int[resolution+1];
```

```
        edgeVertexIndices[0] = octEdges[i];

        for (int divisionIndex = 0; divisionIndex < resolution -
             1; divisionIndex++)
        {
             float t = (divisionIndex + 1f) / (resolution);
             edgeVertexIndices[divisionIndex + 1] =
                   vertexList.nextIndex;
             vertexList.Add(Slerp(startVertex, endVertex, t));
        }
        edgeVertexIndices[resolution] = octEdges[i + 1];
        int edgeIndex = i / 2;
        edges[edgeIndex] = new Edge(edgeVertexIndices);
}
```

Codi 2. Creació dels vèrtexs a les artistes del octaedre.

Primer de tot es crea un vector de *Edges* on guardarem els costats amb tots els vèrtexs nous que es generaran. Es recorrerà el vector *octEdges* on hi ha la informació dels vèrtexs que formen cada aresta. Aquests dos punts es guarden a les variables *startVertex* i *endVertex*.

A continuació es crea un vector de dimensions *resolution+1* on la resolució vindria a ser el nombre d'arestes final que hi haurà a cada aresta de l'octaedre, per tant sempre hi haurà un vèrtex més que arestes, després s'afegeix el primer vèrtex al vector.

Tot seguit es generaran tots els vèrtexs de l'interior de l'aresta. Primer s'obtindrà el valor entre 0 i 1 on es col·locarà el nou vèrtex, guardem l'índex a la llista de l'aresta que rebrà aquest vèrtex en afegir-lo a *vertexList*, i per últim s'afegirà a la llista de vèrtexs el punt calculat amb la funció *Slerp,* que interpola entre dos punts, *startVertex* i *endVertex*, un punt a una distància t.

Es repetirà aquest procés fins a obtenir tots els vèrtexs de l'aresta, un cop fet això, s'afegirà l'últim punt a la llista de vèrtexs, i tot aquest vector es guardarà a la llista d'arestes de l'octaedre.

Tot seguit, per cada aresta, es cridarà a la funció *CreateMesh* on es farà el càlcul dels vèrtexs interiors i la triangulació de tots aquests vèrtexs. El procés és semblant, aquest cop en comptes de generar punts a les arestes, anirem agafant els vèrtexs de dalt a baix del triangle i generant vèrtexs entre dos punts que estan a la mateixa altura a costats oposats, tot seguit es fa la triangulació d'aquells vèrtexs i se salta a la següent fila de vèrtexs.

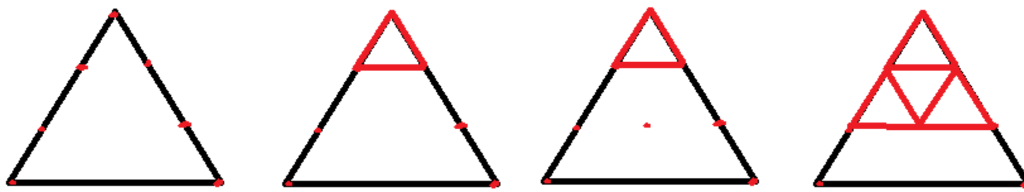


Figura 32. Procés de com es crea la triangulació de les cares de l'octaedre.

En aquest fitxer hi ha una altra funció anomenada *CreateUvsMesh* i *getUVs* que s'explicaran més endavant.

3.3 Generació d'Altures

Per generar les altures de l'esfera el que s'ha de fer és, calcular amb algoritmes de soroll l'altura de cada un dels vèrtexs de l'esfera que hem creat a l'apartat anterior. Per dur això a terme s'utilitzen un total de quatre fitxers, dos per a la configuració de l'usuari i els altres dos per al càlcul d'aquestes altures.

El primer fitxer de configuració és anomenat *NoiseSettings* i com el seu propi nom indica, conte les opcions per modificar la generació de soroll, aquests valors han sigut comentats en l'apartat 2.2.4 on s'explicava el funcionament del Fractal Noise. Aquests valors són la lacunaritat, la freqüència, els octaus i la persistència. A part d'aquests tenim uns altres modificadors addicionals:

- Multiplicador: Permet multiplicar tots els punts resultants d'usar els algoritmes de soroll per un valor introduït per l'usuari, per augmentar les dimensions del resultat.
- Valor mínim: Aquest paràmetre limitarà el valor més baix que pot retornar l'algoritme de soroll, en cas que el valor sigui menor a l'introduït per l'usuari, el punt tractat rebrà una altura igual al radi base del planeta.
- Valor màxim: Per tal de controlar l'altura màxima que es pot generar, utilitzarem aquest valor, no es limitarà l'altura màxima sinó que s'afegirà factor reductor a tots als punts.
- Desplaçament de Perlin: Aquest paràmetre consta de tres entrades, aquestes entrades són les direccions d'un espai tridimensional, X, Y i Z, modificant aquests valors, l'algoritme retorna diferents resultats per tant es generen masses de terreny diferents.

L'altre fitxer de configuració, *HeighSettings* conté opcions més generals envers l'altura, com pot ser el radi del planeta comentat anteriorment, que servirà per fer més gran o més petit el planeta generat, alhora, hi ha una llista de *NoiseLayer*, que és una classe, on tenim quatre paràmetres més:

- Aplicat: Aquest és un valor booleà per indicar si aquesta capa ha de ser representada al planeta o no.
- Màscara: En cas que tinguem més d'una capa de soroll, voldrem que les capes superiors funcionin respecte a les capes que tenen per sota, aquesta opció ens permet habilitar això.
- RidgeNoise: Amb aquest altre valor booleà, podrem indicar si s'utilitzarà l'algoritme de Perlin o el de Ridge.

- Opcions de capa: Aquesta variable fa referència a la classe que conté les opcions de l'algoritme de soroll del fitxer explicat anteriorment.

Per tant, l'usuari introduirà el nombre de capes de soroll i apareixeran aquella quantitat d'opcions per modificar cada una de les capes.

A continuació ens fixarem en els altres dos fitxers, aquests serviran per calcular l'altura dels punts de l'esfera, primer parlarem del fitxer *NoiseCalculator*. Aquest fitxer s'encarrega dels càlculs de cada un dels punts individuals, tenim dues funcions, una per calcular un punt a partir del Perlin Noise i un altre per al Ridge Noise, donat que els dos són pràcticament iguals només analitzarem l'algoritme de Perlin.

