Rapport UE-INF2315M Modélisation Géométrique

COTTIER Alexandre

Le code : https://github.com/Mantador01/modelisation-geometrique

Sujet 1. Modélisation à l'aide de Blobs (description) :

Ma classe implémente un champ scalaire implicite défini comme la somme des contributions de plusieurs primitives.
Chaque primitive correspond à une source d'influence dont l'effet décroît avec la distance.

J'ai une fonction pour clear la liste des primitives existantes pour repartir d'un champ vide et une qui définit la valeur iso utilisée comme seuil de la surface implicite.

Ensuite je défini AddPoint(P, R, w, c) qui ajoute une primitive de type point.
Chaque point agit comme un blob influençant le champ autour de lui dans un rayon R.

Ensuite AddSegment(A, B, R, w, c) qui ajoute une primitive de type segment.
Cette primitive relie deux points et crée un tube lisse entre les blobs.

Et enfin Value(x) qui calcule la valeur du champ au point x :

- Si le champ ne contient aucune primitive, elle renvoie une simple sphère.
- Sinon, elle somme les contributions de toutes les primitives :
- La fonction d'influence utilisée est le noyau borné de Wyvill, cubique et lisse à la frontière.

Modélisation à l'aide de Blobs (exemple) :

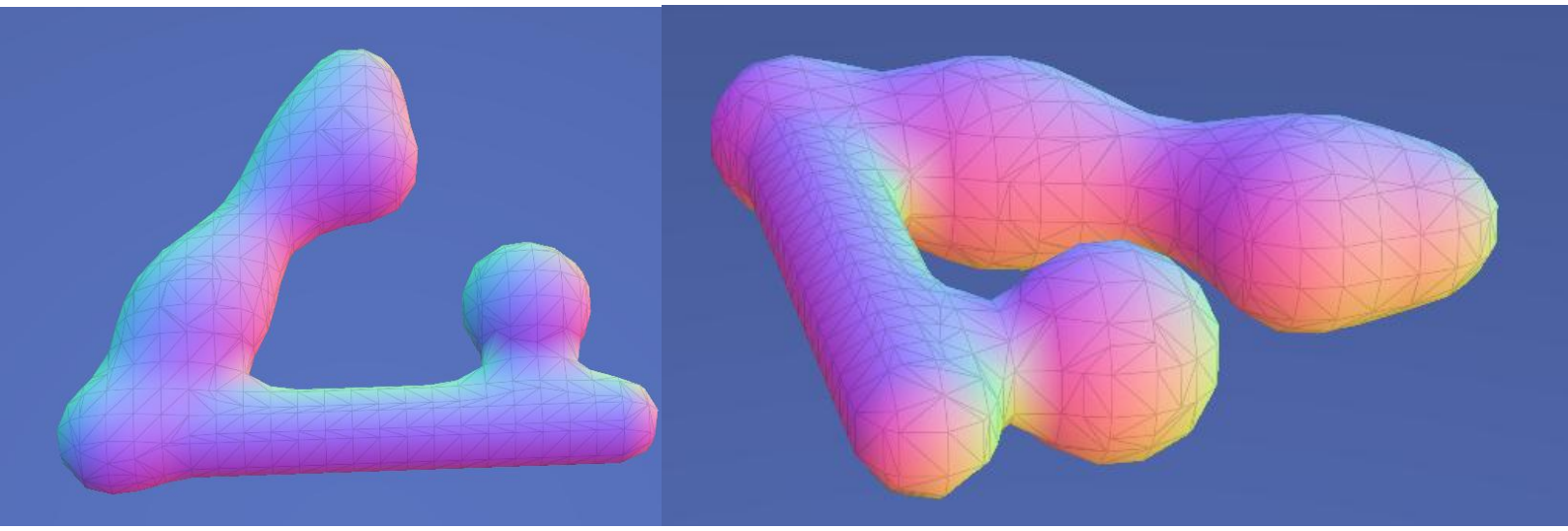


Figure 1 : Exemple d'un blob avec 3 points et 2 segments.

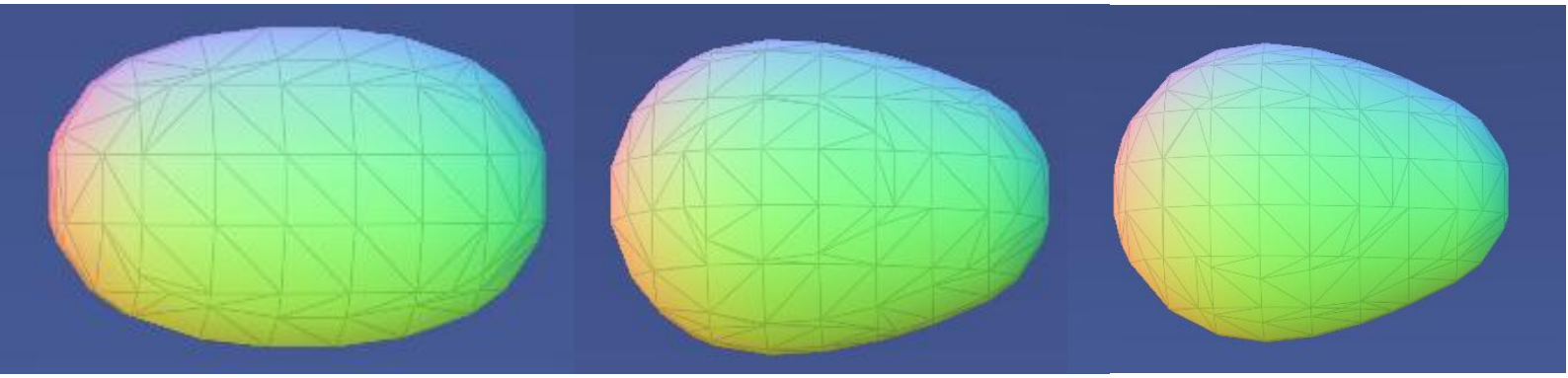
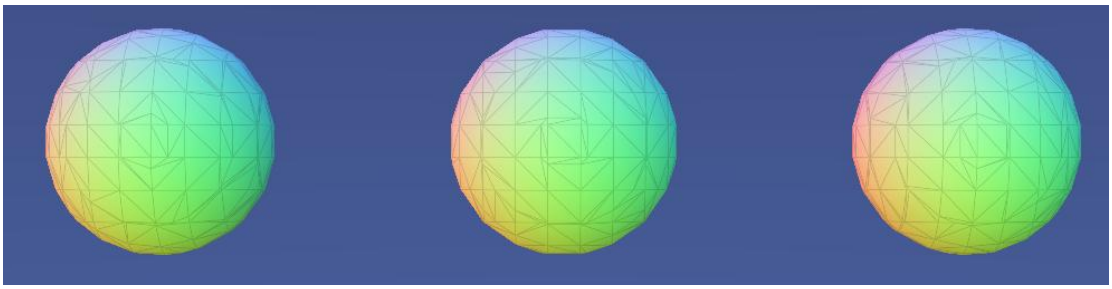


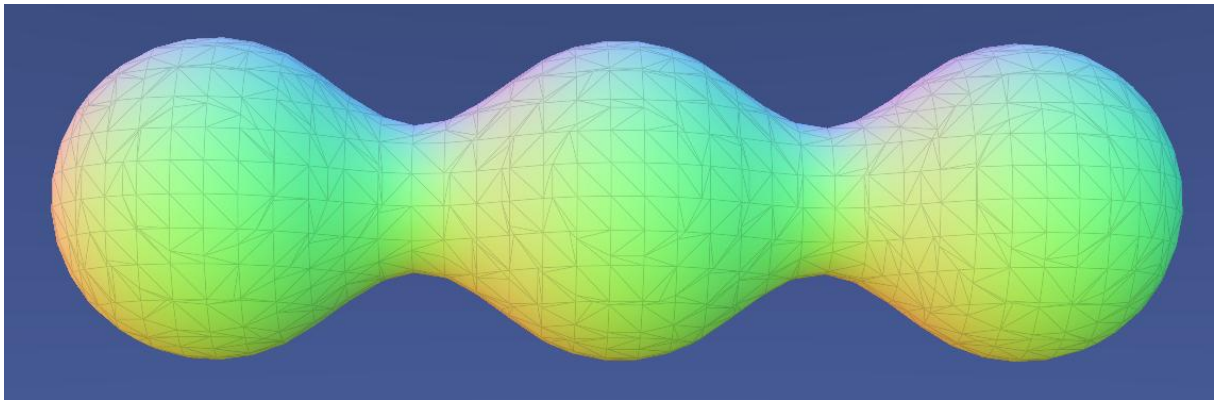
Figure 2 : Exemple d'un blob mais avec le point de gauche ayant un w croissant (1.0, 2.0 et 3.0, celui de droite reste a 1.0), la bulle de plus grand poids "tire" la surface vers elle.

Figure [3,4,5,6] : *Trois blobs alignés fusionnent progressivement lorsque le rayon augmente.*

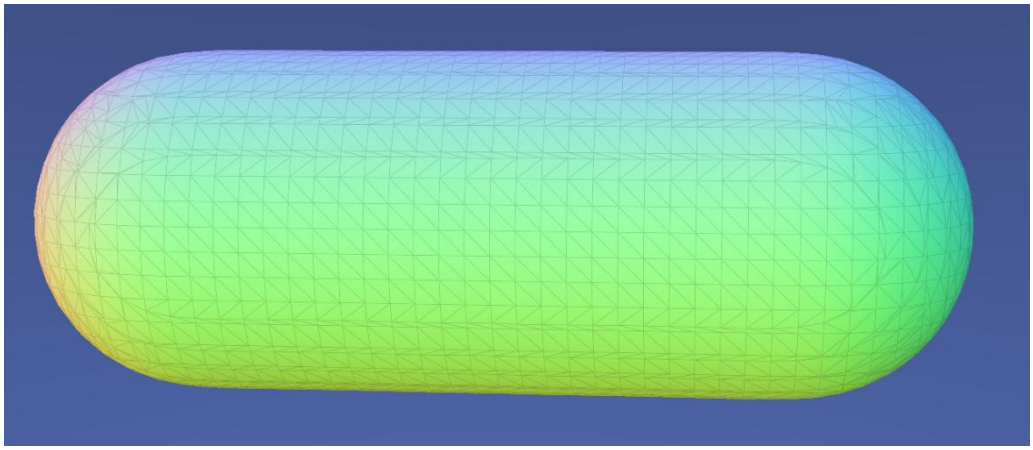
Figure 3 : blobs avec rayon a 0.15 → trois sphères séparées.



