INF2705 Infographie

Spécification des requis du système

Travail pratique 4

Les panneaux et les lutins s'envolent!

Département de génie informatique et génie logiciel
Polytechnique Montréal
Benoît Ozell, Automne 2021

Table des matières

1 Introduction

Ce document décrit les exigences du TP4 « *Les panneaux et les lutins s'envolent!* » (Automne 2021) du cours INF2705 Infographie.

1.1 But

Le but des travaux pratiques est de permettre à l'étudiant de directement appliquer les notions vues en classe.

1.2 Portée

Chaque travail pratique permet à l'étudiant d'aborder un sujet spécifique.

1.3 Remise

Faites la commande « `make remise` » ou exécutez/cliquez sur « `remise.bat` » afin de créer l'archive « **INF2705_remise_TPn.zip** » (ou .7z, .rar, .tar) que vous déposerez ensuite dans Moodle. (Moodle ajoute automatiquement vos matricules ou le numéro de votre groupe au nom du fichier remis.)

Ce fichier `zip` contient tout le code source du TP (`makefile`, `*.h`, `*.cpp`, `*.glsl`, `*.txt`).

2 Description globale

2.1 But

Le but de ce TP est de permettre à l'étudiant d'assimiler des notions de mouvements d'objets basés sur des phénomènes physiques tels la gravité, le temps et les collisions en mettant en pratique le mode de rétroaction pour faire des calculs sur GPU. Ce TP permet aussi de se familiariser avec les panneaux et les lutins.

2.2 Travail demandé

Partie 1 : le mode rétroaction pour déplacer des particules sur GPU

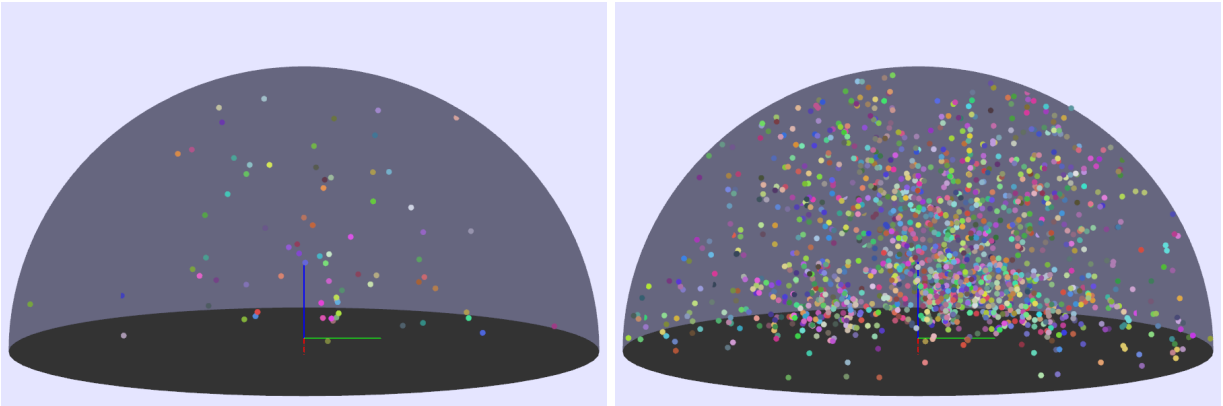

FIGURE 1 – Système de particules avec peu ou énormément de particules avec un nuanceur de géométrie qui affiche des points (sans utiliser de texture)

On demande d'afficher un système de particules évoluant dans le temps (Figure 1). Des particules naissent dans un puits de particules et meurent après une certaine période de temps de vie. Un nombre constant (mais variable) de particules restent actives, ce qui signifie que les particules mortes se « réincarnent ». Les caractéristiques courantes de chaque particule (position, couleur, vitesse, temps de vie) sont conservées dans un VBO afin que tous les calculs soient faits par le GPU.

Les caractéristiques des particules sont conservées dans un premier VBO qui servira d'entrée à un programme GLSL de rétroaction qui produit ses résultats dans un second VBO. Ensuite, le VBO nouvellement rempli sera utilisé pour afficher les particules à leurs nouvelles positions et on recommencera le cycle après avoir inversé (*swap*) la fonction des deux VBO. Autant pour les calculs de la physique que pour l'affichage subséquent, les caractéristiques des particules sont donc toujours conservées, modifiées et utilisées sur le GPU sans jamais les récupérer sur le CPU.
En mode d'affichage, un programme GLSL utilise le VBO rempli avec les caractéristiques courantes. En mode rétroaction, un autre programme GLSL avec un unique nuanceur de sommets effectue tous les calculs : faire (re)naître les particules, les faire avancer et gérer les collisions avec les parois.

Naissance : À la naissance ou renaissance d'une particule, le nuanceur assigne à chacune une direction aléatoire de départ, une durée de vie aléatoire (entre 0 et `tempsDeVieMax` secondes) et une couleur aléatoire (entre COULMIN et COULMAX) en utilisant la fonction de génération de nombres pseudoaléatoires déjà fournie dans le nuanceur de sommets. À chaque fois qu'une particule a épuisé son temps de vie, la particule meure et renaît à la position du puits de particules avec de nouvelles caractéristiques aléatoires. Au démarrage, les particules sont initialisées avec aucun temps de vie restant afin qu'elles renaissent dès le départ.

Déplacement : Une simple méthode d'intégration d'Euler incluant la gravité est suffisante (annexe E). Ce n'est pas la méthode la plus précise, mais elle offre l'avantage d'être très simple.

Collisions : Les collisions avec le sol (en `z = 0`) sont assez simples à gérer : la particule doit rebondir. Les collisions avec les parois de la bulle (un demi-ellipsoïde déformable) peuvent aussi être relativement faciles à gérer : on doit seulement vérifier que la particule demeure toujours à l'intérieur de la bulle. Il est possible de transformer les positions et les normales vers une sphère de rayon unitaire pour y faire les calculs de collision (détails à l'annexe E).

