



ÉCOLE CENTRALE LYON

INFO
RAPPORT

Projet de visualisation de données en réalité virtuelle

Élèves :
Victor ABRESOL
Khalid AISSI

Enseignant :
Romain VUILLEMOT

Table des matières

1	Introduction	2
2	L'environnement de travail	2
2.1	Unity3D	2
2.2	C#	2
2.3	SteamVR	2
3	Définition Générale	3
3.1	Les Couleurs	3
3.2	Niveau de gris	3
4	Structure de données	5
4.1	Les algorithmes de visualisation	6
4.2	Code pour générer un modèle 3D	7
4.2.1	Interface des paramètres	9
4.2.2	Paramètre 1 : nombre de billes	9
4.2.3	Paramètre 2 : Décalage	10
4.2.4	Paramètre 3 : Hauteur en fonction de couleurs	10
4.2.5	Paramètre 4 : Gris	10
4.2.6	Paramètre 5 : Offset	11
4.2.7	Paramètre 6 : Taille des billes	11
4.3	Depth map	12
4.3.1	Définition	12
4.3.2	Exemple	13
4.3.3	Utilité des depths maps	13
4.3.4	Exemple de génération d'image 3d a partir du depth map	13
4.4	Application en réalité virtuelle	14
4.4.1	Application en réalité virtuelle de l'image 2d en 3d	14
4.4.2	Programme pour interagir	14
4.4.3	Application en réalité virtuelle d'interaction de particules avec un terrain	14
4.5	Bibliothèque Mapbox	16
4.6	Gestion des particules avec Unity	20
4.7	Résultat	23
4.7.1	Résultat en faisant varier le rebond	23
4.7.2	Résultat en faisant varier Dampen	26
4.7.3	Résultat en faisant varier la forme	27
5	Conclusion	29

1 Introduction

Ce document constitue un rapport de projet d'option informatique réalisé par 2 élèves-ingénieurs à l'école Centrale de Lyon. L'objectif du projet est de tester des visualisation provenant d'objet 2D en 3D en réalité virtuelle. 2 approche ont réalisé dans ce projet. La première permet de réaliser un objet 3D à partir de n'importe quelle image 2D et de le visualiser en 3D. Mais cette approche ne permet aucune interaction. La seconde approche est l'utilisation d'un framework javascript pour récupérer un terrain depuis google map. Cette approche permet des interactions avec des particules en réalité virtuelle.

2 L'environnement de travail

2.1 Unity3D

Unity est un moteur de jeu multiplate-forme (smartphone, ordinateur, console de jeux vidéo et Web) développé par Unity Technologies. Il est l'un des plus répandus dans l'industrie du jeu vidéo, aussi bien pour les grands studios que pour les indépendants du fait de sa rapidité aux prototypages et qu'il permet de sortir les jeux sur tous les supports. Il a la particularité de proposer une licence gratuite dite « Personal » avec quelques limitations de technologie avancée au niveau de l'éditeur, mais sans limitation au niveau du moteur.



2.2 C#

Le logiciel à la particularité d'utiliser un éditeur de script compatible mono (C#). C# est un langage de programmation orienté objet, commercialisé par Microsoft depuis 2022 et destiné à développer sur la plateforme Microsoft.NET.

2.3 SteamVR

SteamVR est un ensemble de ressources disponibles pour la programmation de la réalité virtuelle avec Unity. Un certains nombre de "Prefabs" sont fournis, ce sont des objets Unity pré-construit qui permettent des interactions en réalité virtuelle comme la configuration des manettes. Lien : *[Cliquer ici](#)*



3 Définition Générale

3.1 Les Couleurs

Les couleurs en informatique fonctionnent par synthèses additives selon trois composantes : Rouge, Vert et Bleu. La synthèse additives correspond au fait que notre oeil perçoit une somme de lumière de couleur sous la forme d'une autre lumière de couleur. Concrètement une somme de couleur donne une nouvelle couleur :

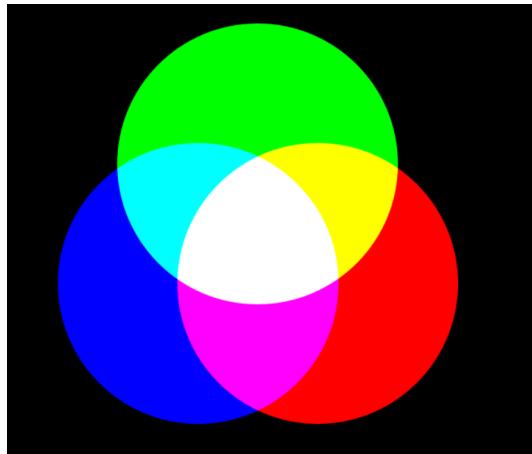


FIGURE 1 – Illustration de la synthèse additives des couleurs

Elles sont chacune codées sur 8 bits soit 256 possibilités pour chaque couleur. La valeur affectée à une couleur définit son intensité, si on attribut 255 à une couleur cela signifie que son intensité est maximum et 0 si elle est minimum. Une image est souvent représentée par un tableau de dimension 4 avec la couleur représentée par un vecteur de taille 3 (RGB) et la dernière dimension étant associée à la transparence.



FIGURE 2 – Illustration du format RGB

Cependant les couleurs peuvent aussi s'exprimer en Luminance et en chrominance (YUV). La luminance est calculée à partir des trois composantes de couleurs et elle permet de déterminer l'intensité de lumière, utile pour afficher une image en niveau de gris par exemple.

3.2 Niveau de gris

Dans des cas particuliers lors de ce projet, nous serons amené à travailler en niveau de gris, c'est pourquoi il est nécessaire de comprendre son fonctionnement :

Dans une image numérique, le niveau de gris représente la luminosité d'un pixel, lorsque les valeurs de ses composantes de couleur sont identiques. La plupart des formats de fichier image offrent un mode de reproduction en niveaux de gris, qui divise par trois le nombre d'octets nécessaire au codage. Cependant l'œil humain perçoit les trois composantes, rouge, vert et bleu d'une manière différente, c'est la synthèse additive des couleurs. L'humain voit donc une forte inégalité entre les trois couleurs : une lumière verte apparaît plus claire qu'une lumière rouge, et encore plus qu'une lumière bleue.

C'est ce que traduit l'équation de la Luminance :

$$Luminance = 0,2126Rouge + 0,7152Vert + 0,0722Bleu$$

Cette équation nous est donc utile pour reconstruire une image en niveau de gris.

4 Structure de données

Afin de visualiser des données avec Unity, nous allons utiliser un programme de visualisation déjà existant : [Lien github](#)

Ce programme permet d'afficher des données selon le format suivant :

- les trois premiers colonnes représentent les coordonnées spatiales de chaque pixels
- le 4ème colonne représente le rayon de la sphère qui représente chaque pixels
- les trois derniers colonnes représentent les coordonnées RGB de chaque pixels

	coordinate			radius	RGB		
	10	110	120		31	40	50
1	-6.23	-22.65	42.79	1.585	0.6	0.6	1↓
2	-6.87	-23.33	41.62	1.738	0.6	0.6	0.6↓
3	-6.87	-22.55	40.33	1.738	0.6	0.6	0.6↓
4	-6.43	-21.37	40.33	1.554	1	0.6	0.6↓
5	-7.30	-23.18	39.20	1.585	0.6	0.6	1↓
6	-7.26	-22.57	37.87	1.738	0.6	0.6	0.6↓
7	-5.83	-22.17	37.35	1.738	0.6	0.6	0.6↓
8	-4.85	-22.29	38.07	1.554	1	0.6	0.6↓
9	-5.72	-21.65	36.12	1.585	0.6	0.6	1↓
10	-6.70	-21.45	35.11	1.738	0.6	0.6	0.6↓
11	-6.26	-22.26	33.88	1.738	0.6	0.6	0.6↓
12	-6.90	-23.24	33.58	1.554	1	0.6	0.6↓
13	-5.13	-21.86	33.29	1.585	0.6	0.6	1↓
14	-4.59	-22.63	32.17	1.738	0.6	0.6	0.6↓
15	-3.17	-22.14	31.71	1.738	0.6	0.6	0.6↓
16	-3.08	-21.35	30.75	1.554	1	0.6	0.6↓
17	-2.03	-22.61	32.30	1.585	0.6	0.6	1↓
18	-0.75	-22.18	31.90	1.738	0.6	0.6	0.6↓
19	-0.34	-22.82	30.52	1.738	0.6	0.6	0.6↓

FIGURE 3 – Format des données prises en argument de l'algorithme

4.1 Les algorithmes de visualisation

Le programme contient trois sous-projets permettant l'affichage sous les 3 formats visuels suivant :

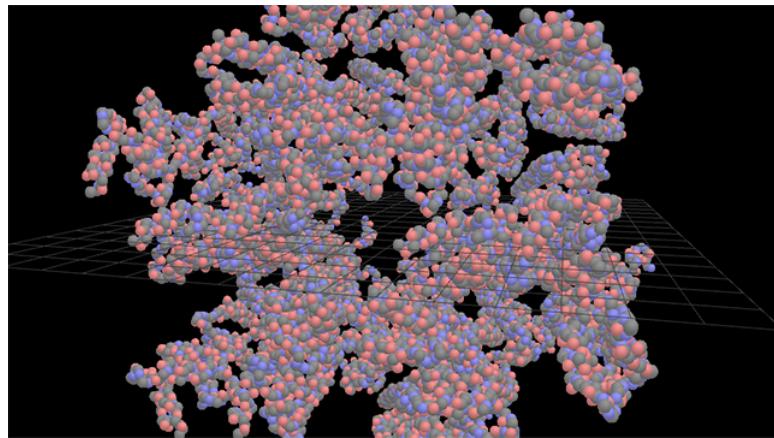


FIGURE 4 – script AsciiDataToParticle_Simple

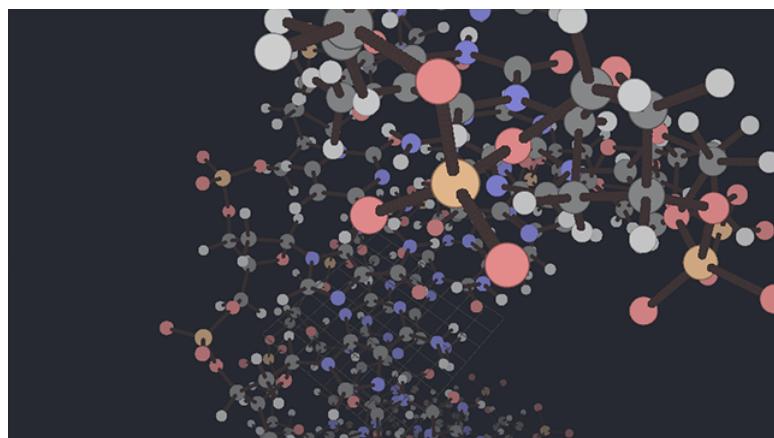


FIGURE 5 – script AsciiDataToParticle_add

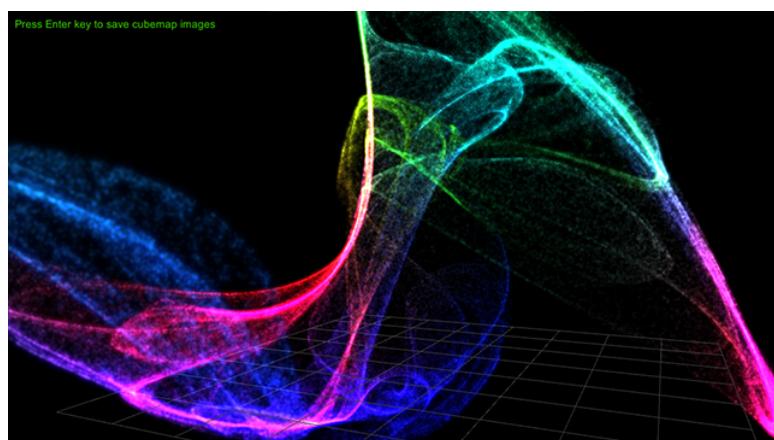


FIGURE 6 – script AsciiDataToParticle_Fog

4.2 Code pour générer un modèle 3D

Nous souhaitons tout d'abord récupérer les données pertinentes d'une image 2d sous un format texte de la forme vu dans la section structure des données. Pour cela, on a réalisé un premier programme python :

```

import matplotlib.image as mpimg
import numpy as np
img = mpimg.imread("image.png")

a = np.zeros(shape=(len(img)*len(img[0]),len(img[0][0])+3))
for i in range(0,len(img)) :
    for j in range(0,len(img[0])) :
        a[i*len(img[0])+j][3:] = img[i][j]
        a[i*len(img[0])+j][:3] = [i,j,0]
np.savetxt('file1.txt', a )