Aquesta funció rebrà un punt en un espai tridimensional i retornarà un únic valor que representarà l'altura d'aquest punt.

```
public float getNoiseValue(Vector3 point)
{
    float noiseValue = 0;
    float frequency = noiseSettings.frecuencia;
    float amplitude = 1;

    for (int i = 0; i < noiseSettings.octavos; i++)
    {
        float v = noise.Evaluate(point * frequency +
            noiseSettings.desplazamientoDePerlin);
        noiseValue += (v + 1) * 0.5f * amplitude;
        frequency *= noiseSettings.lagunaridad;
        amplitude *= noiseSettings.persistencia;
     }

    noiseValue = Mathf.Max(0, noiseValue - noiseSettings.valorMinimo);
    return noiseValue *
          noiseSettings.multiplicador*noiseSettings.valorMaximo;
}
```

Codi 3. Càlcul del valor del algoritme de soroll.

Aquí podem veure en forma de codi el que s'ha explicat anteriorment a l'apartat del Fractal Noise. Els octaus definiran els cops que s'executa el bucle on, primer de tot, es crida a la funció *Evaluate*, que és la funció que conté el Perlin Noise, li passem el punt amb les modificacions de la posició del desplaçament comentada anteriorment a les opcions, i la freqüència, que recordem, serveix per generar canvis de nivell en un espai més petit com si es fes una disminució de zoom.

Tot seguit, se suma el valor obtingut al resultat final amb la modificació de l'amplitud i a continuació es modifiquen la freqüència i l'amplitud, amb la lacunaritat i la persistència respectivament.

Un cop s'ha fet el càlcul de tots els octaus, limitarem l'altura mínima amb la funció *Mathf.Max.* Que retorna el valor més gran dels dos introduïts, com el valor que retorna la funció *getNoiseValue* se sumarà al punt de l'esfera, s'introdueix l'altura restada amb l'altura mínima, que en cas que doni un valor negatiu retornarà zero.

Un cop s'ha delimitat el valor resultant, se li afegeix el multiplicador i la modificació d'altura màxima.

La funció de Ridge Noise funciona exactament igual excepte després de cridar a la funció *Evaluate* es restarà a 1 el valor absolut del resultat i tot seguit es calcularà el quadrat del valor.

L'altre fitxer que forma part de la generació d'altures és el fitxer *HeightGenerator*, aquest fitxer té una única funció que rep el vector de tots els punts del planeta i va iterant per cada un dels valors del vector, calculant l'altura de cada punt, dins aquest bucle, n'hi ha un altre que s'itera tants cops com capes de so tenim. Hi ha una variable *altura* que a cada punt nou que es calculi, estarà inicialitzada a zero i se li sumarà el resultat de cada capa. Cal destacar que, aquí és on s'aplica l'opció de la màscara, en cas que una capa tingui l'opció de màscara aplicada, el resultat d'utilitzar l'algoritme de soroll serà multiplicat per l'altura del mateix punt a la capa anterior. D'aquesta manera, els resultats de la capa actual depenen de la capa anterior, per exemple, si generem una primera capa que és la base del planeta i una segona on posem muntanyes, les muntanyes apareixeran a les zones més altes de la capa anterior, ja que és on tindran major altura i per tant en aplicar la màscara a la segona capa, podran obtenir una major altura que si, per exemple, es volgués generar una muntanya a la costa, on l'altura base és més baixa.

3.4 Generació de Colors

Per generar els colors de cada planeta, tenim una estructura pràcticament idèntica a la de generar altures. Hi ha quatre fitxers, dos de configuració, *ColorNoiseSettings* i *ColorSettings.* El primer fitxer d'aquests dos, és igual a *NoiseSettings* del apartat anterior, conté els mateixos paràmetres que en els càlculs de l'altura excepte la persistència, ja que en aquest cas l'amplitud del soroll no té cap efecte, i tampoc hi ha les opcions d'altura màxima i mínima, però en canvi hi ha:

- Desplaçament: Permet moure els biomes en l'eix vertical, ja que depenent de la força aplicada tots els biomes pateixen un desplaçament, per tant amb aquesta variable podem reajustar els colors.
- Força: Amb aquest paràmetre es dona intensitat al resultat del soroll, per tant el desplaçament del bioma serà més gran.

L'altre fitxer de configuració, *ColorSettings* permet escollir la quantitat de biomes que hi ha al planeta, en els quals es podrà escollir l'altura en l'eix vertical en què volem que comenci i acabi el bioma, junt amb els colors d'aquest. Per una altra banda, també permet escollir la força de la fusió dels diferents biomes.

El tercer fitxer del qual parlarem serà *ColorNoiseCalculator*, aquest fitxer calcula el valor del soroll per un cert punt, igual que en l'apartat anterior, obté el soroll en un punt de l'espai utilitzant la funció *Evaluate* i se li apliquen els octaus, freqüència i lacunaritat.

Ara bé, abans de parlar de l'últim fitxer, explicaré com es dona color al planeta.

Per tal de poder donar color a diferents planetes sense haver de generar una textura per cada un d'ells utilitzarem un shader, els shaders, són un tipus de scripts que permeten modificar com es veu un material depenent d'un conjunt de dades, per exemple a partir d'informació dels vèrtexs d'un objecte i textura, amb aquestes dades el shader donarà color a cada planeta depenent dels colors introduïts per l'usuari i les altures que ha generat el programa.

Per tal que el nostre shader funcioni correctament necessitarem tres coses. Primer de tot, indicarem el bioma en el qual es troba cada un dels punts de l'esfera i establirem el valor a la variable de UVs de la mesh. Aquesta variable té un altre us normalment, però en aquest cas s'utilitzarà per emmagatzemar informació de cada punt. En segon lloc és necessari tindre constància de l'altura màxima i mínima d'entre tots els vèrtexs del planeta, aquests dos valors es guarden quan es genera l'altura de tots els punts. Per últim necessitem una textura amb els colors de cada bioma.