<u>Figure 4 :</u> blob avec rayon 0.25 → on se rapproche.



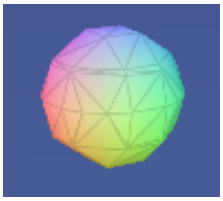
<u>Figure 5 :</u> blob avec rayon 0.35 → fusion partielle.



<u>Figure 6 :</u> blob avec rayon 0.5 → volume englobant (forme lisse).

Modélisation à l'aide de Blobs (performances) :

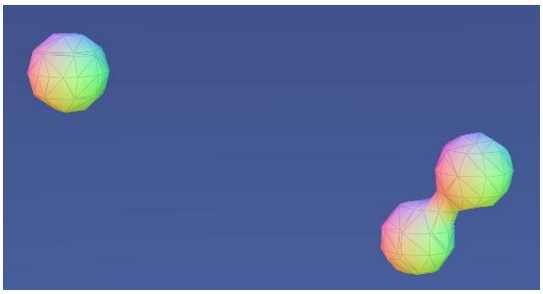
Les tests ici sont réalisées avec un n = 64 (résolution), dans une box



Nb de primitives : 1

Temps en ms : 13

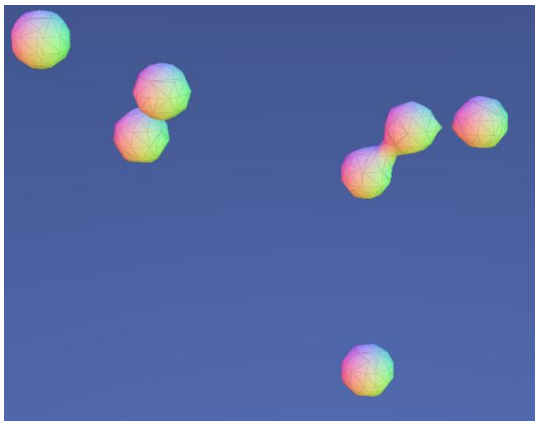
Nb de triangles : 108



Nb de primitives : +2 (en plus des précédents)

Temps en ms : 20

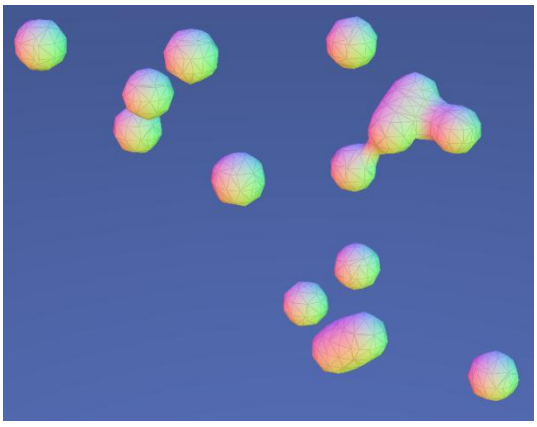
Nb de triangles : 352



Nb de primitives : +4

Temps en ms : 32

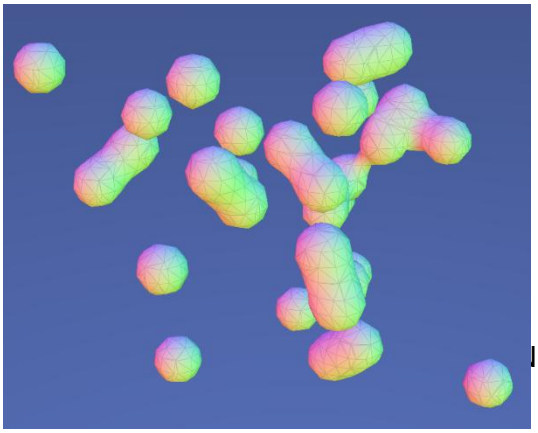
Nb de triangles : 808



Nb de primitives : +8

Temps en ms : 57

Nb de triangles : 1712

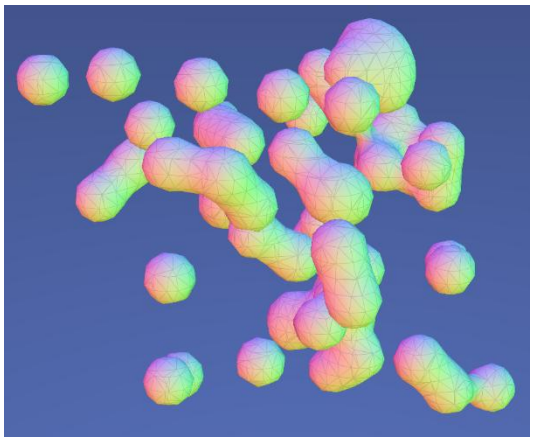


Nb de primitives : +16

Temps en ms : 103

Nb de triangles : 3496

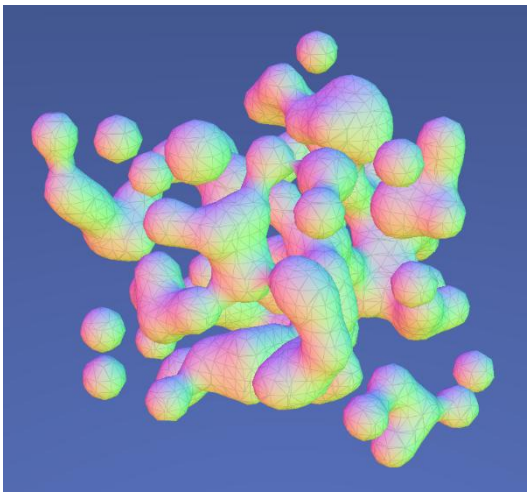
1



Nb de primitives : +32

Temps en ms : 222

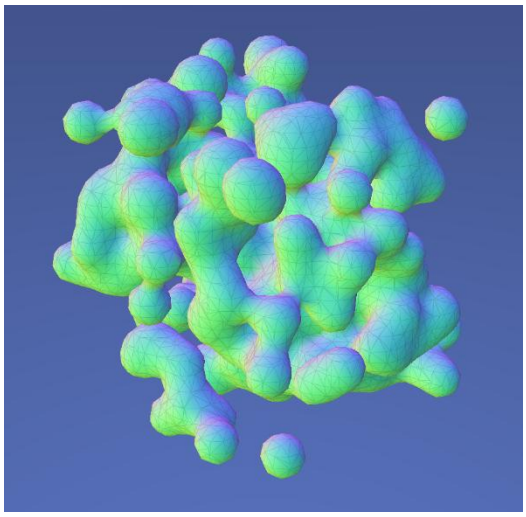
Nb de triangles : 6852



Nb de primitives : +64

Temps en ms : 476

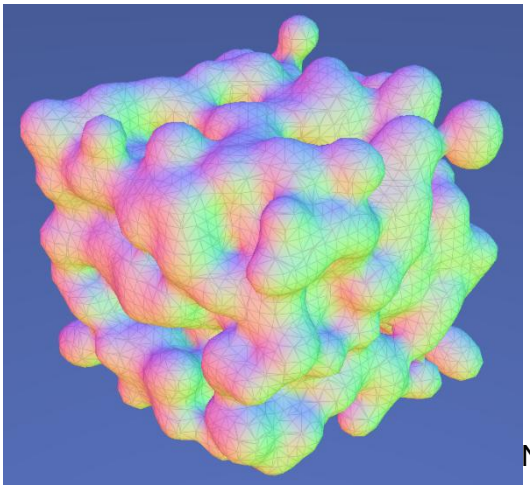
Nb de triangles : 12556



Nb de primitives : +128

Temps en ms : 1081

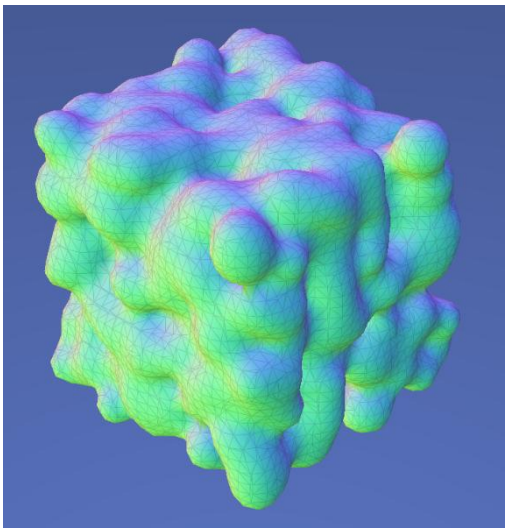
Nb de triangles : 18524



Nb de primitives : +256

Temps en ms : 2200

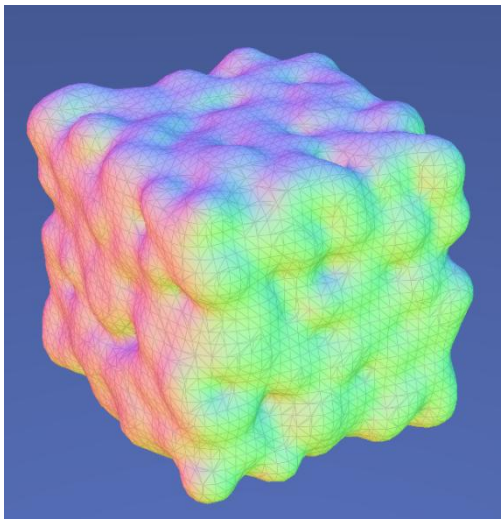
Nb de triangles : 19368



Nb de primitives : +512

Temps en ms : 3846

Nb de triangles : 14144 (ça décroit ! → fusion des blobs)



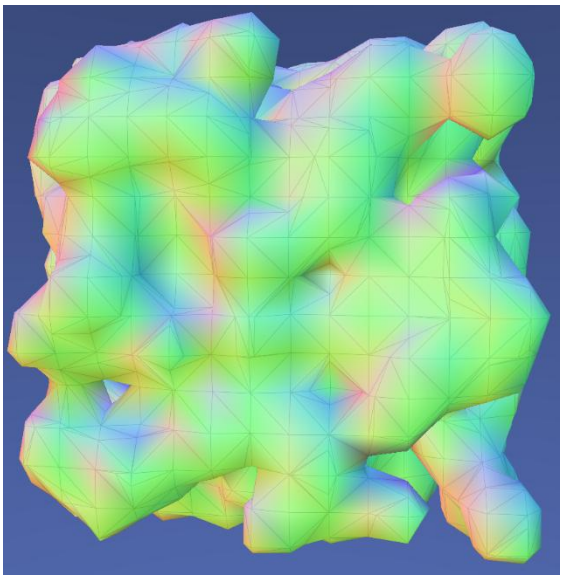
Nb de primitives : +1024

Temps en ms : 7654

Nb de triangles : 12564 (encore mieux)

On observe une croissance linéaire du temps avec le nombre de primitives, on a une complexité de $O(n^3 \times m)$, où n^3 est à la résolution spatiale et m le nombre de blobs.