```

FIGURE 7 – Programme pour convertir un format PNG en texte

Désormais nous voulons réaliser un second programme afin d'interpréter ces données avec Unity et créer un modèle 3D à partir de ces données.

Dans ce but nous avons modifier le programme de visualisation Unity que nous avions repris.

Dans un premier temps nous initialisons des variables de gestion et permettons à l'utilisateur de pouvoir faire varier certaines de ces variables directement depuis une interface :

```

//Définition du paramétrage
public int nb_bille = 10; //Nombre de bille sur chaque droite verticale (y)
public int decalage_bille = 5; //Taille de l'espace entre 2 billes consecutives selon y
public int CoefRouge = 1; //Proportion de Rouge dans le positionnement suivant l'axe y
public int CoefVert = 0; //Proportion de Vert dans le positionnement suivant l'axe y
public int CoefBleu = -1; //Proportion de Bleu dans le positionnement suivant l'axe y
public bool grey = false; //Affichage selon le niveau de gris
public int offset = 0; // Espace entre le début de chaque gamme de couleur
public float taille = 1F; // Taille des billes

```

FIGURE 8 – Extrait du programme - Partie pour l'initialisation des variables

Ensuite on prépare un tableau qui vas contenir l'ensemble des informations liées aux billes (la couleur, la position, la taille). Ce tableau doit être de taille le nombre de couche multiplié par le nombre de ligne de données. Où le nombre de couche est donnée par l'utilisateur dans le champs NbBilles de l'interface paramètre :

```
// ----- Count data rows
int _particleCount = 0;
string[] _allLines = File.ReadAllLines(@_path);
for (int _i=0; _i<_allLines.Length; _i++)
    if (_allLines[_i].Length > 0)
        //On prépare un tableau de la taille nombre de billes*nombre de lignes de données
        _particleCount += nb_bille;
```

FIGURE 9 – Extrait du programme - Partie pour la construction des données

Nous faisons ensuite un algorithme qui suit le fonctionnement suivant :

1. Le paramètre "gris" est-il coché ?
2. Si oui -> On construit la position verticale selon l'équation du niveau de gris et on modifie aussi chaque coordonnées RGB par cette équation afin d'afficher l'image en gris si elle ne l'est pas déjà
3. Si non -> On construit la position verticale selon l'équation des couleurs définie par :

$Z = [\text{CoefRouge} * \text{pixelRouge} + \text{CoefBleu} * \text{pixelBleu} + \text{CoefPixelVert}]$ où les coefficients sont donnés en paramètre par l'utilisateur.

```
// ----- store data in ATOM structure array
int _k = 0;
for (int _i = 0; _i < _allLines.Length; _i++) // On fait une boucle pour chaque ligne de données.
{
    for (int _j = 0; _j < nb_bille; _j++) // Puis On fait une boucle pour chaque couche (nb_bille)
    {
        if (_allLines[_i].Length > 0)

            if (grey == false) // si l'option gris n'est pas choisie alors :
            {
                string[] _data = _allLines[_i].Split(); // On construit la position verticale selon la couleur et les composantes rentrées en paramètres :
                _Atoms[_k].position = new Vector3(float.Parse(_data[0]), (CoefRouge * float.Parse(_data[3]) + CoefVert * float.Parse(_data[4]) + CoefBleu *
                _Atoms[_k].radius = taille; // Permet de définir la taille de la bille
                _Atoms[_k].rgb = new Color(float.Parse(_data[3]), float.Parse(_data[4]), float.Parse(_data[5]), 1F);
                _k++;
            }
            else // Sinon :
            {
                string[] _data = _allLines[_i].Split(); // On construit la position verticale selon l'équation de gris :
                _Atoms[_k].position = new Vector3(float.Parse(_data[0]), (0.7152f * float.Parse(_data[4]) + 0.0722f * float.Parse(_data[5]) + 0.2126f * float.P
                _Atoms[_k].radius = taille; // On modifie aussi la couleur pour l'afficher toujours en niveau de gris :
                _Atoms[_k].rgb = new Color(0.7152f * float.Parse(_data[4]) + 0.0722f * float.Parse(_data[5]) + 0.2126f * float.Parse(_data[3]), 0.7152f *
                _k++;
            }
    }
}
```

FIGURE 10 – Extrait du programme - Partie pour la construction des billes

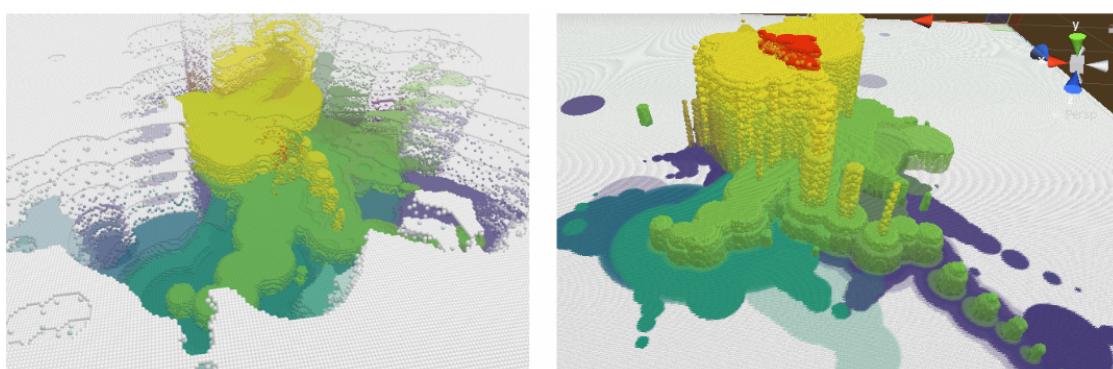


FIGURE 11 – Exemples d'images de sortie de l'algorithme

4.2.1 Interface des paramètres

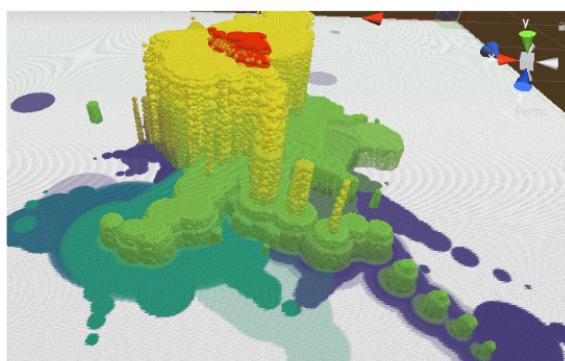
Cette interface nous permet de changer les différents paramètres depuis l'interface de Unity

Nb_bille	10
Decalage_bille	1
Coef Rouge	1
Coef Vert	0
Coef Bleu	-1
Grey	<input type="checkbox"/>
Offset	0
Taille	0.5

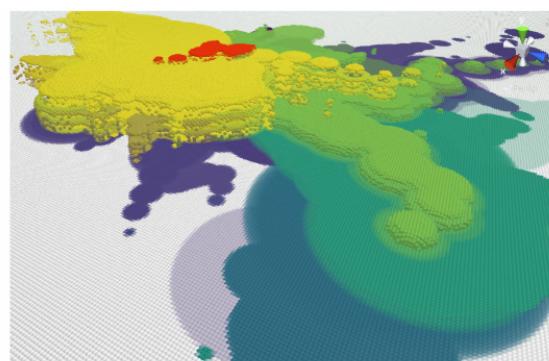
FIGURE 12 – Interface des paramètres

4.2.2 Paramètre 1 : nombre de billes

Le nombre de billes c'est le nombre de couches construites à partir chaque pixel de l'image 2D de base



Nombre de billes = 10

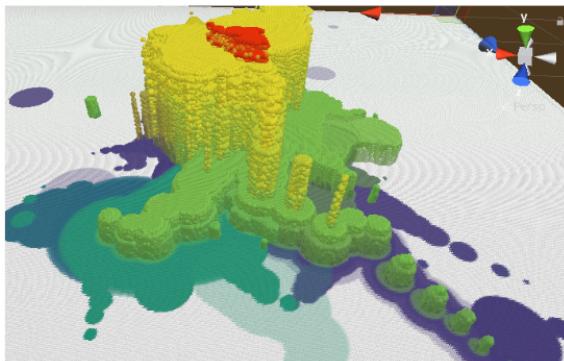


Nombre de billes = 3

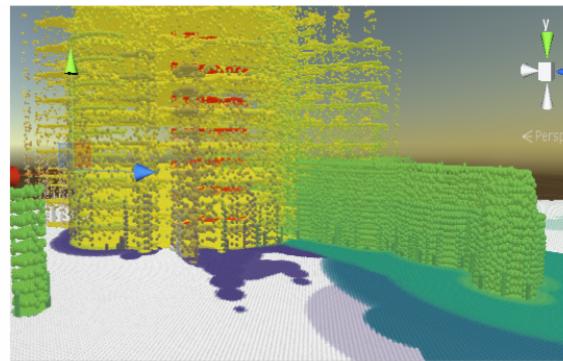
FIGURE 13 – Illustration de l'influence du paramètre "nombre de billes"

4.2.3 Paramètre 2 : Décalage

Le décalage c'est l'espacement entre les couches



Décalage = 5



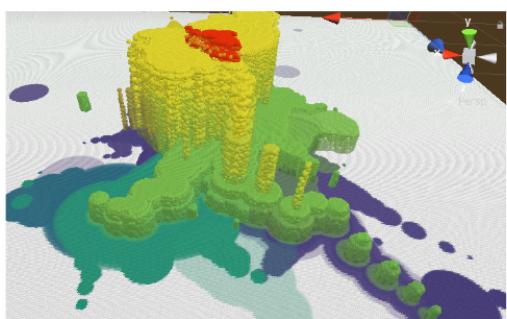
Décalage = 15

FIGURE 14 – Illustration de l'influence du paramètre "Décalage"