Començarem explicant com es creen els UVs de l'esfera. Per fer això utilitzem dues funcions situades al fitxer *PlanetMesh*, aquestes dues funcions són *getUvs* i *CreateUvsMesh,* són pràcticament idèntiques a les funcions per generar el planeta, la primera, *getUvs* calcula tots els punts de l'esfera amb el mateix procediment, es posen punts a les arestes i després al centre de cada cara, hi guardem el bioma en el qual es troba cridant a la funció *whichBiomeIsThis*, la funció *CreateUvsMesh* fa el càlcul dels vèrtexs interiors de cada cara. Aquest vector ha de tindre la informació dels biomes en el mateix ordre que la llista de vèrtex de l'esfera i d'aquesta manera la mesh ja sap quin UV pertany a cada vèrtex.

Es pot arribar a pensar que per no duplicar codi es podria cridar aquesta funció quan generem el planeta, però el problema és que en tenir dues parts al codi, una de generació d'altura i una altra de colors, es modifiquen per separat, és a dir, si canviem configuració dels colors, la part d'altura no es tornarà a executar, per tant, hem de separar el càlcul dels biomes del càlcul de les altures.

Per últim explicaré el fitxer *ColorManager*, aquest fitxer conte la creació de la textura a partir dels colors que ha introduït l'usuari i també el càlcul de quin bioma pertany a cada posició del planeta, junt amb el control de les altures dels biomes, per tal que no se superposin entre ells.

Primer explicaré com es genera la textura a partir dels colors. L'usuari pot introduir el nombre de biomes que vol, i per cada un d'aquests biomes podrà escollir una paleta de colors.

```
public void setColors()
{
    Color[] colors = new Color[detail * settings.biomes.biome.Length];
    int colorCounter = 0;
    UpdateColour(settings);
```

```
    foreach (var b in settings.biomes.biome)
    {
        Gradient biomeGradient = b.planetColour;
        for (int i = 0; i < detail; i++)
        {
            colors[colorCounter++] = biomeGradient.Evaluate(i /
                                           (detail + 1f));
        }
    }
    planetTexture.SetPixels(colors);
    planetTexture.Apply();
    settings.planetMaterial.SetTexture("_planetTexture",
                                                  planetTexture);
}
```

Codi 4. Creació de la textura d'un planeta.

Amb la funció *setColors* crearem un vector de dimensions iguals al detall del color, que és un valor fix, en aquest cas 64, multiplicat pel nombre de biomes. Un cop generat aquest vector, es divideix el vector que conté el color dels biomes en una quantitat de parts igual al detall de la textura, i es va introduint cada porció al vector de colors que s'ha generat, això es repetirà per cada bioma. Un cop el vector està complet, es transformarà en textura i s'aplicarà al material del planeta.

Bé, a continuació comentaré com funciona la funció whichBiomeIsThis. Aquesta funció rep un punt en l'espai i es calcularà a quin bioma pertany, és aquí on s'aplica l'algoritme de soroll per als colors. Primer de tot s'obté l'altura en l'eix y del punt rebut i es transforma en un valor de 0 a 1.

```
float pointHeight= (punt.y + 1) / 2f;
pointHeight += (calcularNoise(punt.y) - desplaçament * força
```

Codi 5. Obtenció de l'altura d'un punt en l'esfera.

Amb això s'aconsegueix que un punt rebi un valor que modifica l'altura sobre la qual es calcularà el bioma, per tant un punt que estaria a un bioma en concret, en calcular el valor del soroll i afegir-li aquest resultat, passaria a estar a un bioma diferent i per tant tindrà un color diferent.

Ara ens fixarem en com es calcula quin bioma pertany al punt i com es fa la suavització entre biomes.

```
for (int i = 0; i < settings.biomes.biome.Length; i++)
{
      var biome = settings.biomes.biome[i];
      if (pointHeight >= biome.alturaMinimaBioma && pointHeight <=
            biome.alturaMaximaBioma)
```

```
        {
                float biomeCenter;
                if (i == 0)
                {
                        biomeCenter = 0;
                }
                else if (i == settings.biomes.biome.Length - 1)
                {
                        biomeCenter = 1;
                }
                else {
                        biomeCenter = (biome.alturaMaximaBioma +
                                biome.alturaMinimaBioma) / 2;
                }
                float distance = pointHeight - biomeCenter;
                offset = Mathf.Min(0.5f, distance * blend);
                offset = Mathf.Max(-0.5f, offset);
                counter += offset;
                result = (1+counter * 2) / (settings.biomes.biome.Length*2);
        }
        counter++;
        if(result != 0)
        {
                break;
        }
}
return result;
```

Codi 6. Càlcul del color d'un punt de l'esfera.

En aquest bucle es van recorrent els diferents biomes, i comprovem si el punt en el qual ens trobem està entre l'altura mínima i màxima del bioma. En cas afirmatiu es busca el centre del bioma, si el bioma és, o bé el que està a la part inferior, o a la superior, s'establirà el centre com el punt 0 o 1 de l'esfera respectivament, si no, es farà el càlcul del centre sumant els límits i dividint entre dos. Amb aquest valor calcularem com de lluny és el punt que estem analitzant del centre i a continuació controlarem que aquest valor multiplicat pel factor de suavitzat dels biomes estigui entre 0.5 i -0.5.

Això és degut a com es calcula el bioma de cada punt, recordem que hem generat una textura amb tots els colors dels biomes. Doncs per indicar el bioma s'ha de calcular quin color de la textura ha de rebre, aquest color es calcula amb l'operació:

(1+ índex del bioma actual * 2) / (nº de biomes * 2) (6)

Aquesta operació ens donarà un valor d'entre 0 i 1 que correspondrà a la posició de la textura d'on més endavant s'agafarà el color per pintar el planeta.

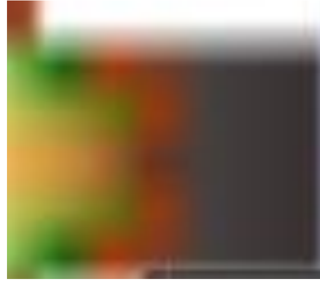


Figura 33. Textura d'un planeta.

Com es pot observar, els colors de cada bioma es van ajuntant amb els colors dels seus costats, per tant, l'operació comentada anteriorment ens retornarà la posició on apareixen els colors reals de cada bioma. Es podria pensar que, seria lògic que la funció fos:

Índex del bioma / nº de biomes (7)

Però això no és així pel fet que no tots els biomes dins la textura tenen la mateixa longitud. Tant el primer com l'últim bioma són més grans que els altres per tant tots els colors estan desplaçats.