Et si on touchais a la résolution ?



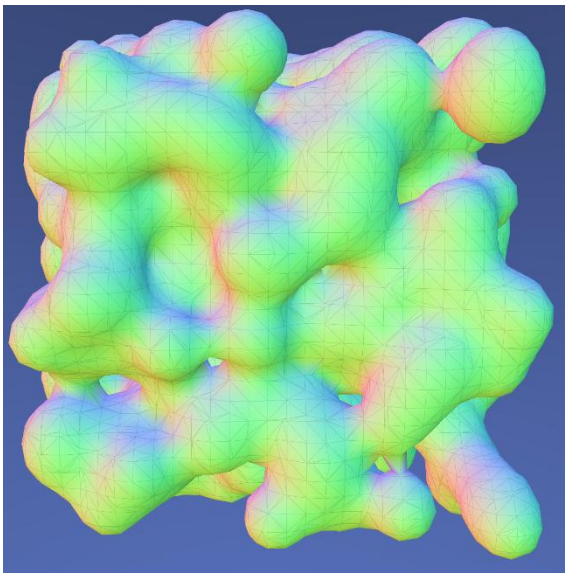
n = 32

Nb de primitives : +256

Temps en ms : 314 (2200 en n=64)

Nb de triangles : 4492 (19368 en n=64)

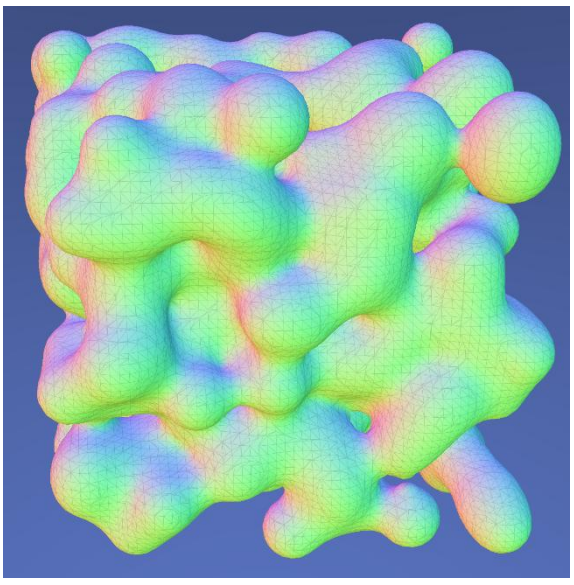
Maillage très brouillon/brute mais rapide.



n = 64

(Juste pour comparaison visuelle, données identiques que dans la partie du dessus)

Surface fluide, un bon compromis.



n = 128

Nb de primitives : +256

Temps en ms : 12127

Nb de triangles : 79976

Surface très détaillée, calcul beaucoup plus long.

Analyse des performances

Blobs Benchmark (n=32)

count	ms	triangles
1	1	16
2	2	64
4	3	152
8	7	320
16	14	728
32	30	1388
64	68	2764
128	156	4320
256	314	4492
512	564	3424
1024	956	2640

Blobs Benchmark (n=64)

count	ms	triangles
1	14	108
2	20	352
4	32	808
8	57	1712
16	103	3496
32	222	6852
64	476	12556
128	1081	18524
256	2200	19368
512	3846	14144
1024	7654	12564

Blobs Benchmark (n=128)

count	ms	triangles
1	90	464
2	122	1452
4	196	3408
8	364	7180
16	689	14840
32	1344	28196
64	2888	51748
128	6197	76408
256	12127	79976
512	22588	59252
1024	43568	51348

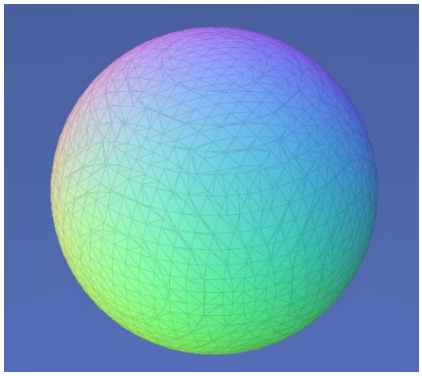
Figure 7 : tableaux des 3 différentes résolutions tester, je fini avec pour valeurs max

n = 32 → ~950 ms, n = 64 → ~7650 ms, n = 128 → ~43 s

À nombre de primitives fixe, le temps est multiplié par environ 8 à chaque doublement de n

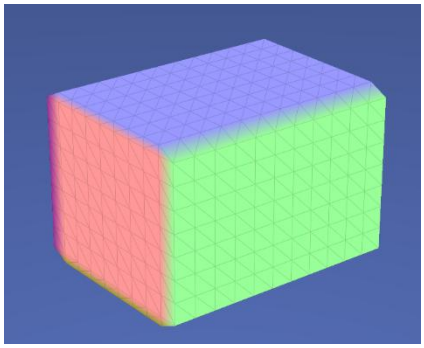
Sujet 2. Modélisation à l'aide de distances signées :

J'implémente plusieurs classes qui dérivent d'une base commune SDFNode :

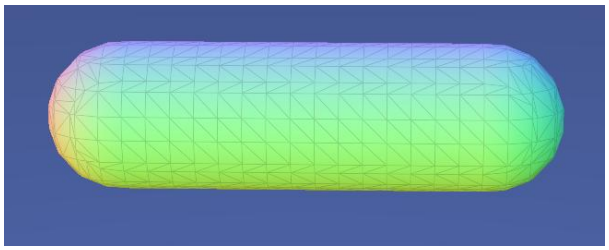


SphereNode : $Value(p) = Norm(p - c) - r$

Sphere centrée en c de rayon r

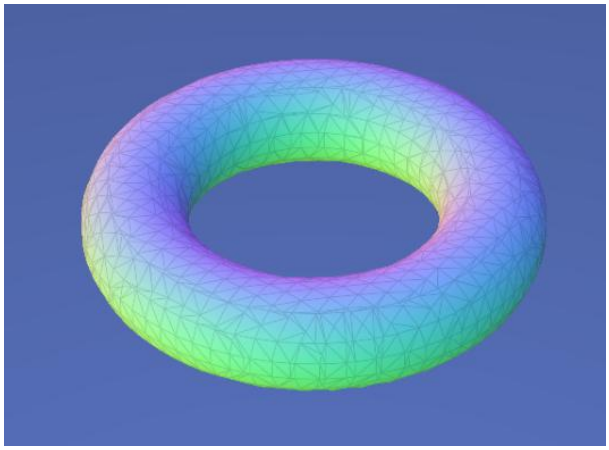


BoxNode : p - c



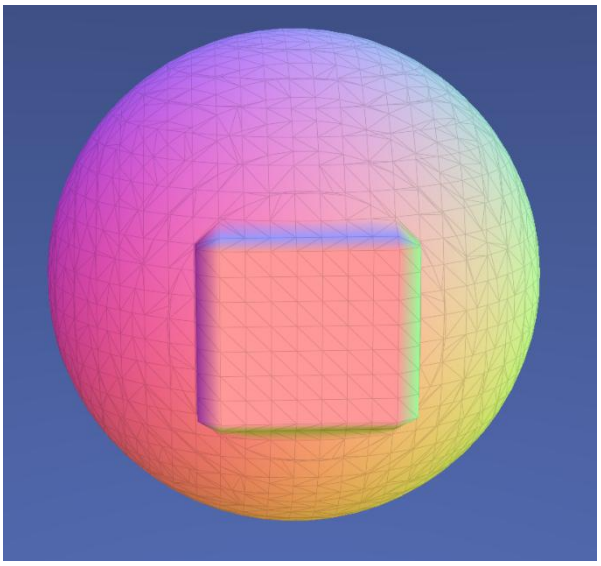
*CapsuleNode : Value(p) = Norm(pa – ba * h) – r*

Segment [A,B] gonflé d'un rayon r



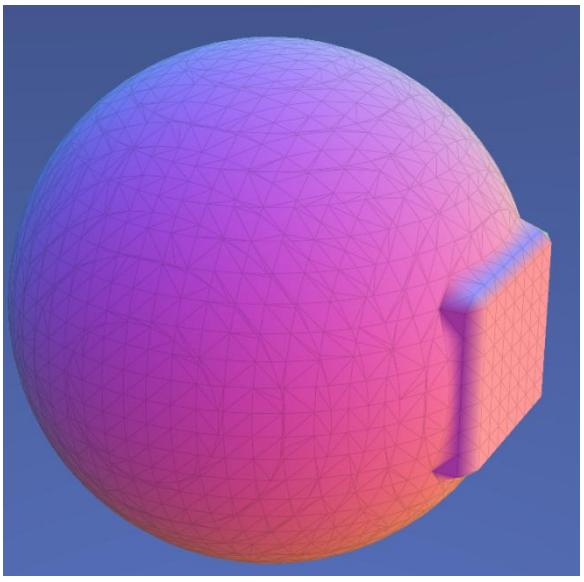
TorusNode : Value(p) = sqrt(x²+y²) - R, puis norme avec z

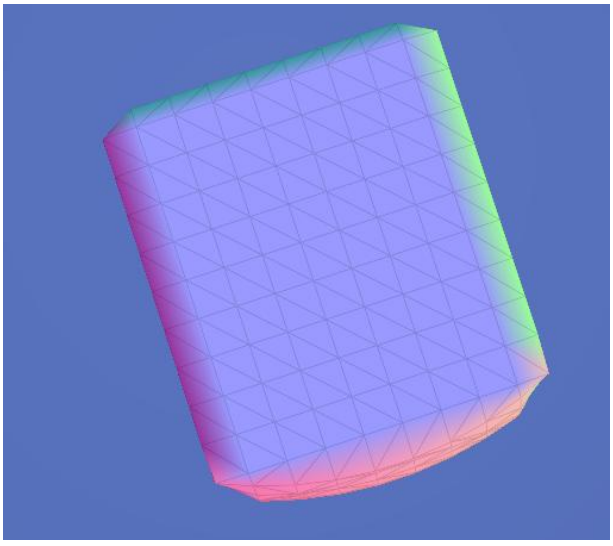
Tore autour de l'axe Z



UnionNode : min (A,B)