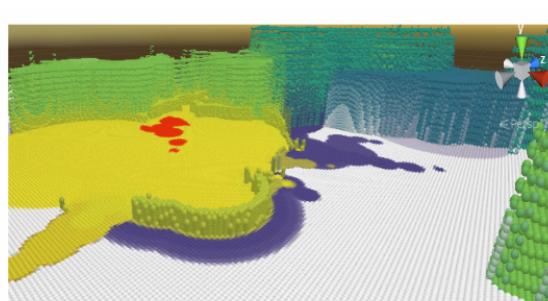
4.2.4 Paramètre 3 : Hauteur en fonction de couleurs

Dans l'image 2D , chaque pixel a 2 coordonnées x et y , et pour avoir une visualisation 3D on ajoute un autre coordonnée z qu'on calcule en fonction des intensités des couleurs selon l'équation suivante :

$$Z = \text{Coef Rouge} * \text{Rouge} + \text{Coef Vert} * \text{Vert} + \text{Coef Bleu} * \text{Bleu}$$



(1,0,-1)



(-1,1,0)

FIGURE 15 – Illustration de l'influence du paramètre "Hauteur selon les couleurs"

4.2.5 Paramètre 4 : Gris

la case gris permet de affecter le niveau de gris de chaque pixel à la couleur de la bille correspondante

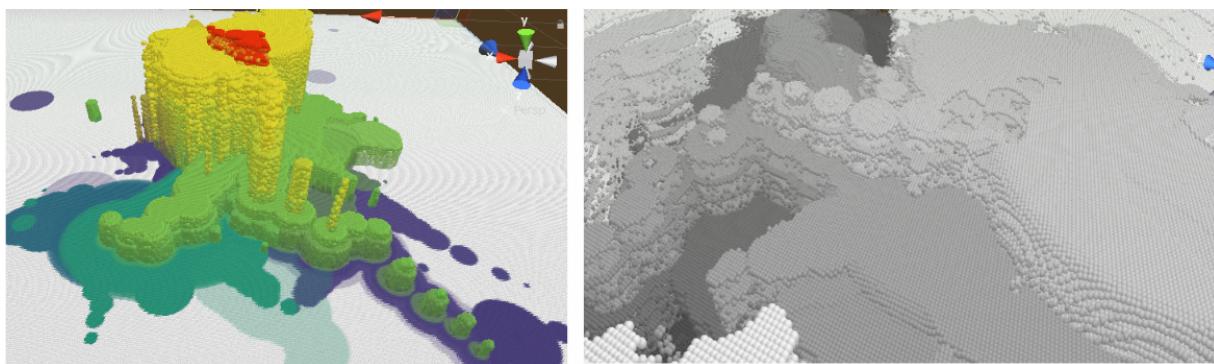


FIGURE 16 – Illustration de l'influence du paramètre "Gris"

4.2.6 Paramètre 5 : Offset

L'offset permet de faire un décalage entre chaque couleur : chaque couleur à un niveau d'affichage dans l'espace

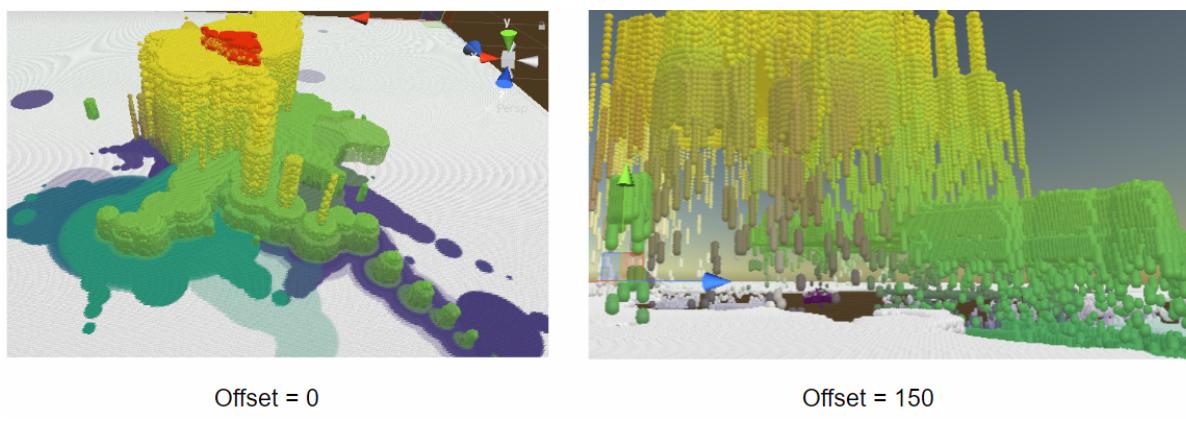


FIGURE 17 – Illustration de l'influence du paramètre "Offset"

4.2.7 Paramètre 6 : Taille des billes

Taille de bille c'est le rayon des billes

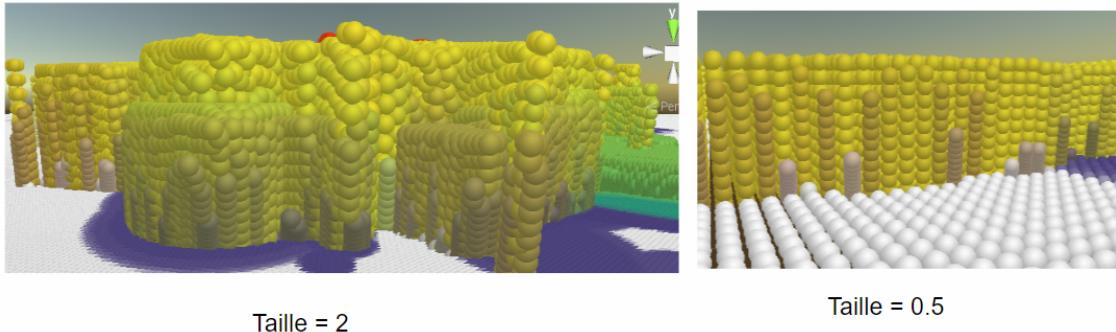


FIGURE 18 – Illustration de l'influence du paramètre "Taille des billes"

Enfin pour finir voici un exemple de l'utilisation du programme sur une map 2D avec des paramètres bien adaptés. Nous avons choisi un relief selon l'intensité soit la somme

des trois composantes de couleurs.

On prend comme image la suivante :



FIGURE 19 – Map en image 2D

Et on obtient le résultat suivant :

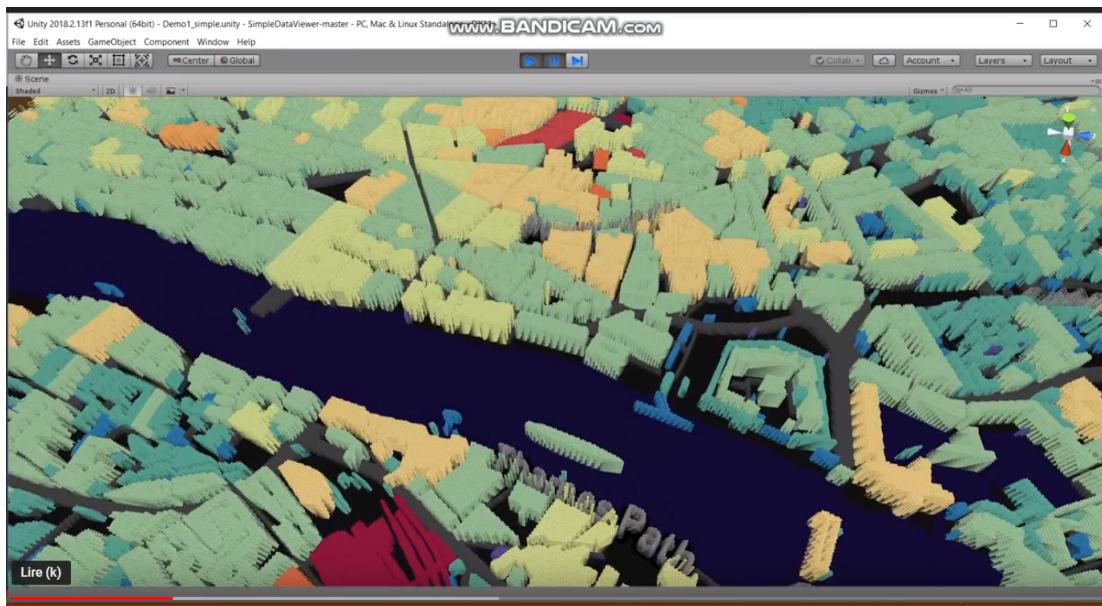


FIGURE 20 – Map en modèle 3D

4.3 Depth map

4.3.1 Définition

une carte de profondeur est une image ou un canal d'image contenant des informations relatives à la distance des surfaces des objets de la scène par rapport à un point de vue.

4.3.2 Exemple



Structure cubique

Carte de profondeur:
Plus proche est plus
sombre

Carte de profondeur:
plus le plan focal est
sombre

FIGURE 21 – Exemple de depth map

4.3.3 Utilité des depths maps

Le depth map fournit les informations de distance nécessaires à la création et à la génération d'une image ou carte 3D à partir une image ou carte 2D

4.3.4 Exemple de génération d'image 3d a partir du depth map

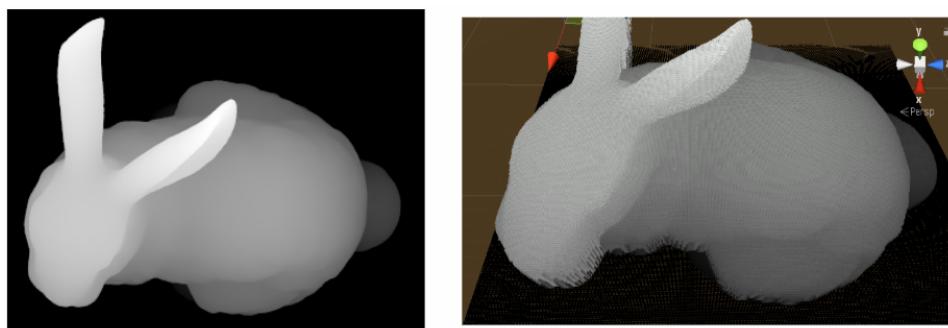


FIGURE 22 – Exemple de génération d'image 3d à partir d'une depth map

4.4 Application en réalité virtuelle

4.4.1 Application en réalité virtuelle de l'image 2d en 3d

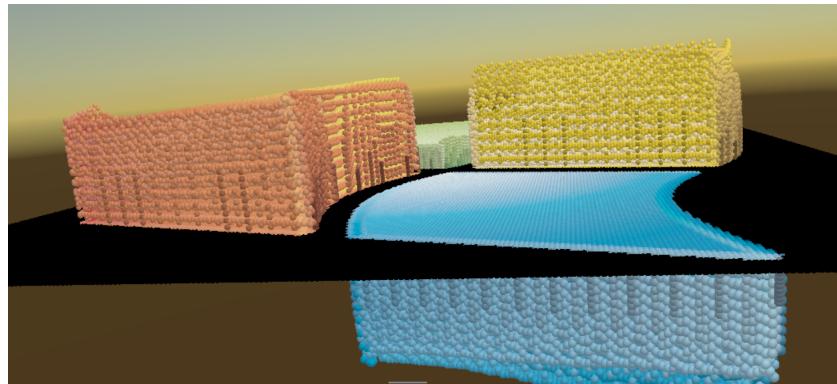


FIGURE 23 – Logo Windows en réalité virtuelle

4.4.2 Programme pour interagir

```

public float speed = 5;
public int hauteur = 0;
private Rigidbody rb;
public Transform myChildObject;
public Transform myParentObject;
public bool detachChild = false;
public ParticleSystem particles;
void Start() {
    rb = GetComponent<Rigidbody>();
}
void FixedUpdate() {
    hauteur = 0;
    float moveHorizontal = Input.GetAxis("Horizontal");
    float moveVertical = Input.GetAxis("Vertical");
    if (Input.GetKey(KeyCode.X)) {
        hauteur = -1;
    }
    if (Input.GetKey(KeyCode.Z)) {
        hauteur = 1;
    }
    Vector3 movement = new Vector3(moveHorizontal, hauteur, moveVertical);
    transform.Translate(movement * Time.deltaTime * speed);

    if (Input.GetKey(KeyCode.M)) {
        particles.Pause();
        myChildObject.parent = null;
    }
    if (Input.GetKey(KeyCode.N)) {
        myChildObject.transform.parent = myParentObject.transform;
        myChildObject.transform.localPosition = new Vector3(0, 0, 0);
        myChildObject.transform.localRotation = new Quaternion(0, 0, 0, 0);
        particles.Play();
    }
}