Tornant al perquè del control del valor obtingut de suavitzat entre 0.5 i -0.5. Com es veu a la fórmula per calcular el color del bioma, l'índex sempre és un valor enter, per tant, el punt on dos colors adjacents estan més barrejats serà a la meitat d'aquests, és a dir, a 0.5 punts de distància.

Un cop obtingut l'índex del bioma junt amb el desplaçament del suavitzat s'utilitza l'operació per calcular la posició de la textura i es retorna aquest valor.

Ara, el shader ja té tota la informació necessària per donar color al planeta depenent de l'altura i el bioma de cada un dels seus punts, però s'ha d'indicar quin color ha de rebre cada vèrtex, per això s'utilitza una interfície especial dels shaders, on, a partir d'un conjunt de nodes, es poden fer operacions per indicar quin color s'ha de representar.

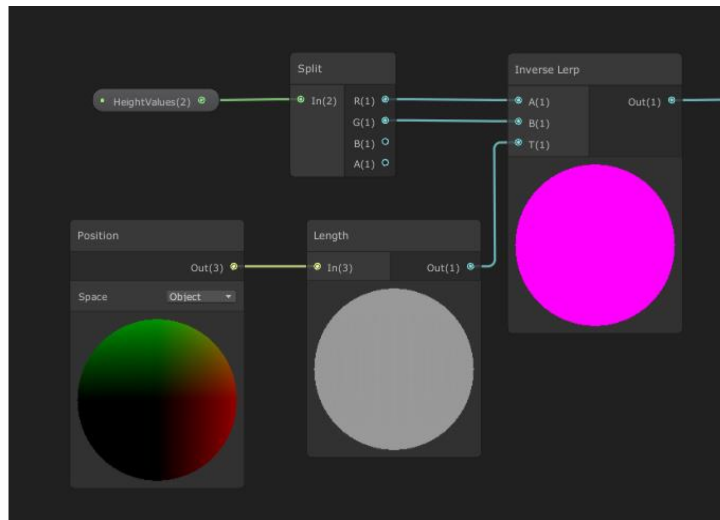


Figura 34. Interfície del shader, primera part.

En aquesta imatge es calcula l'altura percentual de cada punt del planeta. Cada un d'aquests requadres és anomenat node, i cada node conte una informació o permet fer operacions, els nodes poden tindre inputs o outputs per operar o donar informació a altres nodes.

En aquest cas, a dalt a l'esquerra tenim el primer node anomenat *HeightValues*, aquest valor està enllaçat amb el codi anteriorment vist, i conte el punt més alt i més baix d'un planeta, per tant, conte dos valors, utilitzarem un node anomenat *Split*, que ens permet dividir informació en més d'un valor, en aquest cas tindrem dues altures.

Ara ens dirigim a la part interior, el primer node anomenat *Position*, té la informació de cada punt de l'esfera, per tant, el seu output és cada un dels punts i aquesta informació ens servirà per calcular la longitud del vector des del centre de l'esfera.

Un cop tenim aquest valor utilitzarem la funció *InverseLerp,* aquesta funció, rep tres valors, i retorna un sol valor entre 0 i 1. Els dos primers paràmetres indiquen el rang en el qual es faran els càlculs, i el tercer valor és el valor que acabarà transformat en un valor entre 0 i 1, depenent de la distància a la qual es trobi del primer o el segon valor respectivament.

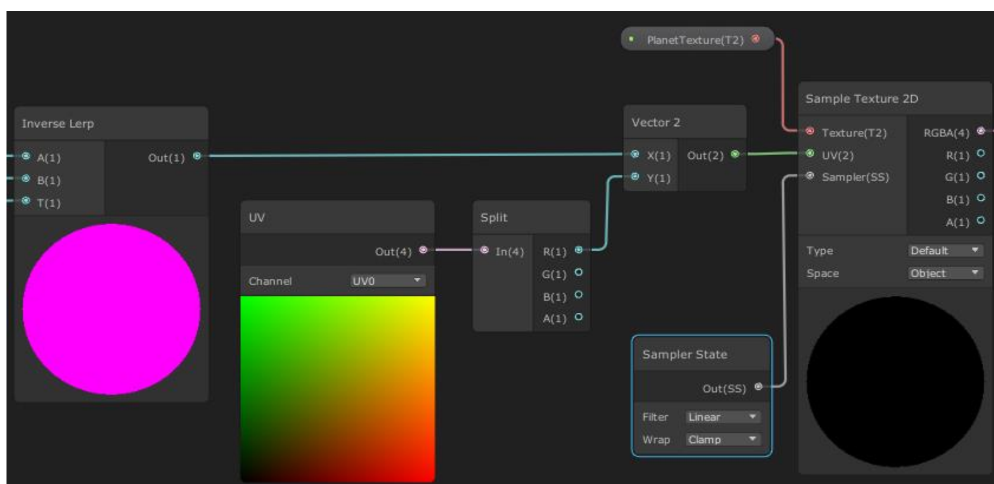


Figura 35. Interfície del shader, segona part.

Un cop ja s'ha obtingut el valor entre 0 i 1, s'ha de calcular el color d'aquest vèrtex.

Tenim el node *UV*, que donarà el valor de la UV de cada vèrtex de la mesh on hem emmagatzemat el bioma en el qual es troba, aquest valor tot i ser un vector de quatre posicions, només té informació en la primera posició, per tant amb el node *Split* agafem aquest únic valor i s'uneix amb el valor resultant del *InverseLerp.*

El node que hi ha a la part superior, anomenat *PlanetTexture*, conte la textura calculada anteriorment on hi ha els colors de tots els biomes, aquesta textura s'enllaça a la funció *SetColors.*

Per tal d'acabar donant el valor correcte a cada punt de l'esfera necessitem un node anomenat *Sample Texture 2D*, aquest node, a partir d'una textura i una posició, calcula el color que rebrà cada vèrtex. A aquest *Sampler* se li ha d'indicar la configuració amb el node *Sampler State*, que té dues variables, *Filter,* que indica com es representaran els píxels al material, en aquest cas, *linear* indica que cada píxel podrà ser el resultat de la mitjana de dos colors diferents, i l'altre variable *Wrap*, indica com es representarà un punt en cas que es trobi fora de la textura, en aquest cas, se li donarà el color més a la vora de la textura.