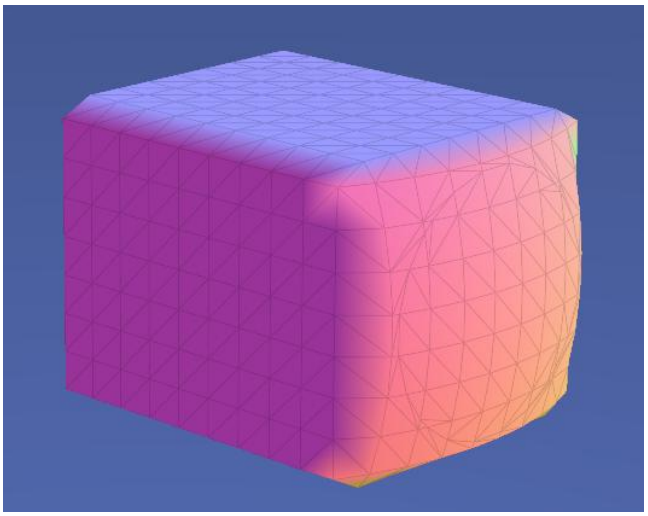
Union d'une sphère et d'une box (on voit que c'est bien mais très brut)

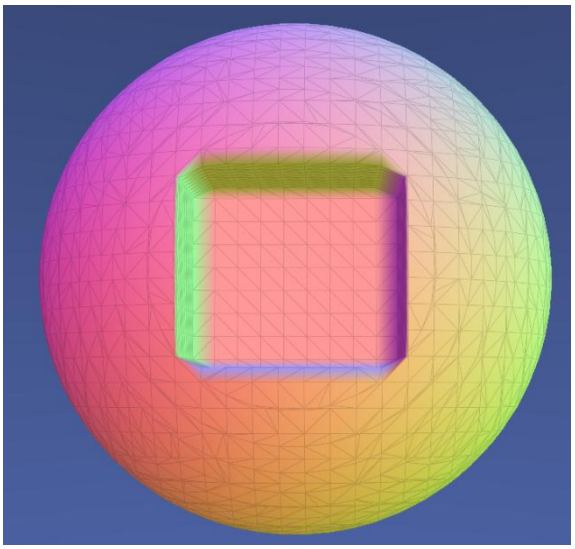




IntersectionNode : max (A,B)

Intersection d'une box et d'une sphère (j'ai essayé de donner un effet gomme avec le côté arrondi)

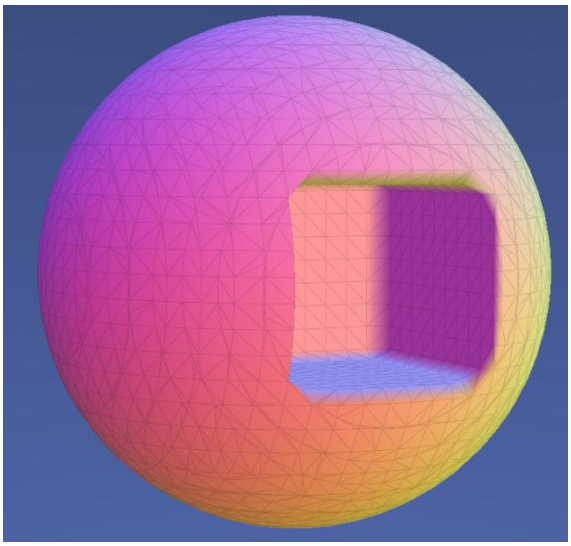


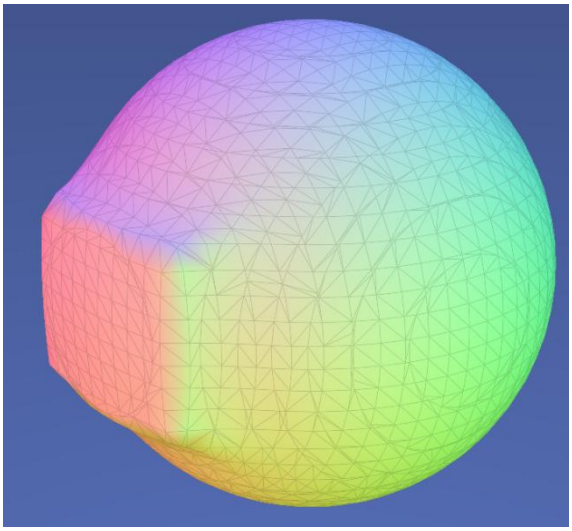


DifferenceNode :

max (A, -B)

Différence d'une box dans une sphère (ça creuse la sphère avec la forme forme cubique)

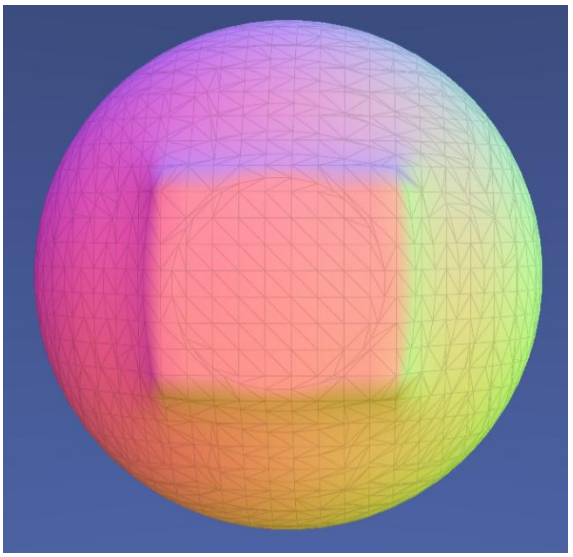


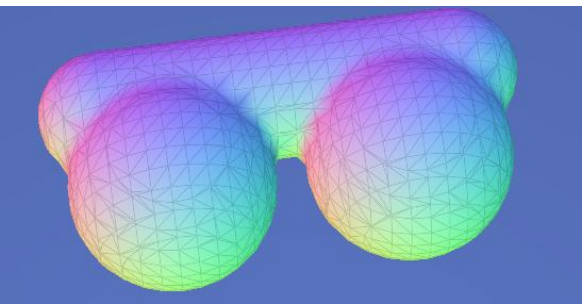


BlendNode :

Smooth-min(A,B,k)

C'est une fusion lissée d'une sphère et d'une box comme précédemment mais on voit clairement que le rendu est bien meilleur qu'avec une union simple (on pourrais même voir avec smoother-step)



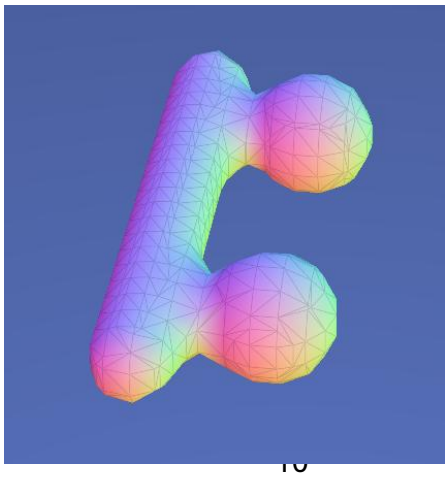


BlobApproxSDFNode : smooth-min(sphères/capsules)

Conversion métrique approximé

BlobAsSDFNode :

Conversion directe des blobs (non métrique)



Qualité du maillage :

L'augmentation de la résolution réduit le résidu quadratique moyen et améliore la précision géométrique, au prix d'un temps de polygonisation et d'un nombre de triangles plus élevés.

On a fait déjà pas mal de teste avec par exemple la résolution avec les blobs, je ne pense pas que c'est nécessaire/pertinent de recommencer ici.

Performances (10^7 appels)

Temps (s) indique le temps global mesuré entre le début et la fin des 10^7 appels.

Ns/appel est la durée moyenne d'une seule évaluation.

Et M calls/s donne le nombre d'appels qui peuvent être fait par seconde (en million).

Sphere :

Temps(s) : 1.065 106.5 ns/appel 9.39 M calls/s

C'est la plus simple SDF possible, d'où les performances écrasantes.

Box :

Temps(s) : 1.417 141.7 ns/appel 7.06 M calls/s

Chaque point doit déterminer s'il est à l'intérieur ou à l'extérieur de la boîte, ce qui le rend plus lent que la sphère (de 25% environ)

Capsule :

Temps(s) : 1.347 134.7 ns/appel 7.42 M calls/s

Une projection sur un segment + une norme → plus d'opérations vectorielles.
Mais aucune branche dure, donc performance correcte (légèrement plus rapide que box).

Torus :

Temps(s) : 1.157 115.7 ns/appel 8.64 M calls/s

Presque aussi rapide que la sphère (deux racines carrées imbriquées).

Union (sphère,box) :

Temps(s) :1.579 157.9 ns/appel 6.33 M calls/s

Nécessite deux appels complets à Value() puis une comparaison (30% plus lent).

Intersection (sphère,box) :

Temps(s) :1.617 161.7 ns/appel 6.18 M calls/s

Pareil qu'union, deux évaluations (les seules variations viennent des distrib. de min/max).

Différence (sphère – box) :

Temps(s) :1.587 158.6 ns/appel 6.30 M calls/s

Encore deux appels, plus une inversion de signe.

Blend (sphère,torus) :

Temps(s) :1.412 141.1 ns/appel 7.08 M calls/s

Là on doit être surpris car les perf. indiquent que le blend est très efficace !

(en théorie le blend est plus complexe que l'union simple, l'union lissé → le blend lui ajoute des calculs supplémentaires, on s'attendrait logiquement à ce qu'il soit plus lent que l'union, pour aller plus loin j'ai essayé de comprendre le pourquoi du comment, et de ce que j'ai compris, le CPU/GPU prefère les calculs arithmétiques (add,mul,fma) mais détestent les branches conditionnelles (if,min,max) car elles cassent la vectorisation et le pipeline d'exécution, l'union qui contient une comparaison fait diverger les threads alors que le blend lui utilise que des opérations continue).

Sujet 3. Génération procédurale de modèles représentés à l'aide surfaces implicites (description) :

Pour cette partie, j'ai implémenté :

- Une intersection par Sphere Tracing pour détecter la rencontre d'un rayon avec une surface implicite.

- Un algorithme d'érosion, simulant une sorte d'usure ou d'érosion (comme avec les cours d'eau avec les terrains) via la projection de sphères.