```

FIGURE 24 – Extrait de la vidéo d'interaction de particules avec un terrain quelconque

4.4.3 Application en réalité virtuelle d'interaction de particules avec un terrain

Nous avons souhaité réaliser des interactions de particules avec un terrain afin de bénéficier de la réalité virtuelle d'une façon innovante.

Pour cela nous voulons pouvoir simuler un certains nombre de paramètre telle que la gravité ou le relief du terrain mais aussi la visibilité des particules et le contrôle de leurs flux.

Pour cela nous avons créé dans un premier temps un terrain quelconque sur lequel on envoie un élément de particules de Unity.

Vous pouvez voir le résultat dans la vidéo suivante : [Cliquer ici](#)

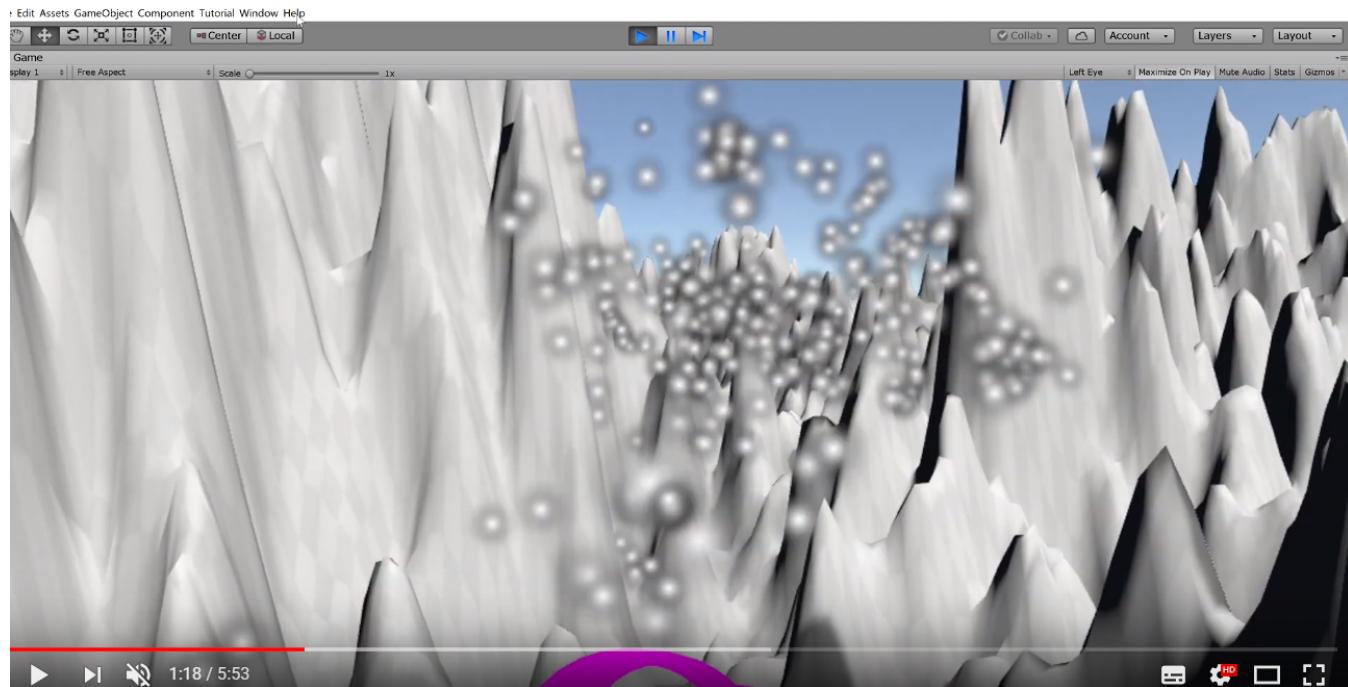


FIGURE 25 – Extrait de la vidéo d’interaction de particules avec un terrain quelconque

On constate que le programme fonctionne mais n'est pas pertinent, on est capable de se déplacer dans un terrain et de jouer sur les particules mais il est difficile de prédire le comportement des particules, ainsi il ne nous permet pas de valider notre modèle de programme.

Pour avoir une idée plus précise de la précision, nous souhaitons par la suite construire un terrain de forme sinusoïdale pour prédire le mouvement attendus des particules.

On peut voir le résultat dans la vidéo youtube suivante :

[Cliquer ici](#)

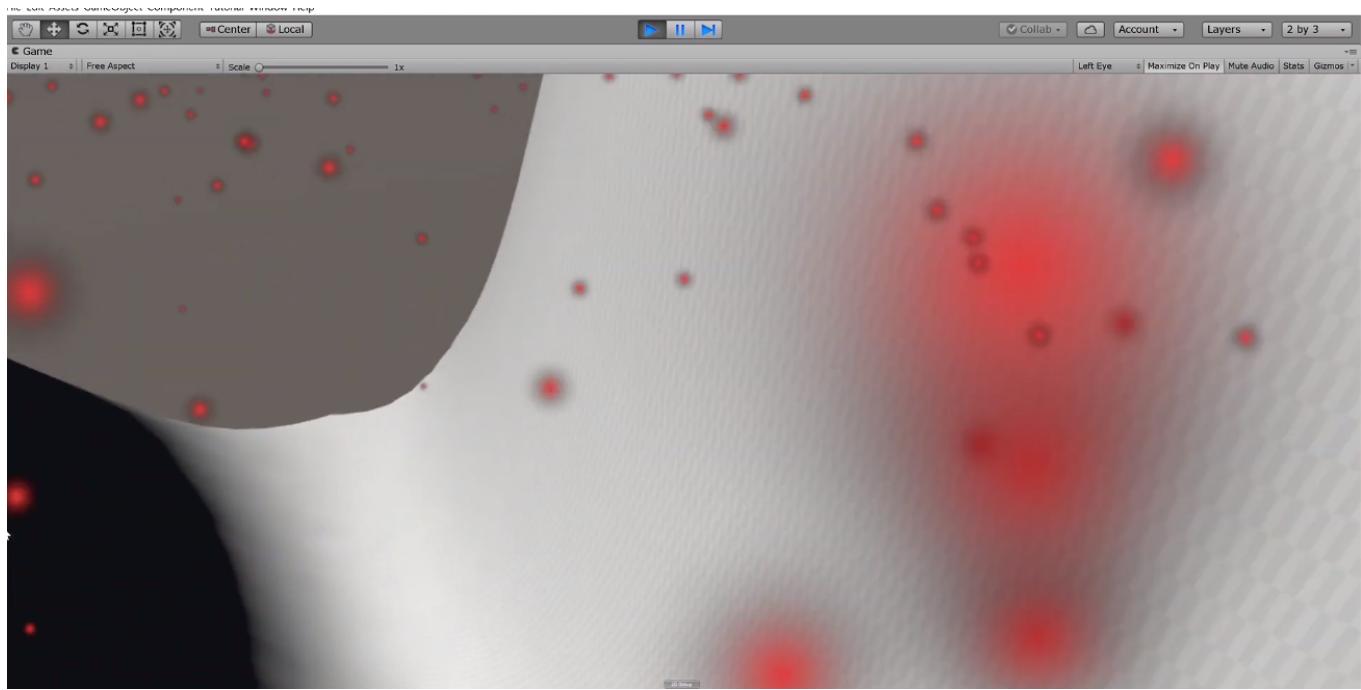


FIGURE 26 – Extrait de la vidéo d’interaction de particules avec un terrain de forme sinusoïdale

On constate cette fois-ci que le résultat est parlant. Le comportement des particules est bien celui attendu. Ce qui pourrait-être intéressant c'est de garder en vue la trajectoire des particules et d'appliquer les interactions sur un terrain réel provenant de google map. C'est donc ce que nous avons fait par la suite.

4.5 Bibliothèque Mapbox

Dans cette section nous allons vous présenter l'intégration de terrain réel dans Unity. Mapbox.js est bibliothèque javascript basée sur Leaflet (Leaflet est une bibliothèque JavaScript libre de cartographie en ligne) et le langage de feuille de style CartoCSS. Dans cette partie on a utilisé Mapbox pour générer une carte de terrain 3D qu'on a utilisé ensuite sur Unity avec le programme d'interaction de particules en réalité virtuelle. Maintenant on a créé une nouvelle scène vide et importer les packages de MapBox comme le montre la figure suivante :

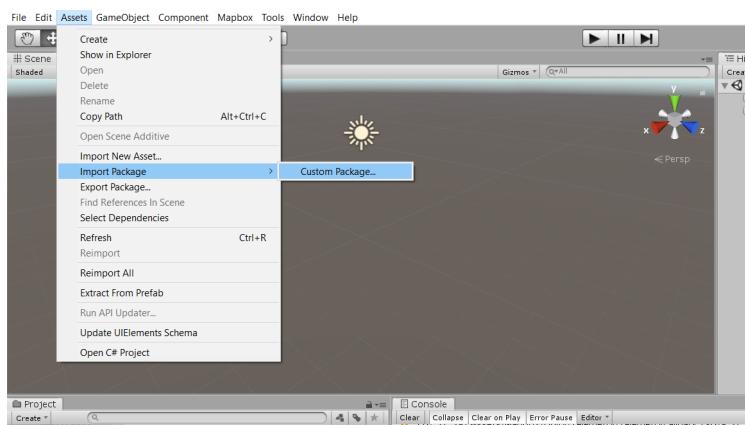


FIGURE 27

Ensuite on aura besoin d'une Objet Map pour afficher une carte , pour ce faire il faut le glisser dans la scène depuis MapBox->Prefabs->Map comme le montre la figure suivante :

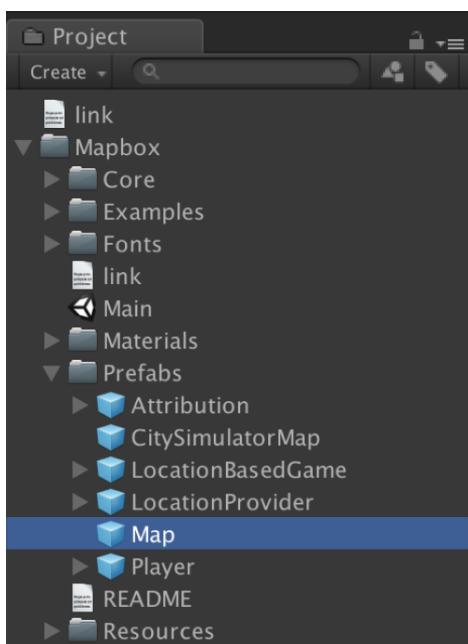


FIGURE 28

Cet objet Map est livré avec un Abstract-Map script dans lequel on a ajusté les paramètres de la carte tels que l'emplacement, le style de la carte, l'altitude, etc. Donc on a sélectionné notre objet Map et dans la fenêtre Inspecteur. Sous les Abstract Map paramètres généraux, on a modifié l'emplacement en 45.374218, -121.688341 (on peut soit chercher un emplacement précis avec son nom , soit mettre ses coordonnées qu'on peut récupérer depuis Google Maps), ensuite on coche la case "Snap Map to Zero".