3.5 Interfície

A continuació s'explicarà com és la interfície per generar el planeta, en primer lloc les opcions generals. En aquesta primera part es pot escollir la resolució del planeta, és a dir, el nombre de polígons generats. El nombre mostrat és el nombre de separacions que es generen a les arestes de l'octaedre original, aquest valor pot anar des d'1 fins a 300, formant un total de 8 a 720.000 costats respectivament.

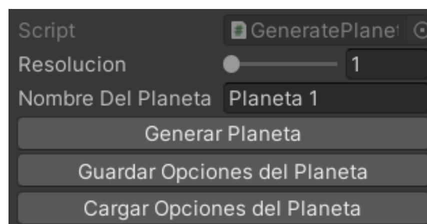


Figura 36. Interfície de generació del planeta.

Tot seguit hi ha el nom que rebrà el planeta al ser generat, que també servirà per guardar o importar les seves opcions amb els botons que es poden veure a la part inferior, junt amb el botó *Generar Planeta*, que executarà tant els algoritmes de generació d'altura com els de la generació de color.

A continuació es troben les opcions d'altura del planeta, aquí es poden trobar totes les variables per modificar la generació del desnivell.

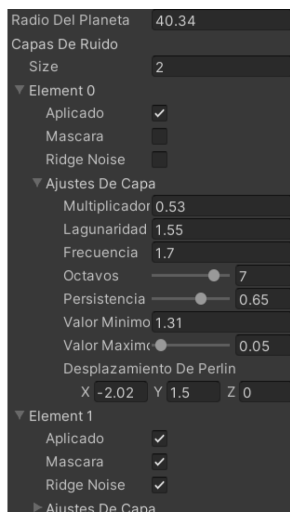


Figura 37. Interfície per modificar els valors de generació d'altura.

El primer valor que és pot veure es al radi del planeta, que permet augmentar o disminuir la mida del planeta, tot seguit es poden escollir el nombre de capes de so que es poden afegir, depenent del valor introduït, apareixeran més o menys conjunts d'opció de cada capa.

En aquest exemple s'han escollit dues capes, dins de cada capa hi ha tres opcions que es poden activar. La primera, *Aplicado*, indica si s'ha de generar o no aquesta capa amb les seves opcions, la segona, *Mascara*, com s'ha comentat en apartats anteriors, provoca que la capa actual depengui de les altures de la capa anterior, per últim, *Ridge Noise*, que provoca que l'algoritme utilitzi l'algoritme de soroll del Ridge Noise en comptes del Perlin Noise.

Tot seguit, es troben les opcions per generar els valors del soroll comentats anteriorment.

Sota d'aquesta interfície es poden trobar les opcions per donar color al planeta.

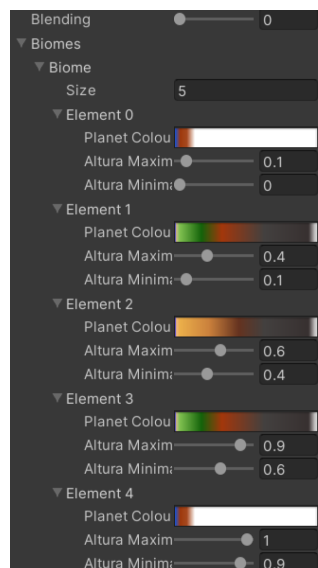


Figura 38. Primera part de la interfície per modificar els colors del planeta.

A la part superior l'opció, *Blending*, permet augmentar o disminuir el factor de suavitzat entre els diferents biomes, després podem trobar els biomes, el primer requadre permet introduir el nombre de biomes que volem al nostre planeta, i igual que amb les capes de so, apareixeran tantes opcions com les que hem seleccionat.

Cada color permet modificar el seu gradient. Que permet modificar els colors i el seu ordre, i si ho desitgem podem guardar la configuració o carregar-ne alguna que hàgim guardat anteriorment.

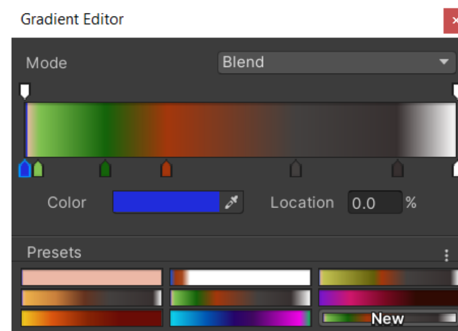


Figura 39. Editor del color del bioma.

També es pot modificar l'altura on comença i acaba cada un dels biomes, aquests dos paràmetres estan relacionats amb l'anterior i següent bioma respectivament, de manera que no es poden superposar.

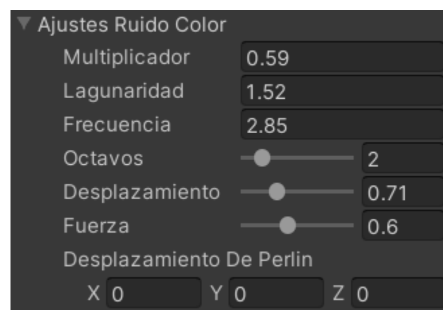


Figura 40. Segona part de la interfície per modificar els colors del planeta.

Les últimes opcions per modificar els colors, tenen a veure amb la màscara de soroll que s'aplica a cada bioma, semblants a les de les altures.

3.6 Problemes durant el desenvolupament del programa final.

Com he comentat en la introducció de l'apartat de la implementació, durant el desenvolupament del programa final, vaig invertir gran part del mes d'agost en arreglar problemes. Tot i que al final les solucions van resultar ser molt senzilles, eren problemes molt amagats i per tant, per trobar aquesta solució, va comportar moltes hores de prova i error.

La idea del programa era poder generar diferents planetes simultanis, aquest ha sigut un gran inconvenient ja que implicava canviar petites coses del projecte en general que no havia tingut en compte en el cas que hi haguessin més d'un planeta generat en pantalla.

Per poder fer això, s'havia de crear un material per a cada planeta i cada material havia de tindre la seva textura diferent. En un principi sol hi havia un material i una textura, això va canviar per poder fer cada planeta diferent, cada planeta tenia el seu material, però la textura es generava per cada planeta, i durant la generació, aquesta textura s'aplicava al material, que posteriorment es guardava en el sistema de fitxers.

Però fent això, quan es tancava Unity i es tornava a obrir, les textures dels planetes desapareixien, després d'un llarg temps d'investigació, vaig trobar que les textures en un material no es guarden, ja que estaven guardades en memòria de Unity, i per tant en esborrar la memòria, les textures desapareixien.