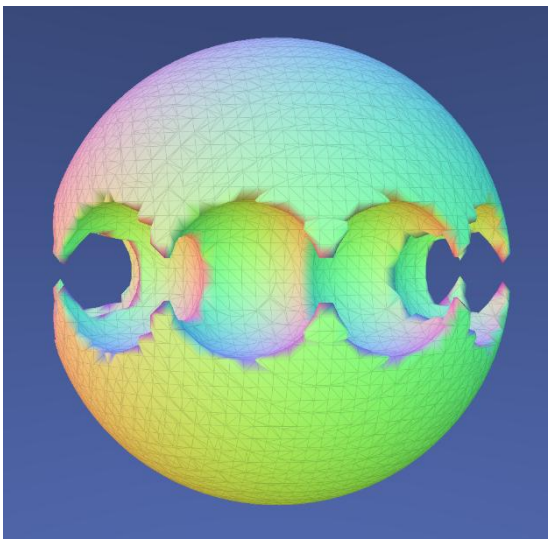
- Une comparaison des performances entre une érosion incrémentale (ajout d'impacts successifs) et batch (application d'un ensemble d'impacts en une seule évaluation).

Pour le Sphere Tracing, il consiste juste à avancer le long du rayon par pas égaux à la valeur de la fonction de distance en ce point, à chaque itération, la distance retournée par la SDF garantit qu'aucune surface n'existe avant ce point. Lorsque d < epsHit, le rayon touche la surface.

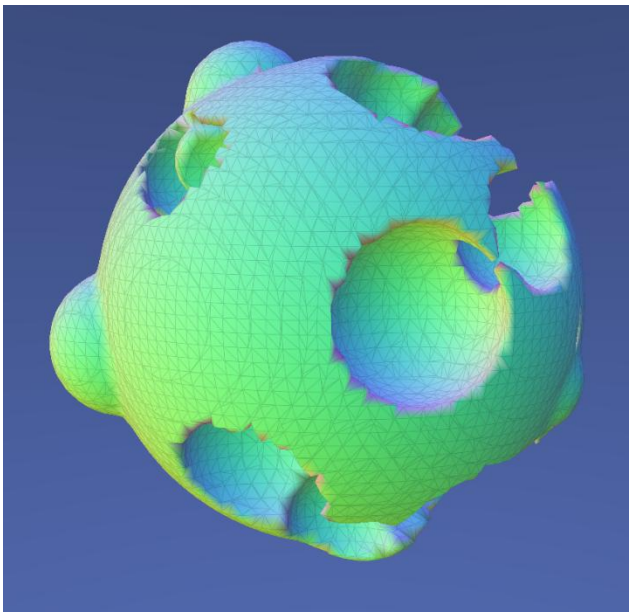
Pour l'érosion elle combine une sphère avec un ensemble de sphères d'érosion, stockées dans un vecteur, chaque sphère modifie localement la surface iso du champ.

Evidemment j'utilise les opérateurs vu et revu (smoothmax pour érosion douce, smoothmin pour une bosse douce et max pour de l'érosion nette/brute)

Tout ça modifiable est modulable à souhait, je ne suis pas aller pousser à fond les différents cas vus que j'ai traité pas mal de cas avant, mais voici certains exemples.

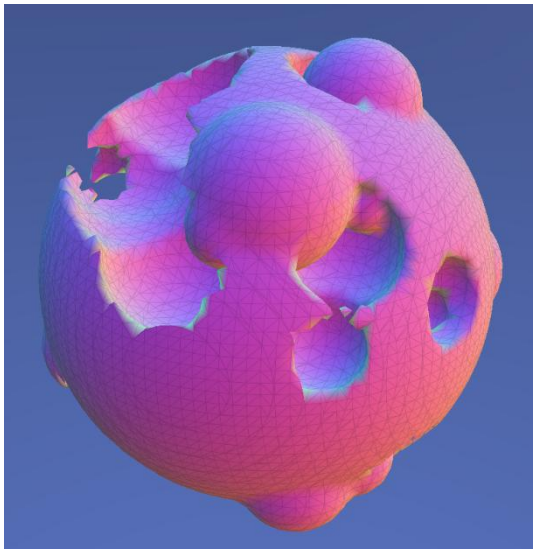


J'ai fait quelques tests, voici des trucs rigolos que je produisais, par exemple tourner avec une boucle for pour faire un joli effet.

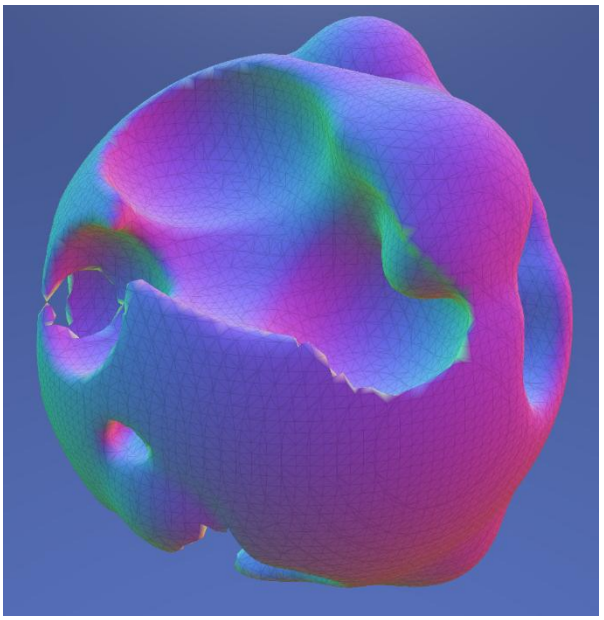


Ici je fais des trous et des bosses aléatoirement.

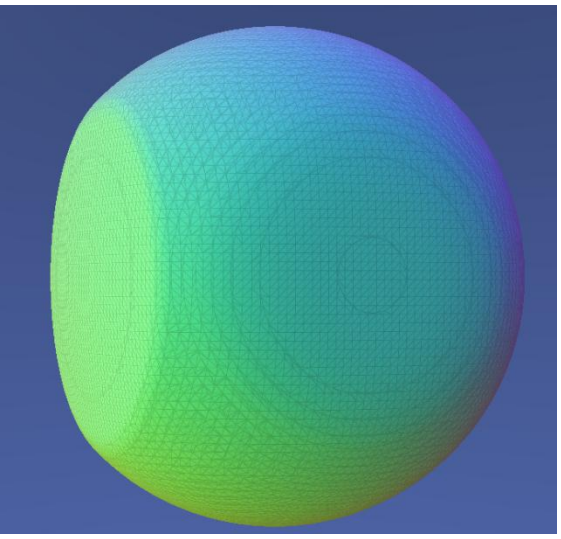
Cet exemple la donnerait vraiment envie de faire une planète beaucoup plus grosse, avec des bosses et des cratères, avec les bonnes textures ça pourrait rendre vraiment bien.



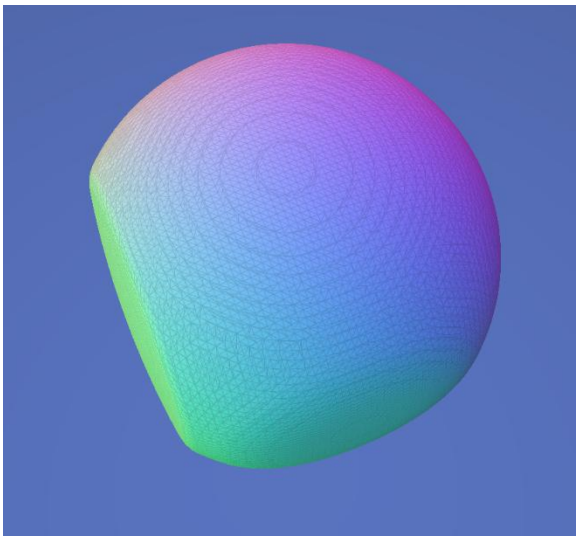
← Same →

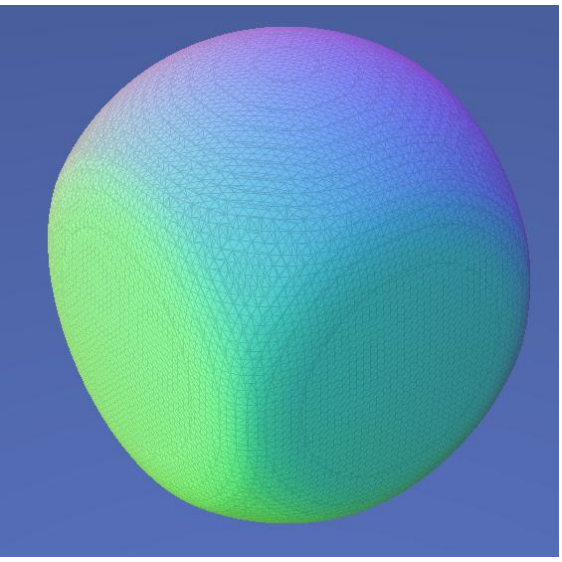


En lissant un peu plus →

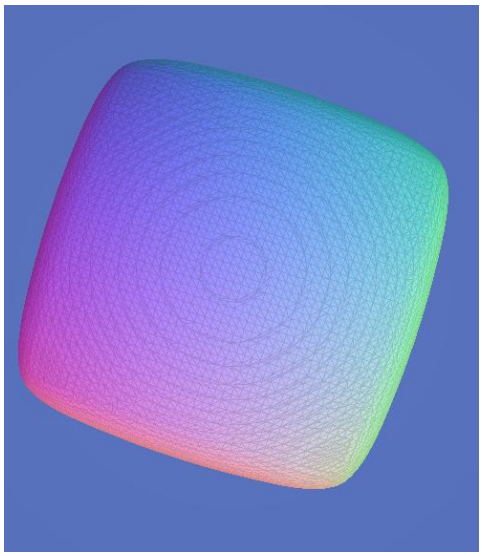


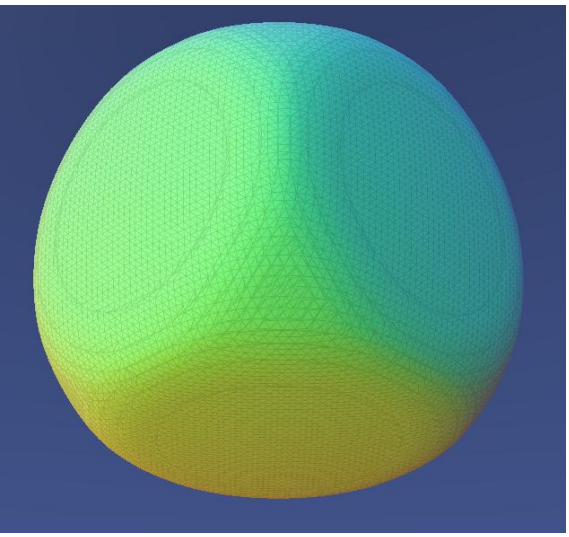
Les exemples suivants sont une implémentation ou j'avais mis cut = sd