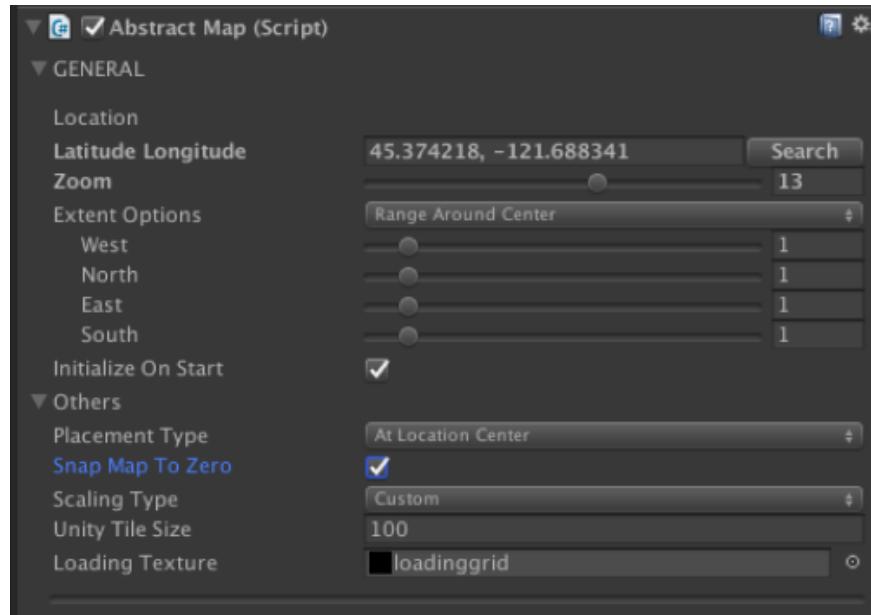


FIGURE 29

Ensuite dans les paramètres de l'image on remplace le "Data Source" par «Mapbox Satellite». Et Enfin, on ouvre les paramètres du terrain et on change le "Elevation Layer Type" en «Terrain with Elevation».

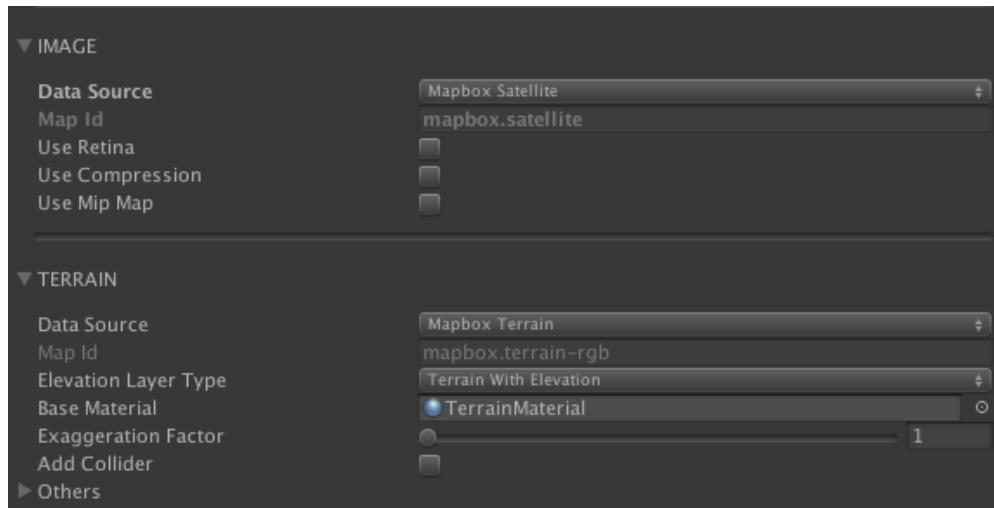


FIGURE 30

Maintenant on enregistre notre scène et on clique sur "Lire" et on aura la carte suivante :

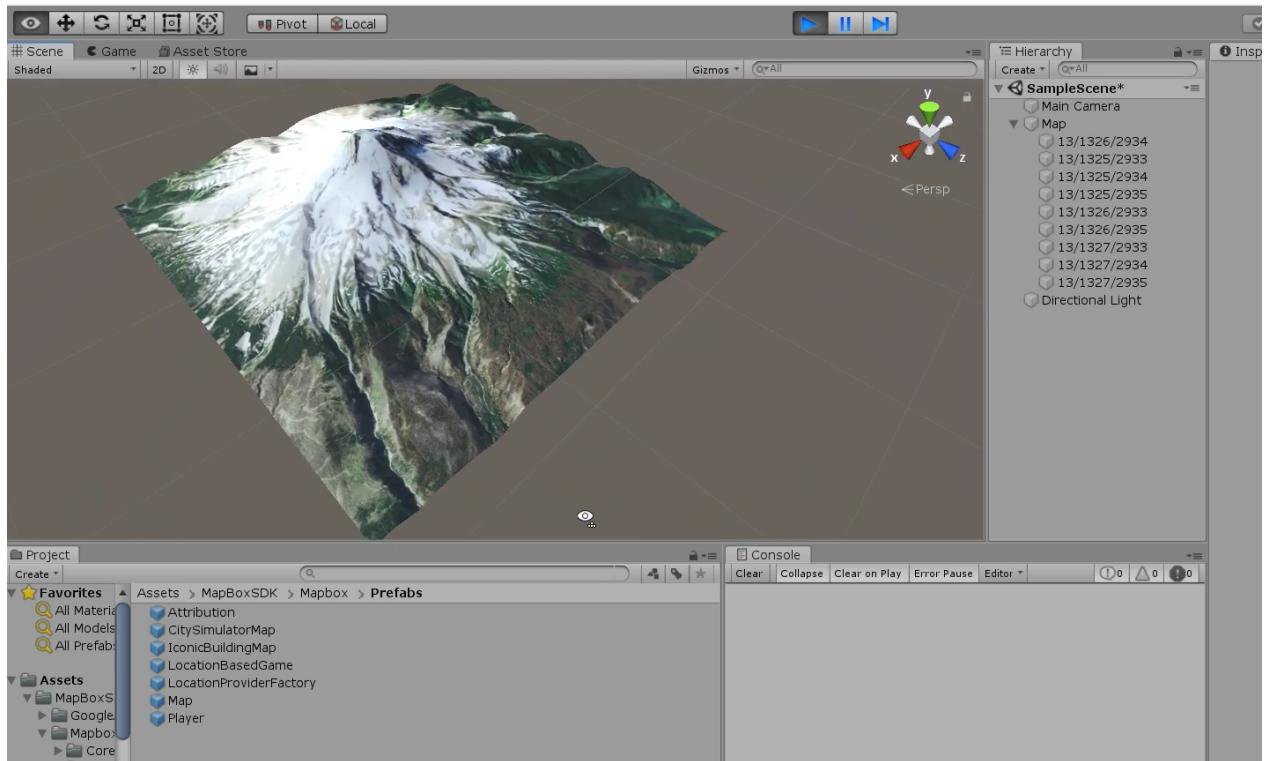


FIGURE 31

A partir du programme réalisé précédemment, nous allons l'exporter dans le programme d'interaction de particules en réalité virtuelle afin de combiner les deux résultats. Ainsi nous obtenons le résultat suivant :

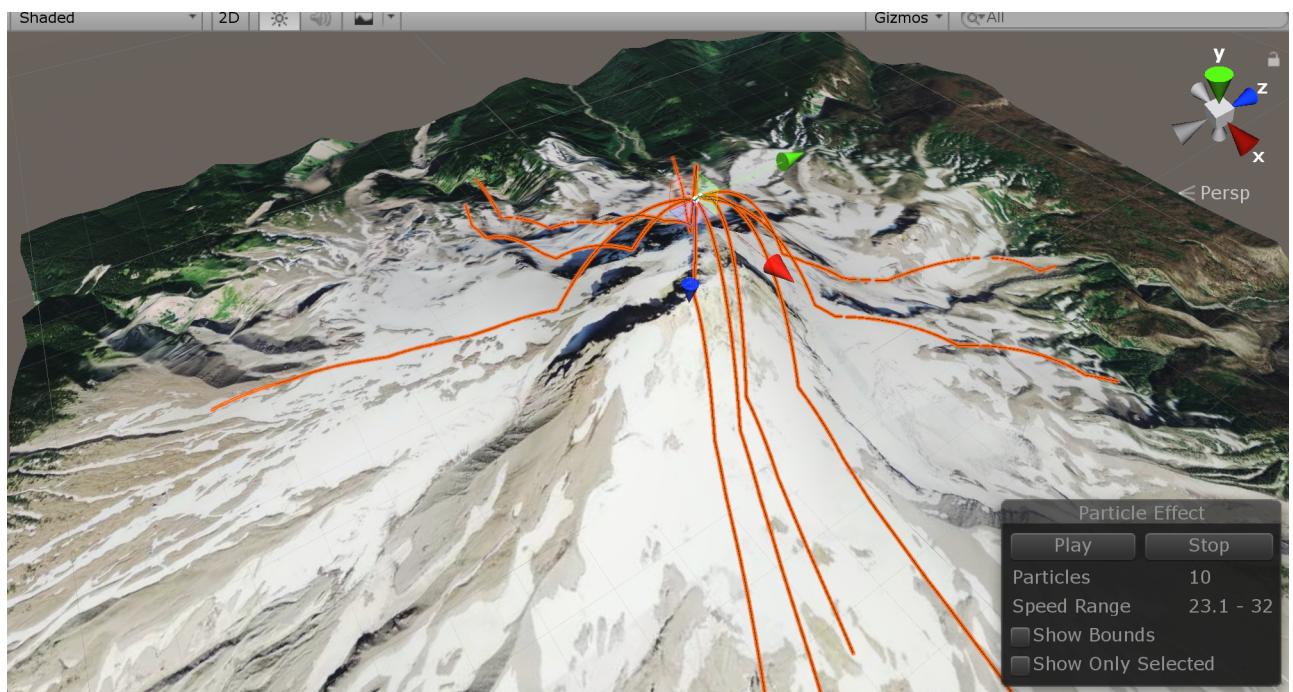


FIGURE 32 – Visualisation

On constate que le résultat obtenu est parlant et utile pour anticiper des mouvements

de particules sur un terrain réel.