Per tant la solució va ser guardar cada textura independentment i després aplicar-les al material, cosa que va comportar un altre error important. Les textures en Unity han de ser múltiples de 4, per tant, si l'usuari introduïa cinc biomes, quan es transformava la textura a formant *png,* es comprimia per tal de complir aquest requisit, això provocava que la textura no es vegues correctament.

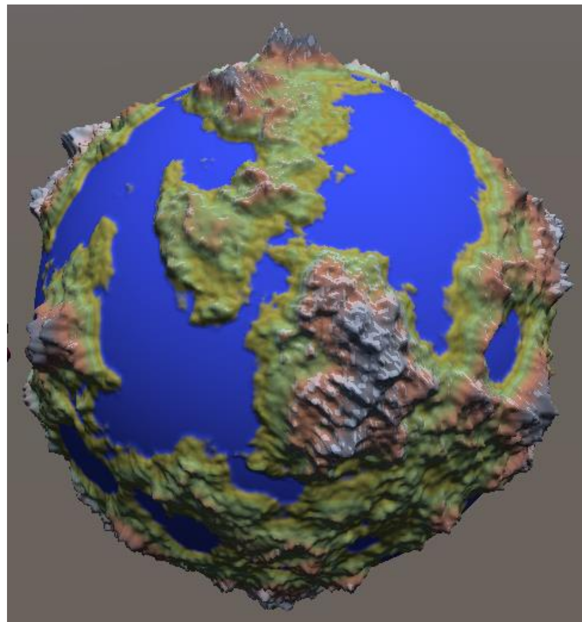


Figura 41. Planeta amb textura comprimida.

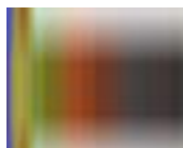


Figura 42. Textura comprimida.

Després de buscar múltiples maneres per guardar la imatge sense compressió i no aconseguir cap resultat satisfactori, simplement mirant les opcions de la textura guardada i tocant diferents variables, vaig trobar que canviant una opció anomenada *TextureType*, l'error s'havia resolt.

4. Resultats

A continuació adjunto un conjunt de planetes generats al programa per demostrar les capacitats d'aquest.

El primer dels planetes, pretén ser una rèplica de la terra, amb els pols nevats, l'hemisferi més groguenc a causa de les altes temperatures i entre aquestes dues zones, terreny més humit, per tant amb el verd dels boscos. Es poden observar formacions muntanyoses realistes amb neu al cim. També s'ha afegit el color blau a les zones més baixes per representar el mar.

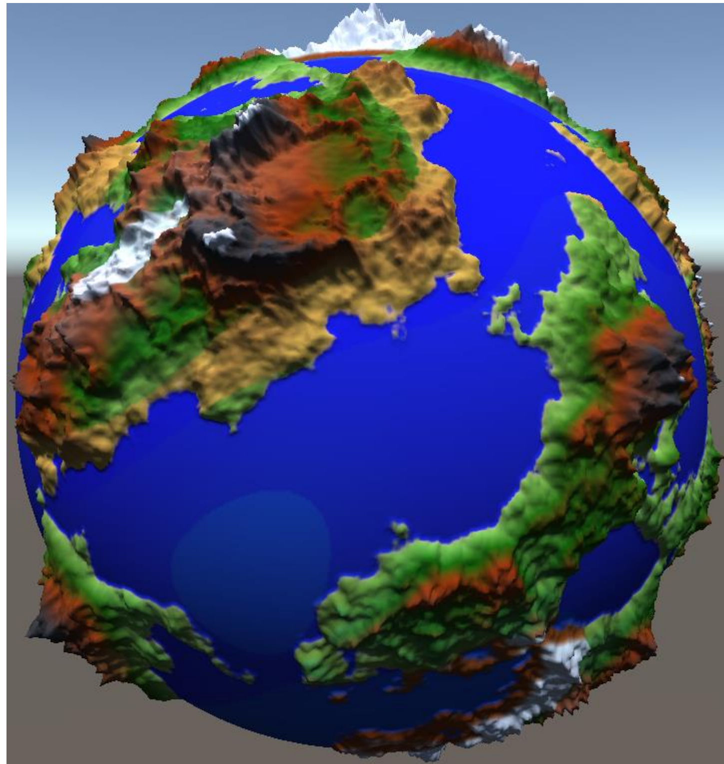


Figura 43. Planeta d'exemple 1.

Figura 44. Opcions del planeta 1.

El segon planeta ja és una invenció, mig inspirada en mart amb un terreny més àrid, però amb colors llampants a les cimes. En aquest planeta s'ha utilitzat de manera accentuada el Ridge Noise, com es pot observar en les muntanyes corbades que es poden trobar per tot el terreny.

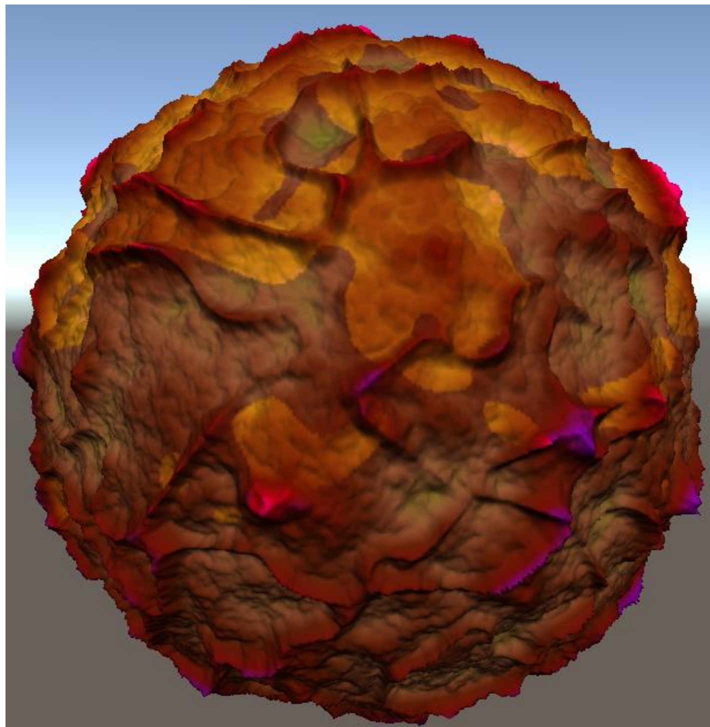


Figura 45. Planeta d'exemple 2.