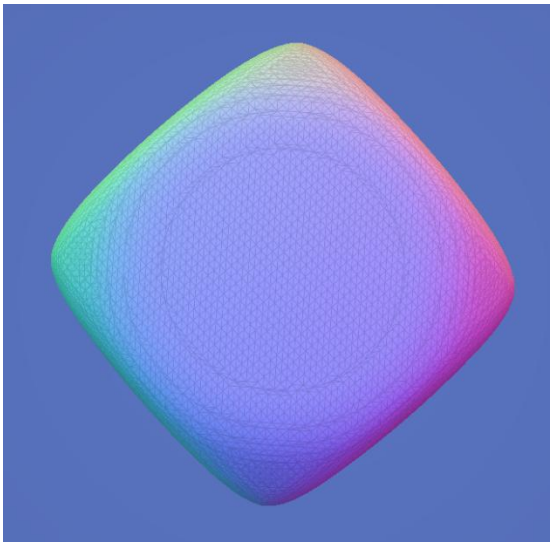
Ici donne entre autres 4 faces

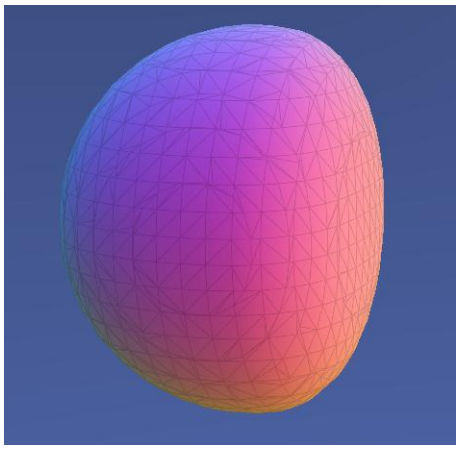




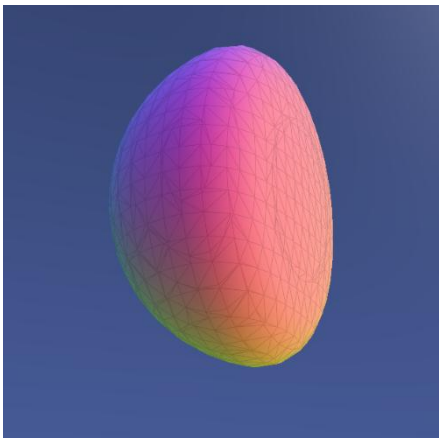
Ici les 6 faces

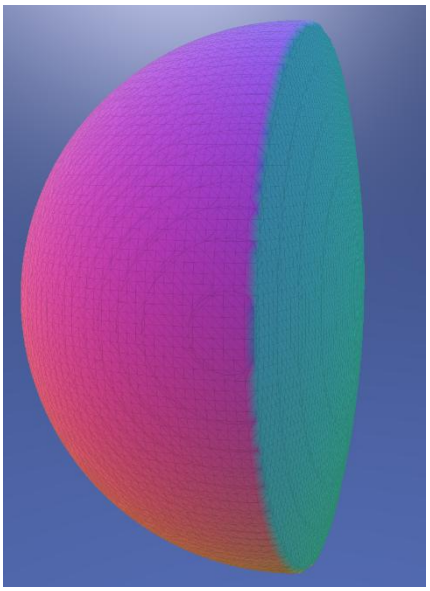
Cela donne un joli dé, avec les arrondis aux bords !



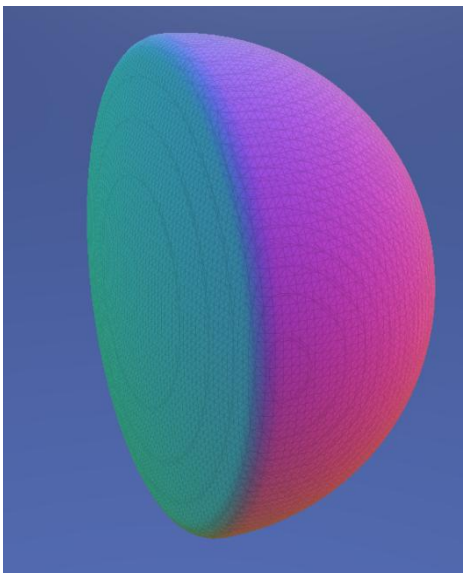


Encore un exemple

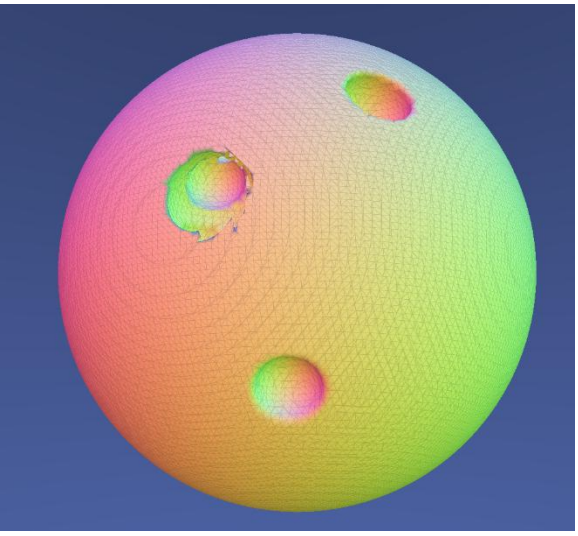




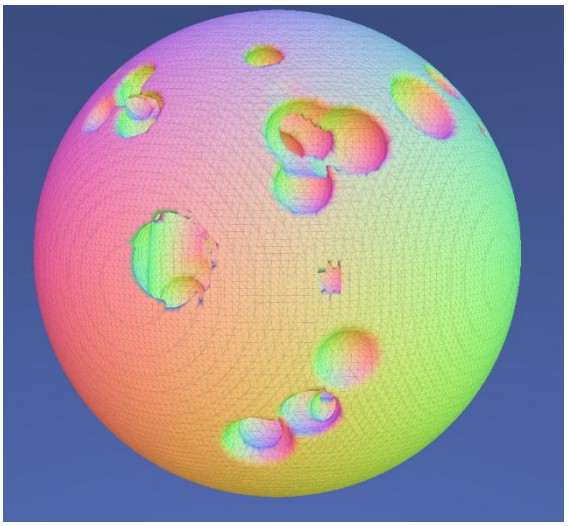
Ici on voie bien avec et sans lissage



Les exemples ici sont la même chose mais avec cut = -sd



Ici je fais des trous aléatoires, ce qui donne des cratères plus ou moins profond si on creuse dans un cratère encore.



Performances :

Pour les valeurs que je n'indique pas, c'est qu'elles étaient négligeables voir nul..

Nb d'impacts	Eval. incrémentale (ms)	Eval. batch (ms)
200	115	126
1000	916	1063
3000	2761	3091

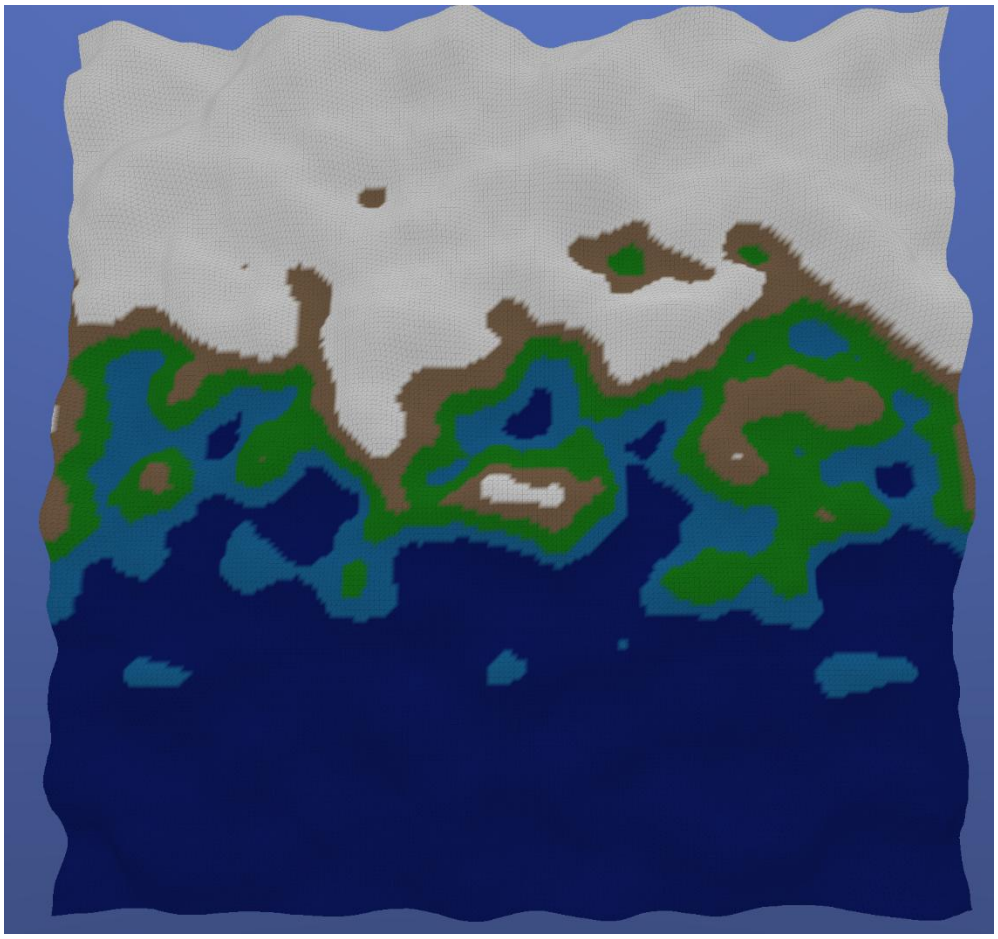
La version incrémentale est a peu près plus rapide de 10-15%