4.6 Gestion des particules avec Unity

Pour la gestion des particules avec Unity, nous avons utilisé 5 tables de Unity. La première table permet de régler des caractéristiques générales sur les particules :

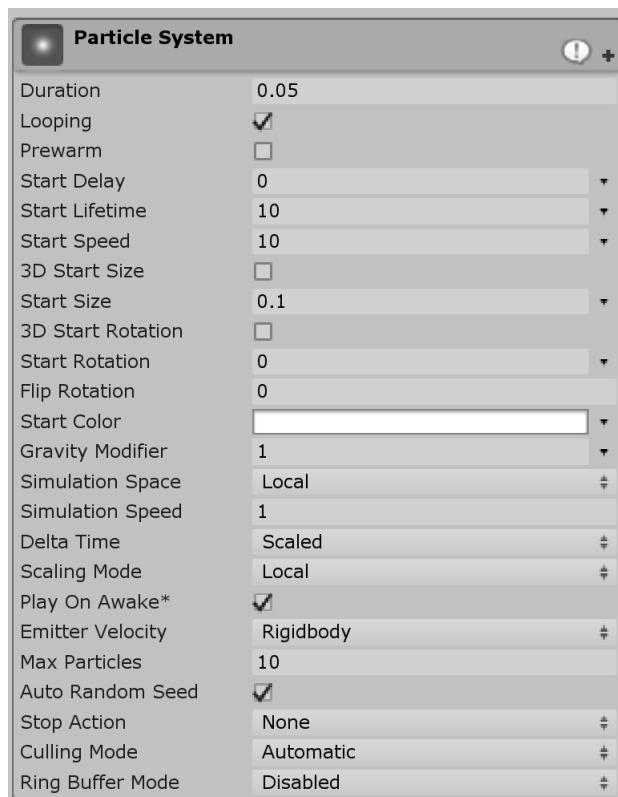


FIGURE 33 – Table de particules - Particle System

- **Duration** : Définit le temps de réapparition des particules après leur destruction.
- **Looping** : Booleen définissant si on exécute le flux de particules une seul fois ou à l'infini (en boucle).
- **Start Lifetime** : La durée de vie d'une particule
- **Start Speed** : La vitesse d'une particule
- **Start Color** : Couleur des particules
- **Gravity Modifier** : Quelle gravité appliquons nous aux particules. Si la valeur est nulle alors les particules ne sont pas affectées par la gravité.
- **Max Particles** : Nombre de particules maximum coexistant dans une scène.

La seconde et troisième table que nous utilisons permettent respectivement de définir les caractéristiques de l'émission et de la géométrie des particules :

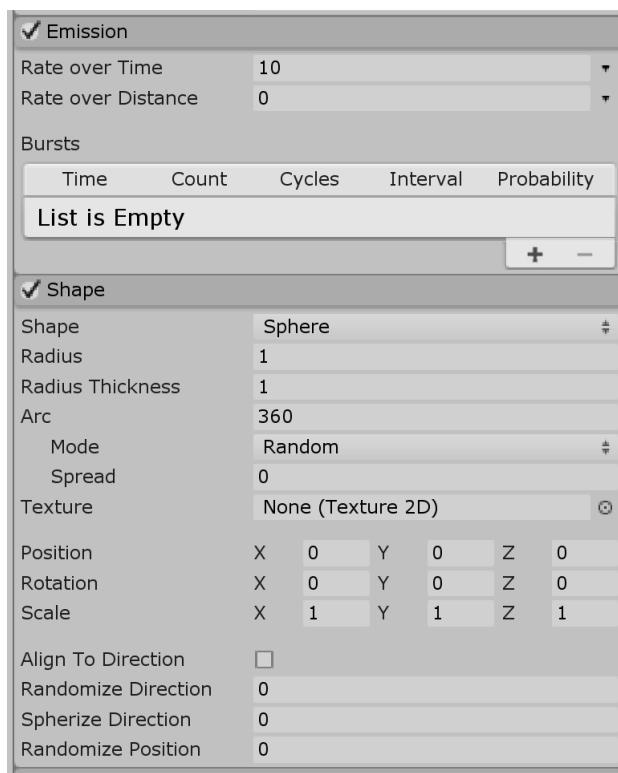


FIGURE 34 – Table de particules - Émission

- **Rate over time** : Nombre de particules par seconde
 - **Shape** : forme des particules
 - **Radius** : Rayon des particules
 - **Position** : position par rapport à l'origine du système de particules
 - **Rotation** : rotation par rapport à l'origine du système de particules
 - **Scale** : Taille d'une particule par rapport à la taille du système de particules
- La troisième table nous permet de gérer le type de collision :

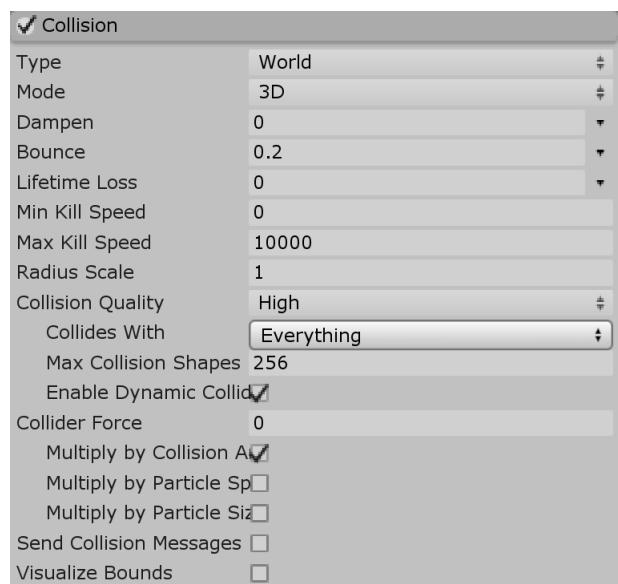


FIGURE 35 – Table de particules - Collision

- Type : Avec quelle système les particules peuvent-elles interagir
 - Dampen : Intensité de l'attachement des particules aux collisions
 - Bounce : Intensité du rebond des particules
- La quatrième table permet notamment de garder le trajet des particules visuellement :

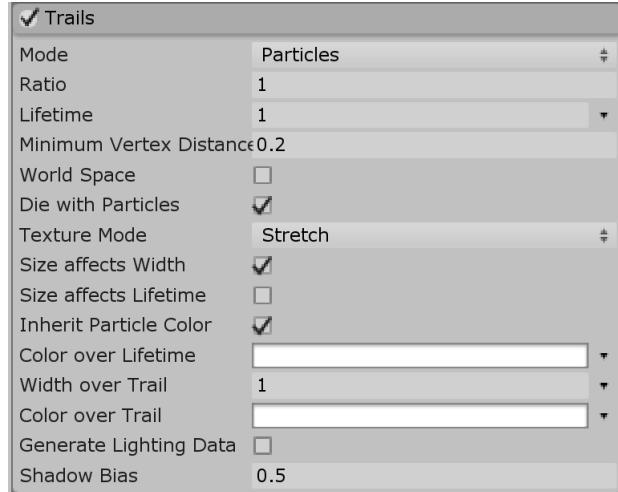


FIGURE 36 – Table de particules - Trails

- **Ratio** : Proportion de particules dont on va voir le trajet
- **Lifetime** : Temps de vie relatif à celui des particules
- **Die With Particles** : Boolean pour choisir si le trajet disparaît quand la particule disparaît
- **Size affect Width** : Boolean pour choisir si la taille des particules affectent proportionnellement ou non l'épaisseur du trajet affiché.
- **Size affect Lifetime** : Boolean pour choisir si la taille des particules affectent proportionnellement ou non la durée de vie du trajet affiché.
- **Color over trail** : Couleur du trajet

Enfin la dernière table nous permet de gérer le rendu du flux de particules :

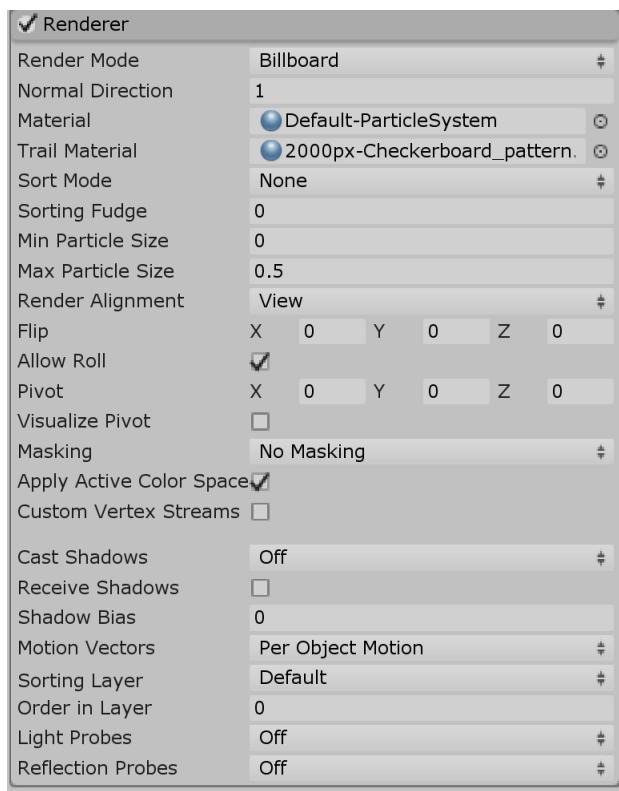


FIGURE 37 – Table de particules - Renderer

- **Render Mode** : Quelle type d'objet unity est une particule du flux
- **Normale direction** : il permet d'orienter le flux de particules
- **Materiaux** : Permet d'attribuer une texture détaillée à une particule
- **Trail Material** : Permet d'attribuer une texture détaillée à une trajectoire de particules
- **Min et Max Particle Size** : Échelle à laquelle on peut observer une particule
- **Cast Shadow** : Possède une ombre ou non

4.7 Résultat

Dans cette partie, on va s'intéresser aux résultats obtenus en réalité virtuelle pour l'interaction d'un flux de particule avec un terrain "réel" selon différents paramètres. Il est important de noter que les observations sont bien plus nettes en réalité virtuelle, et qu'une image ne permet pas de voir le déplacement, la vitesse et l'accélération. Cependant elle peut déjà donner une bonne idée de ce qui est observable.

4.7.1 Résultat en faisant varier le rebond

Dans un premier temps on regarde la variation en fonction du rebond.