Figura 46. Opcions del Planeta 2.

A diferència de les opcions d'aquest planeta, si ens fixem en la capa de so del Ridge Noise, els octaus estan establerts en 1, de manera que el resultat visible és una sola iteració del Ridge Noise, i per tant no genera un Fractal Noise, així obtenim un conjunt de muntanyes unides entre elles i aquestes tenen poques variacions d'altura, ja que no se'ls hi aplica el detall de la freqüència. Per altra banda, en els colors, he establert el *Blending* a 0 per tal que es pugui veure com varia el color generat amb l'algoritme de soroll.

Per últim, un planeta que podria assimilar-se a algun del nostre sistema solar que es troba lluny del sol, amb colors més freds, i en aquest cas, unint dues capes de Perlin Noise, de manera que es generen formacions muntanyoses menys puntegudes.

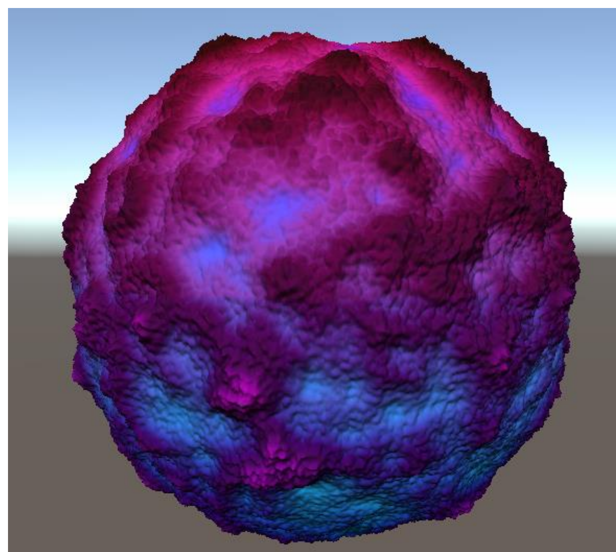


Figura 47. Planeta d'exemple 3.

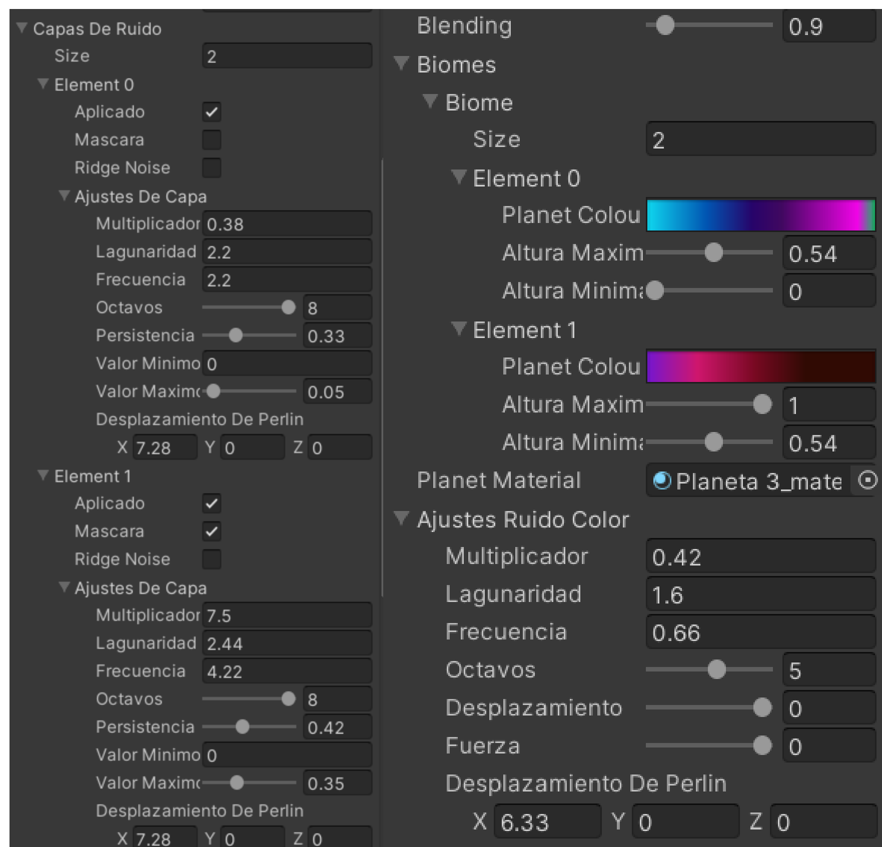


Figura 48. Opcions del Planeta 3.

Aquí podem observar, a diferència dels altres dos planetes, que no utilitzem el Ridge Noise, tenim dos Perlin Noise estàndard, el segon usant de màscara el primer, ambdós amb el valor màxim d’octaus i una freqüència i lacunaritat altes, per tant es pot veure un major detall en el planeta, com que hi ha petites muntanyes per tota la superfície. A banda, si ens fixem en els colors, en aquest cas no hi apliquem soroll, ja que la força està a 0, i el *Blending* és a un valor baix, com es pot observar, això provoca que la part superior i la inferior tenen els colors originals del seu bioma, i al centre es pot observar com ja estan mesclats.

5. Treball Futur

Tot i que l'objectiu del projecte ha sigut assolit, aquest programa té moltes més possibles millores tant per millorar l'aspecte com l'optimització:

- Compute Shaders: Pel fet que hi ha un gran nombre de polígons i punts que s'han de calcular en temps real, en aquest cas s'ha limitat el nombre de punts en 360.000, però en els moments inicials de la creació, vaig intentar utilitzar compute shaders, que utilitzen la potència de la targeta gràfica per fer càlculs en paral·lel, de manera que es podria tindre un nombre molt més elevat de polígons i el programa funcionaria més de pressa.
- Detalls al terreny: Tot i que visualment els resultats ja són prou bons, per tal que aquest programa fos útil en el món real, requereix moltes més opcions, ja sigui afegir-hi coves, que també es generen amb algoritmes de soroll, s'hi poden afegir arbres i roques, cada un amb el seu algoritme de generació procedural o inclús, es podria indicar en quins biomes hi ha més quantitat d'aquests objectes.