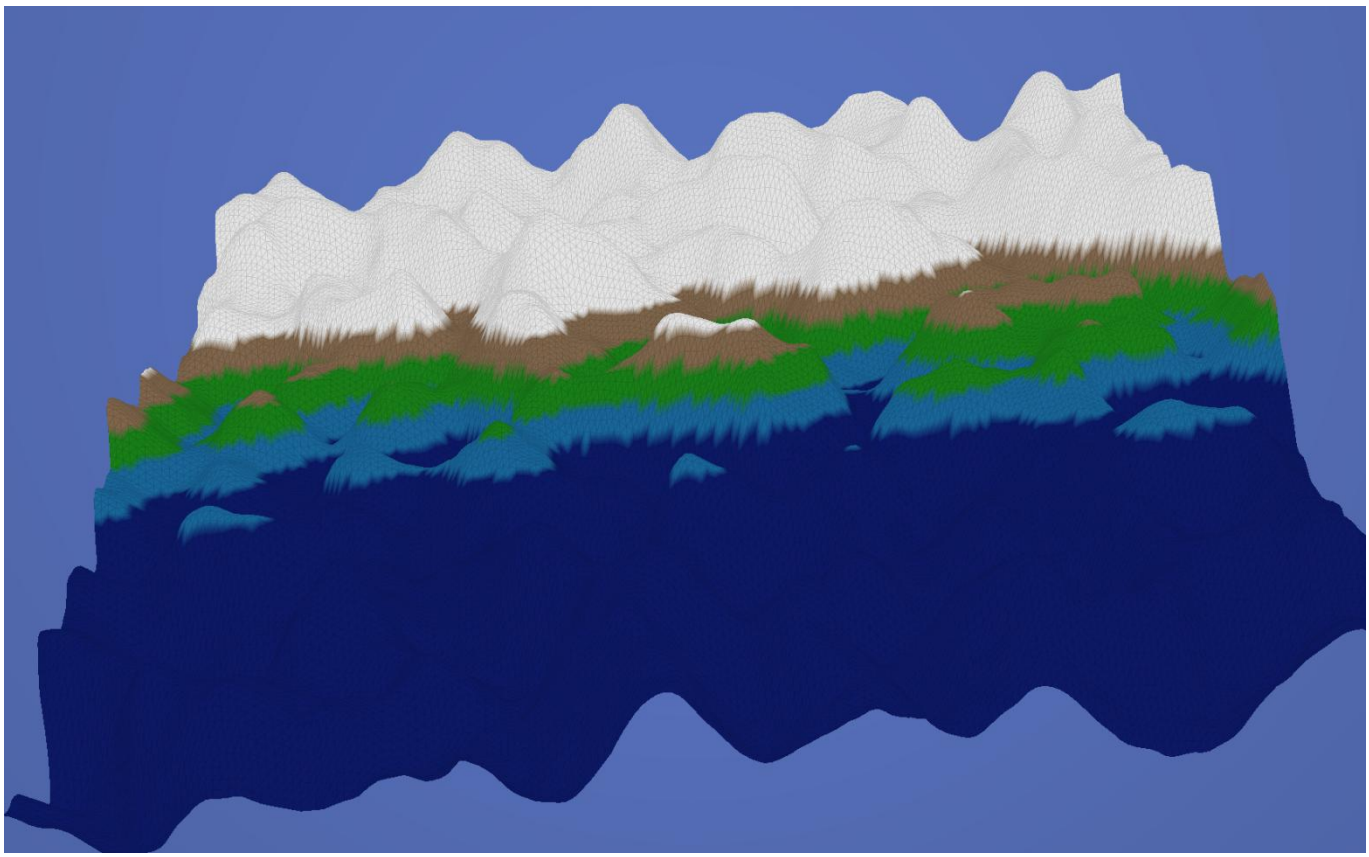
Sujet 4. Génération de terrain procédural (pas dans le sujet) :

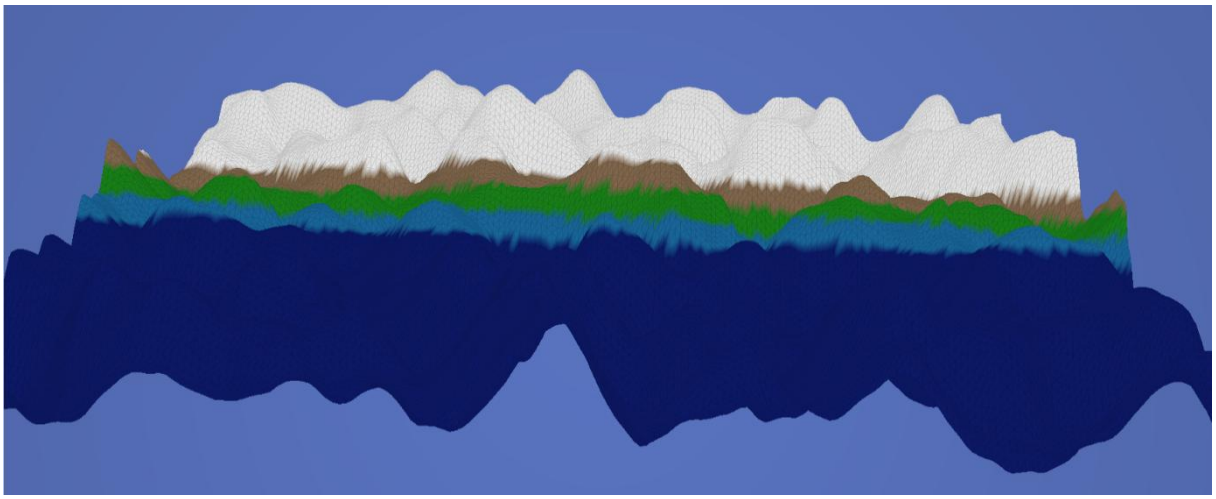
J'avais déjà par le passé fais de la génération de terrain procéduralement via fonction de bruit de perlin sur Unity, j'ai donc essayé de reproduire mon travail mais avec Qt, ce terrain est défini par une hauteur $h(x, y)$ calculée pour chaque point de la grille, ce qui donne un maillage de type *heightmap*.
Le maillage est ensuite coloré en fonction de l'altitude pour simuler différents biomes (eau, herbe, roche, neige).

Pour cette partie je me suis arrêter à la mais les idées ne manquent pas.. J'aurais bien voulu faire quelques fonctionnalités que j'avais fait sur unity comme le fait d'avoir des fondus de biome (moins abrupte), des dégradés de biomes aussi, il y a aussi un sujet intéressant que j'aurais pu rajouter c'est l'érosion du terrain (de mémoire on simule des gouttelettes d'eau qui vont venir shape le terrain d'une manière beaucoup plus réaliste)

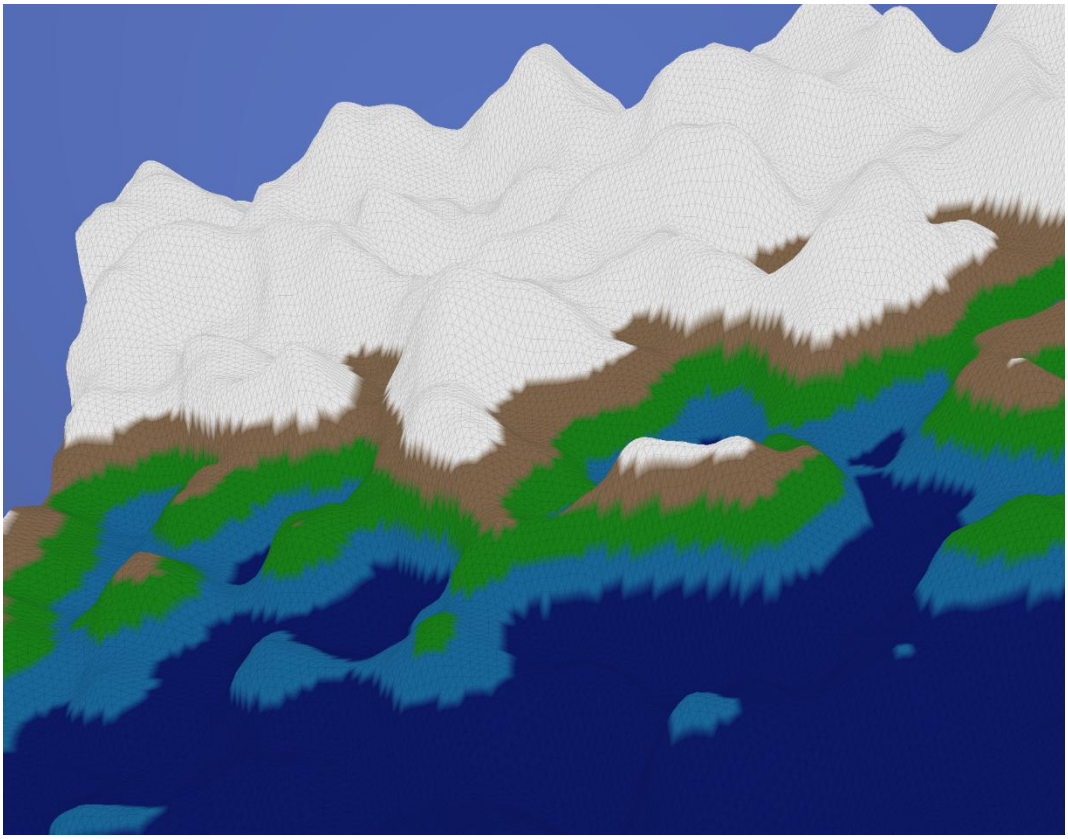
Dans les paramètres utilisées, on a l'amplitude qui augmente la hauteur globale, fréquence et octaves qui densifient les détails, warpStrength qui déforme les lignes naturelles pour casser la symétrie et heightBias déplace le niveau de mer (plus d'eau ou plus de terre), quelques-unes sont des paramètres qui me facilite la tâche car crée un terrain avec du bruit n'est pas aussi simple que juste coder les fonctions de bruit et la heightmap, il faut trouver des valeurs qui matchent bien entre elle pour avoir un terrain plus ou moins réaliste pour l'idée qu'on a, on peut pas mettre n'importe quoi et penser qu'on aura un truc superbe, du coup comme exemple j'ai pu produire ceux-ci avec Perlin2D :