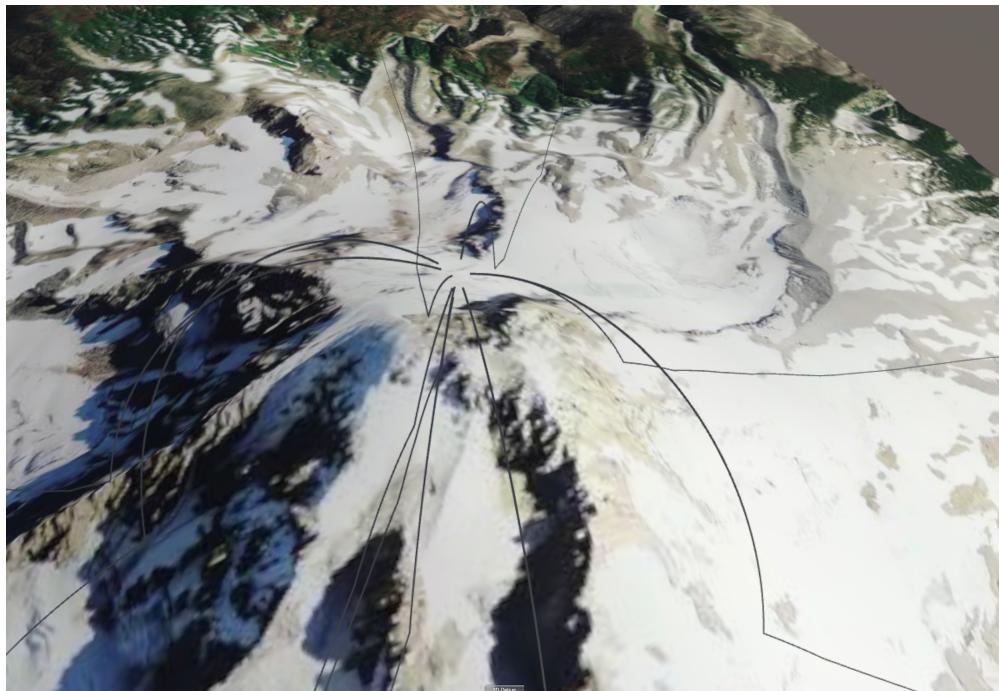


FIGURE 38 – Forme Sphere - Dampen 0 - Bounce 0.2

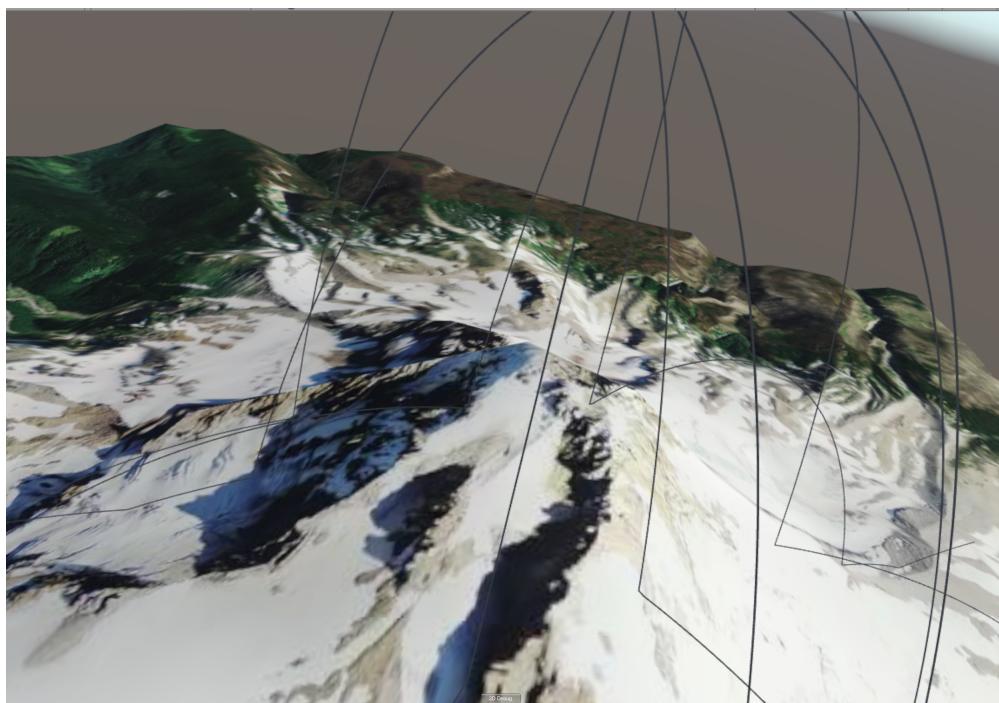


FIGURE 39 – Forme Sphere - Dampen 0 - Bounce 0.5

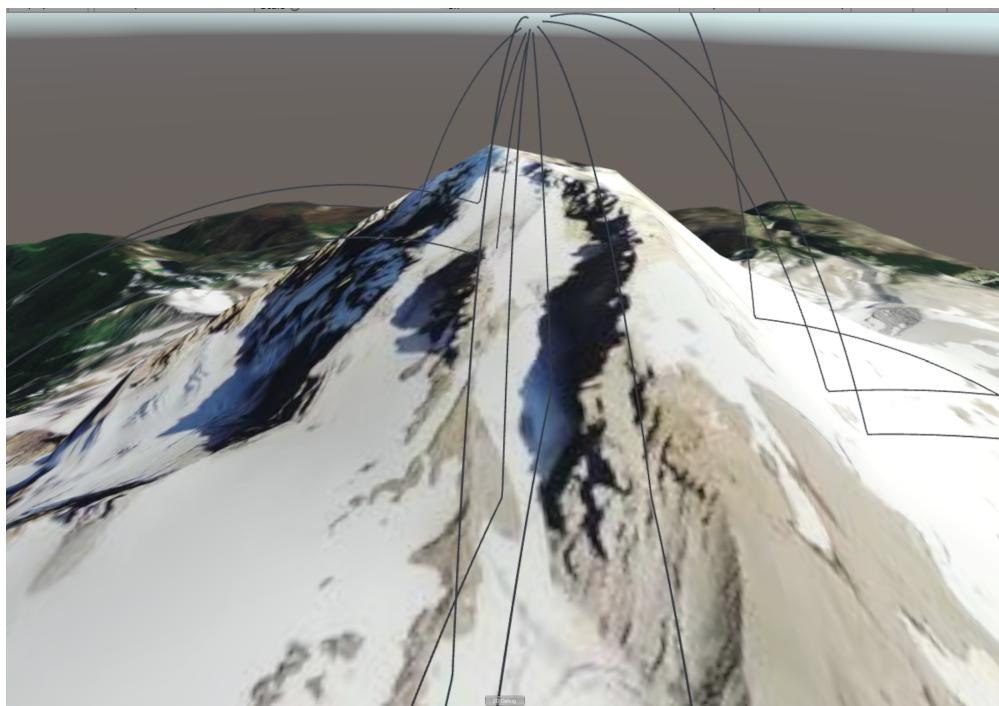


FIGURE 40 – Forme Sphere - Dampen 0 - Bounce 1

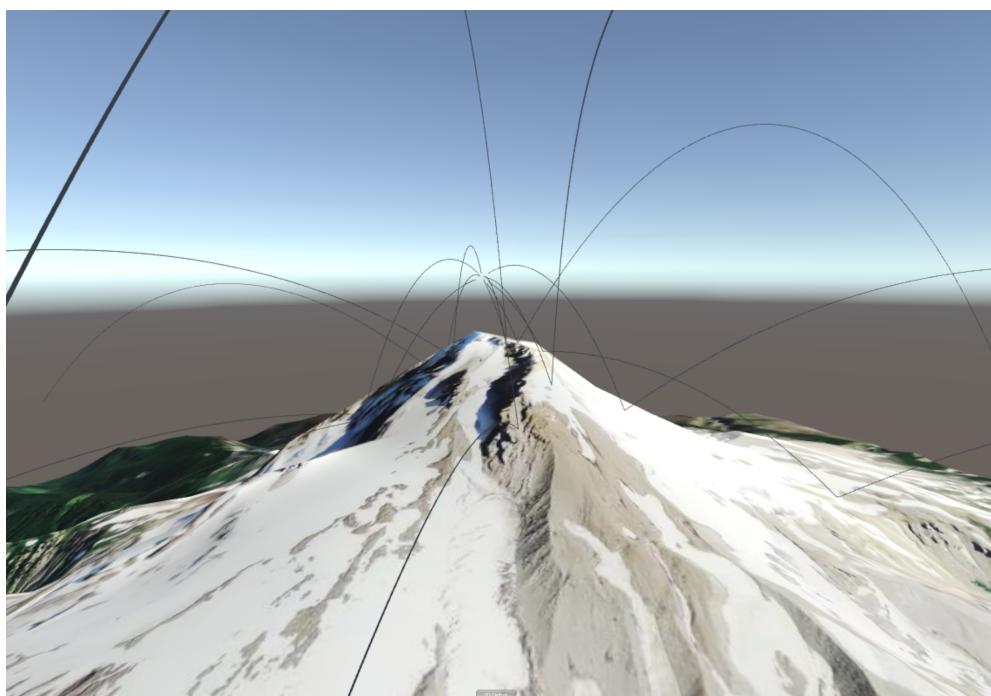


FIGURE 41 – Forme Sphere - Dampen 0 - Bounce 2

Comme on pouvait s'y attendre en augmentant la valeur rebond on constate que les particules rebondissent plus haut.

4.7.2 Résultat en faisant varier Dampen

Le Dampen correspond à l'accrochage entre le terrain et les particules. On s'intéresse ici à son influence sur le flux.

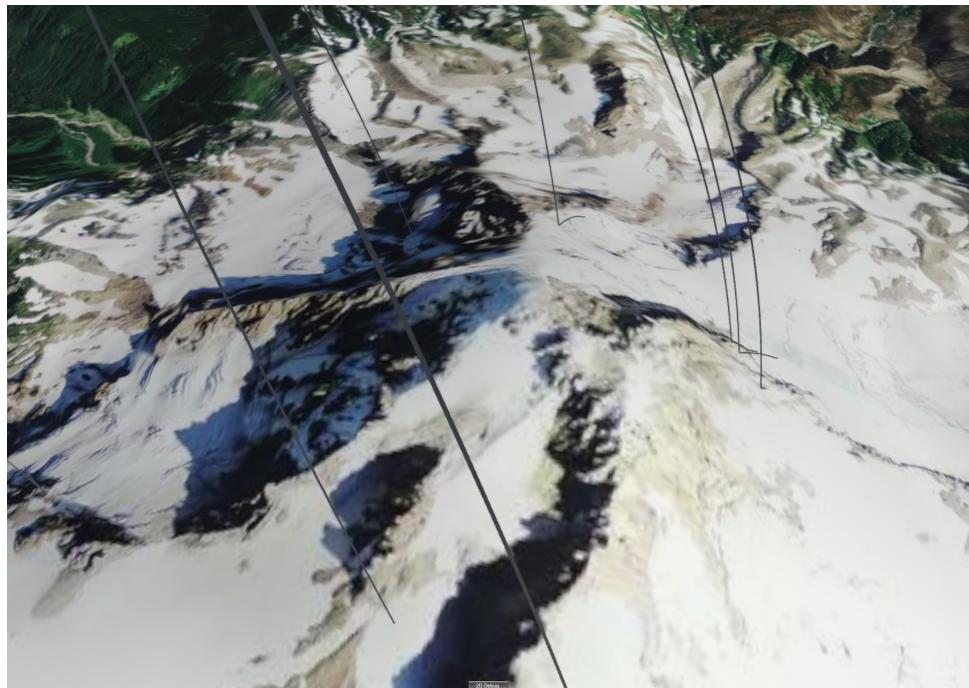


FIGURE 42 – Forme Sphere - Dampen 0.5 - Bounce 0.2

On voit qu'avec un Dampen à 0.5, la particule s'arrête pratiquement immédiatement lorsqu'elle touche le terrain. On voit quand même que selon l'endroit où elle tombe sur le terrain, elle peut se déplacer encore un peu. Si vous regardez bien la particule en haut au centre vous verrez que le trajet continu légèrement une fois au seul. Cette effet est bien visible en réalité virtuelle.