- Estructures: Si pensem en videojocs, seria necessari tindre un conjunt d'estructures on el personatge podria viatjar, com per exemple ciutats o punts d'interès com piràmides als deserts, ruïnes antigues a les muntanyes… I a més, aquestes estructures podrien ser generades proceduralment, la ciutat generaria els seus habitants, el seu govern, els edificis els carrers, i les ruïnes podrien ser masmorres generades proceduralment amb enemics també aleatoris.
- Millor coloració: En comptes d'usar colors plans, hi ha l'opció de donar textures diferents al planeta depenent també del bioma, aquestes textures, com ja he comentat, també es podria donar el cas que fossin generades proceduralment. El fet d'usar textures en comptes de colors, ajudaria a donar més realisme al planeta.

Òbviament, això requeriria moltíssima feina i temps, i per tant, no ha sigut possible implementar-ho en aquest projecte.

6. Conclusions

L'objectiu principal del projecte era entendre i estudiar com funcionen els algoritmes procedurals per ser capaç de generar terrenys realistes, diferents i fàcils de generar en un entorn gràfic per un usuari.

Aquest objectiu ha sigut assolit, tot i que amb els seus problemes durant amb el desenvolupament, però queda demostrat que els algoritmes de generació procedural poden ser molt útils encara que en aquest cas hi hagi molta feina al davant per tal que sigui un programa utilitzat en el món real. Desenvolupant aquest projecte m'he adonat de les dificultats de generar un planeta o terreny amb els algoritmes procedurals i l'admiració de la gent que es dedica a treballar amb aquests algoritmes, que, tot i que en si són senzills d'utilitzar per separat, per tal de donar vida a un terreny, és necessari molt més que generar unes quantes muntanyes, que inclús aquesta tasca, comporta molta més feina de la que vaig anticipar, ja que hi intervenen molts factors externs per generar estructures més complexes i realistes.

Alhora, encara que no fos un objectiu de la pràctica, he après a programar amb Unity i entenc les bases del seu funcionament i les coses que s'hi poden fer, ja sigui el tipus d'objectes i classes que tenen, com l'ús de shaders i de l'editor de nodes, cosa que sempre havia tingut pendent, ja que, sempre m'havia cridat l'atenció poder fer algun petit joc, però mai m'hi havia atrevit. Però alhora, el fet de no conèixer com funciona Unity, és el que m'ha comportat mes problemes per desenvolupar aquets programa, tot i que a Unity es treballa amb el llenguatge C#, que, tot i que no l'he après durant el transcurs de la carrera a la universitat, és molt semblant als seus germans C i C++, per tant no era un problema de no conèixer el llenguatge. El problema residia en què s'ha de treballar amb objectes virtuals, per donar-los-hi forma i color, amb un conjunt de classes i mètodes únics de Unity desconeguts per mi, per tant, un gran temps del treball va ser utilitzat per conèixer que es pot fer i que no.

Un exemple clar del meu desconeixement va comportar una gran inversió de temps en un primer prototip que no va donar els resultats esperats i només em va fer perdre molt de temps, això va ser degut al fet que el programa anava molt lent i un cop vaig aconseguir un resultat relativament bo, em vaig adonar que la personalització per part de l'usuari era molt baixa, ja que només es permetia que, amb una espècie de pinzell, l'usuari augmentes o disminuís l'altura del terreny on tenia el cursor.

Personalment, tot i que no he tingut la millor experiència fent el treball de fi de grau, per molts problemes, mals de cap i temps que m'ha comportat fer-lo, gaudia cada moment que aconseguia un nou objectiu, o cada cop que es notava una millora en la generació, dedicar hores i rebre resultats ha sigut una experiència satisfactòria al cap i a la fi. Em van advertir que escollis algun treball senzill, però em vaig decidir per alguna cosa que m'agrades, i crec que al final això és el que dona la motivació per treballar en el projecte.

7. Referències

[1] Pagina Web: https://www.redblobgames.com/maps/terrain-from-noise/. Making maps with noise funcions from Red Blob Games, Juliol 2015.

[2] Pàgina Web: http://www-cs-students.stanford.edu/~amitp/game-programming/polygon-map-generation/. Polygonal Map Generation for Games, 4 de Setembre de 2010.

[3] Pàgina Web: https://www.redblobgames.com/articles/noise/introduction.html. Noise Functions and Map Generation, 31 d'Agost de 2013

[4] Article Web: https://weber.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf. Simplex noise demystified, 22 de Març de 2005.

[5] Pàgina Web: https://rtouti.github.io/graphics/perlin-noise-algorithm. Perlin Noise: A Procedural Generation Algorithm.

[6] Pàgina Web: https://adrianb.io/2014/08/09/perlinnoise.html. Understanding Perlin Noise, 9 d'Agost de 2014.

[7] Pàgina Web: https://stackoverflow.com/questions/36796829/procedural-terrain-with-ridged-fractal-noise. Procedural Terrain with ridged fractal noise, 22 d'Abril de 2016.

[8] Pàgina Web: https://thebookofshaders.com/13/. Fractal Brownian Motion, 2015.

[9] Pàgina Web: https://www.ronja-tutorials.com/post/027-layered-noise/. Layered Noise, 22 de Setembre de 2018.

[10] Pàgina Web: https://catlikecoding.com/unity/tutorials/cube-sphere/. Cube Sphere,

[11] Pàgina Web: https://medium.com/@oscarsc/four-ways-to-create-a-mesh-for-a-sphere-d7956b825db4. Four Ways to Create a Mesh for a Sphere, 7 de Desembre de 2015

[12] Pàgina Web: https://gamedev.stackexchange.com/questions/166468/how-do-i-generate-a-sphere-mesh-in-unity. How do I generate a sphere mesh in Unity?, 24 de Desembre de 2018.

[13] Pàgina Web: https://www.binpress.com/creating-octahedron-sphere-unity/. Creating an Octahedron Sphere in Unity.

[14] Pàgina Web: https://scaryreasoner.wordpress.com/2016/01/23/thoughts-on-tesselating-a-sphere/. Thoughts on tesselating a sphere, 23 de Gener de 2016.

[15] Pàgina Web: https://es.wikipedia.org/wiki/N%C3%BAmero_triangular. Número Triangular, 26 d'Octubre de 2020.

[16] Pàgina Web: https://docs.unity3d.com/Manual/index.html. Documentació Unity.