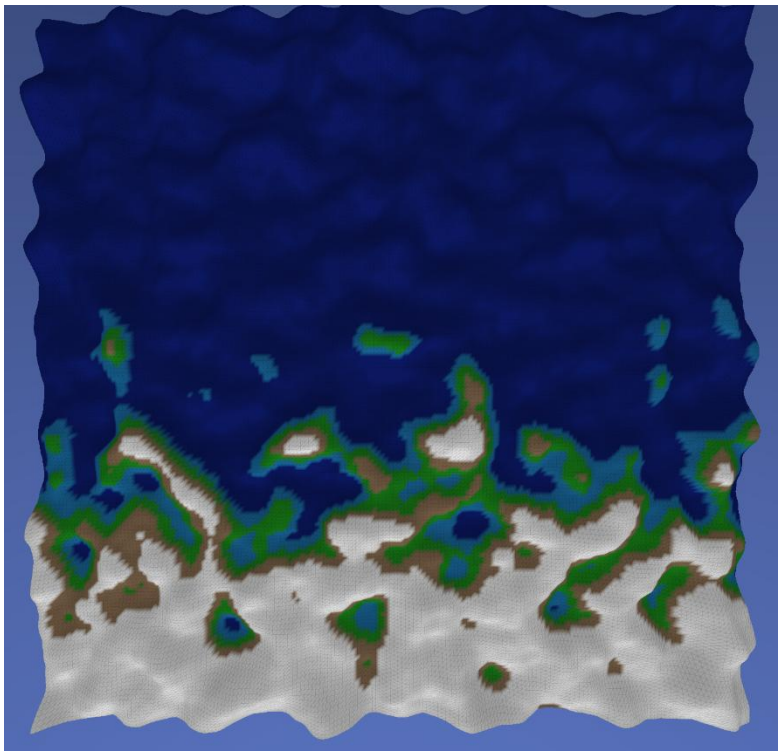




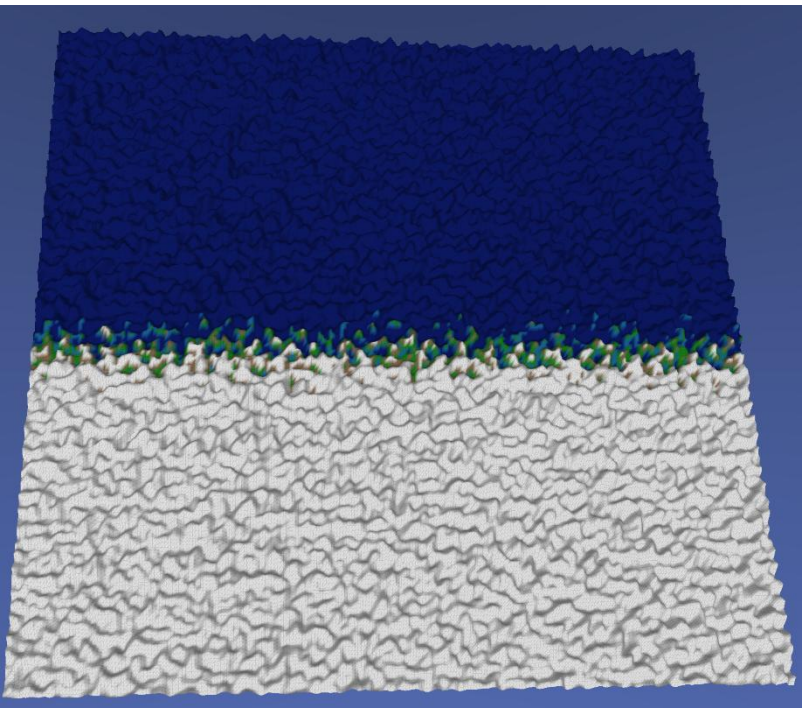


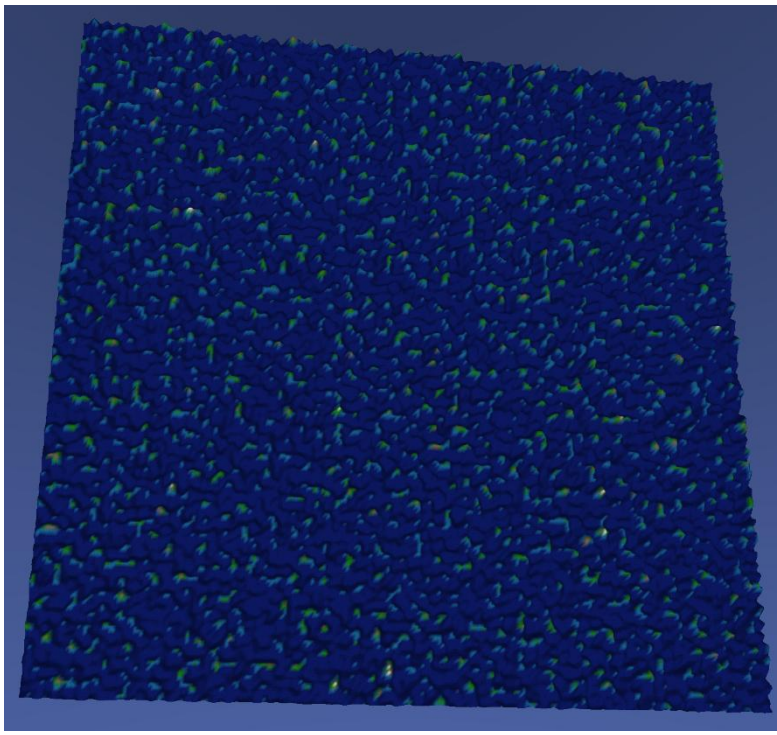
Sur cette capture-là, on voie bien les couches de couleur à chaque hauteur en y



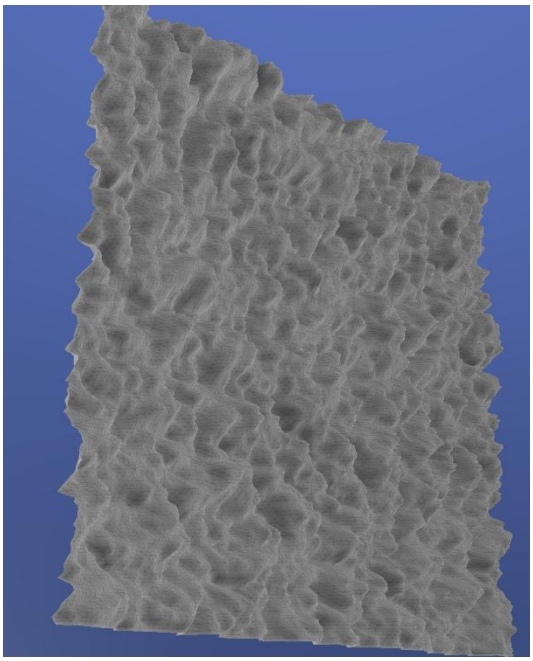


Changement de paramètres, on a beaucoup moins d'altitude haute ici.

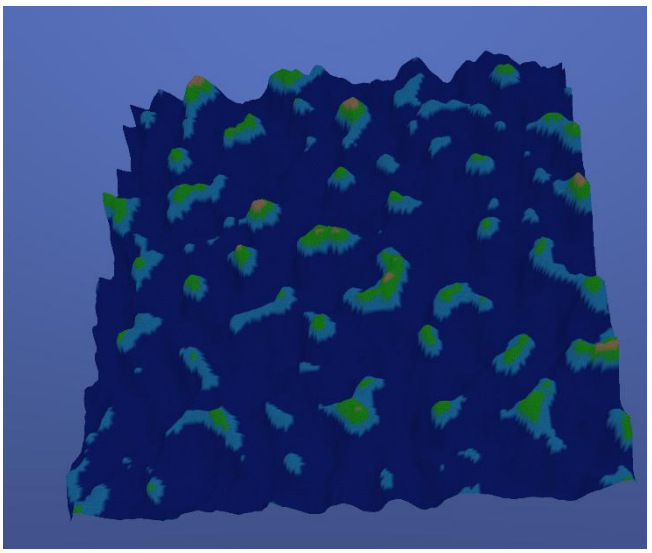


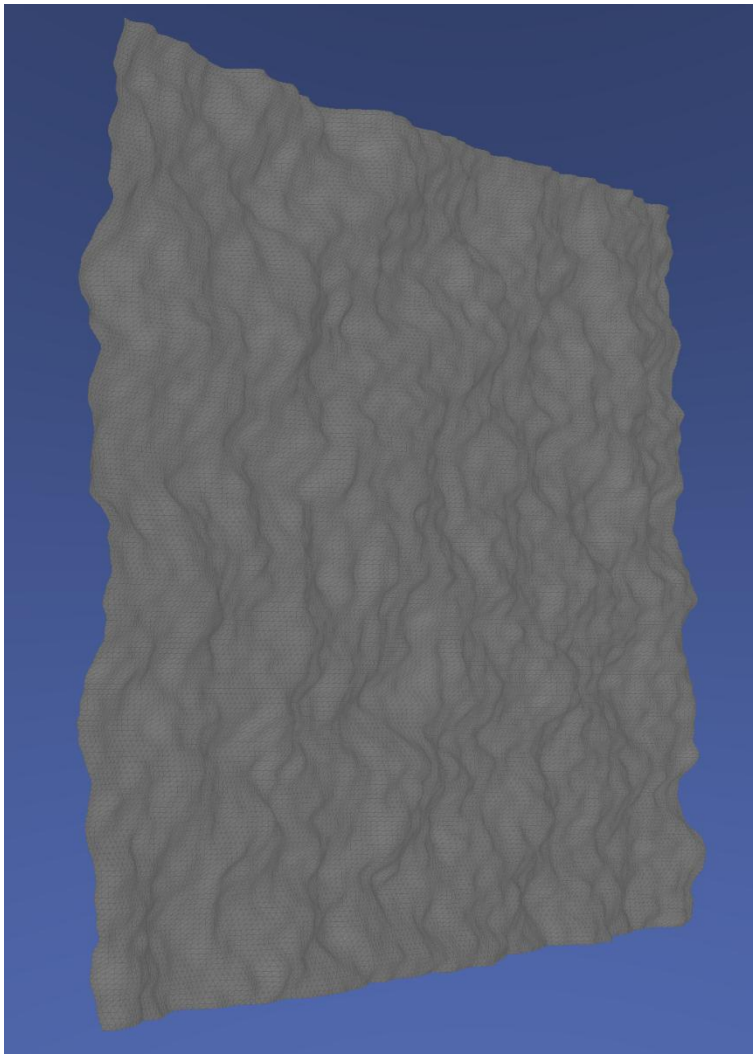


Ici on voit un fail, j'avais implémenté une fonction pour faire la rotation, sauf que vu que je m'y suis mal pris, le y du coup étais à l'envers et donnais les plages de couleur sur la largeur au lieu de la hauteur (je faisais un 180 degré en rotation), et j'utilise ValueNoise au lieu de perlin ici



Quelques exemples encore..





Plus doux..