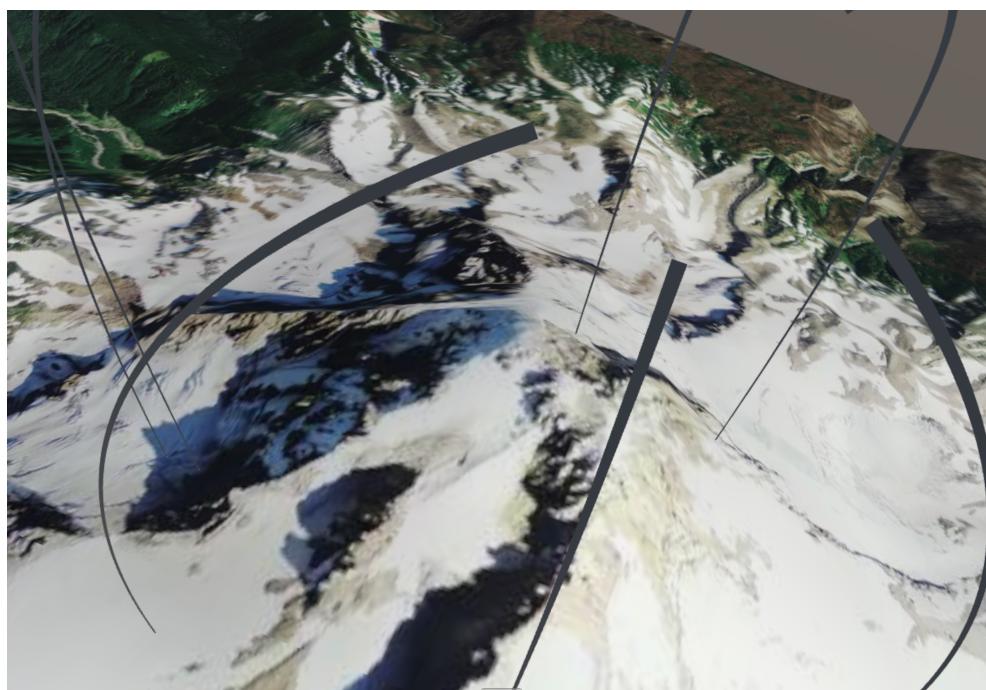


FIGURE 43 – Forme Sphere - Dampen 1 - Bounce 0.2

Comme on peut s'y attendre en augmentant encore la valeur de Dampen toutes les particules ne se déplacent plus une fois en contacte avec le terrain.

4.7.3 Résultat en faisant varier la forme

Ici on s'intéresse à la forme de la source :

Avec un cercle :

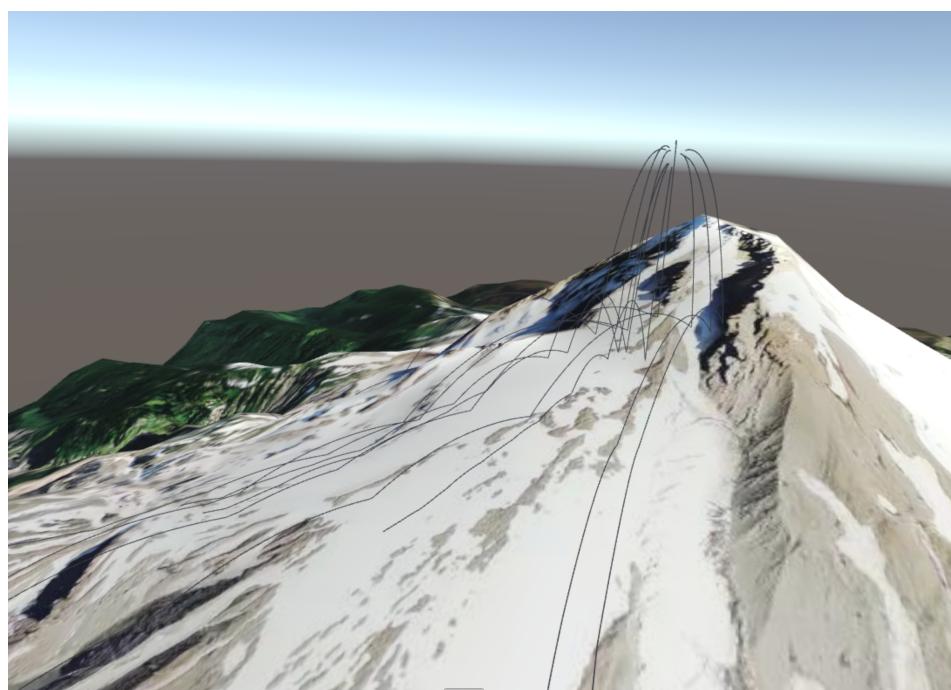


FIGURE 44 – Forme Cercle - Dampen 0 - Bounce 0.5

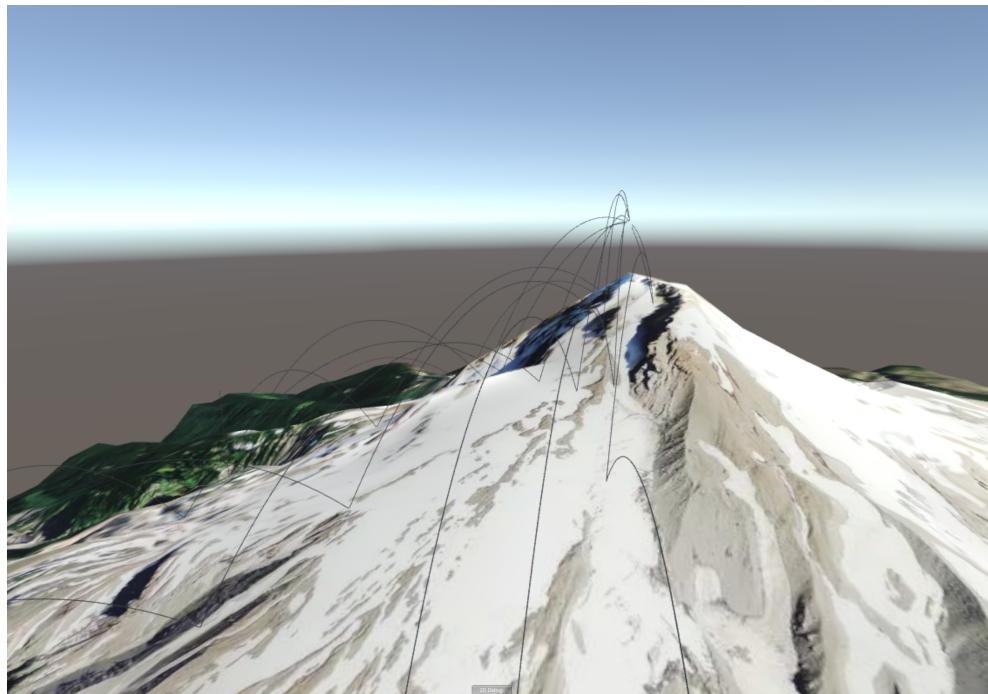


FIGURE 45 – Forme Cercle - Dampen 0 - Bounce 2

Avec une boite (le rendu d'un cône est plutôt similaire) :

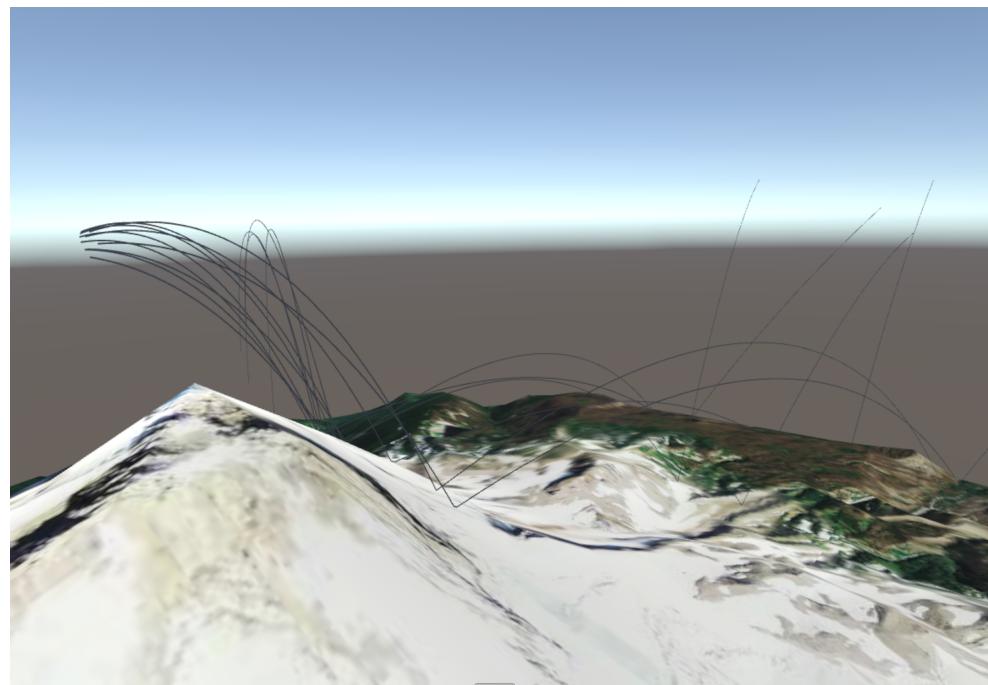


FIGURE 46 – Forme Box/Cone - Dampen 0 - Bounce 2

5 Conclusion

Le projet a permis de mettre au point deux approches différentes avec chacune leurs avantages et leurs inconvénients pour la visualisation de données. Le premier programme permet l'affichage de n'importe quelle image 2d en modèle 3D en réalité virtuelle. Nous avons réalisé une interface pour faciliter son utilisation et adapté la construction du modèle à différentes situations (selon le niveau de gris ou la couleur, nombre de couche, etc...). Cependant le premier projet est améliorable, notamment en rajoutant de la physique au modèle 3D et en optimisant le nombre de billes utilisées pour la construction. Le second projet que nous avons réalisé permet de réaliser des interactions de flux de particules avec un terrain modélisé à partir de google map. Ce second projet doit aussi être d'avantage maîtrisé pour gérer des situations particulières (immeubles, etc...).

Au final nous trouvons que ces réalisations sont pertinentes pour montrer l'intérêt de l'utilisation de la réalité virtuelle pour la visualisation de données de part sa représentation plus proche de la réalité.