Pour démarrer, on peut s'inspirer des fichiers « `main.cpp` » et « `nuanceurs.glsl` » de l'exemple du mode de rétroaction utilisant un VBO :
`https://gitlab.com/ozell/inf2705-exemples/-/tree/master/11-RetroactionC-vbo/`.

Partie 2 : l'utilisation de lutins

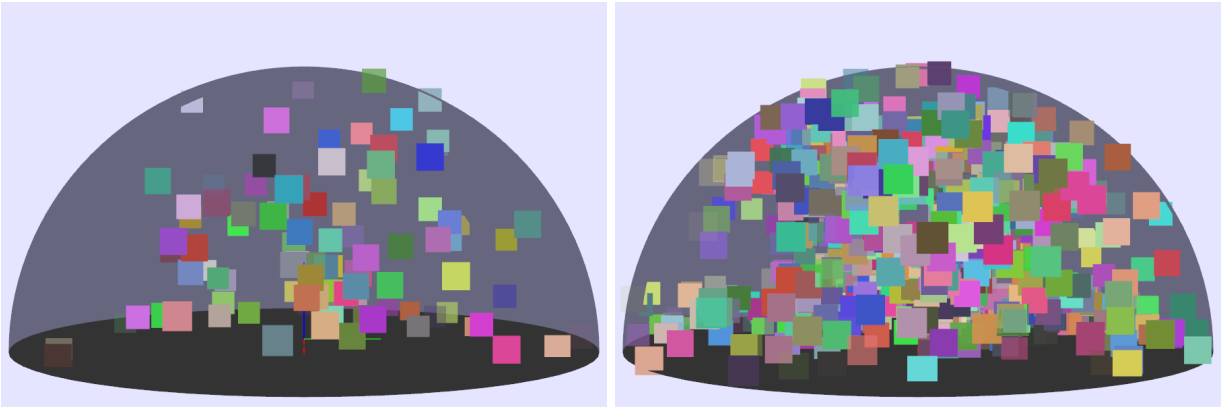

FIGURE 2 – Système de particules avec des panneaux : peu ou beaucoup plus

Les particules devront d'abord être transformées en panneaux carrés (figure 2) ayant des arêtes de longueur « `gl_PointSize` », une valeur qu'on supposera alors directement spécifié en unités du repère de la caméra (plutôt qu'en pixels comme dans la partie 1). On peut donc directement utiliser cette variable de pixels pour dimensionner les panneaux.

Afin d'empêcher les lutins de passer en partie sous le plancher, il faudra modifier un peu les collisions avec le sol et faire rebondir les particules dès « `hauteurPlancher = 0.5 * pointsize` » unités.

Pour démarrer, on s'inspirera des nuanceurs de l'exemple pour l'affichage de panneaux et des lutins :
`https://gitlab.com/ozell/inf2705-exemples/-/tree/master/10-Panneau/`.

Afin de bien comprendre l'utilisation des nuanceurs de géométrie, on y générera des coordonnées de texture qui varieront en fonction du temps de vie restant à chaque « particule » afin d'afficher des lutins représentant un oiseau en vol ou un flocon glacé tournoyant.

Lorsqu'un lutin est utilisé (« `texnumero != 0` »), on mélangera la couleur du panneau et la couleur du texel selon la formule « `mix( couleurPanneau.rgb, texel.rgb, 0.6 )` » pour colorer le rendu final du fragment, tout en prenant soin de conserver la valeur alpha de la couleur du panneau. De plus, la valeur alpha des texels sera utilisée afin de ne pas afficher (`discard`) les fragments transparents des lutins lorsque « `texel.a < 0.1` ».

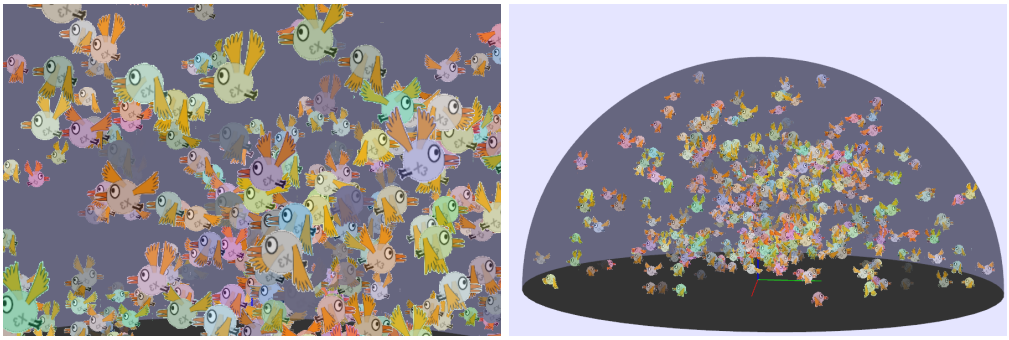

FIGURE 3 – Système de particules avec les oiseaux en lutins

Vol de l'oiseau : On fera voler l'oiseau (figure 3) en faisant varier les coordonnées de texture pour accéder à une des 12 sous-images de la texture des oiseaux. Afin que les oiseaux n'aient pas un vol synchronisé, on utilisera une fréquence de battements d'ailes qui dépend du temps de vie restant de chaque particule : « `18.0 * tempsDeVieRestant` » (18 Hz). La fonction `modulo` permet alors de calculer facilement le numéro de la sous-texture (le décalage à appliquer) dans la texture originale :

```
const float nlutins = 16.0; // 16 positions de vol dans la texture
int num = int( mod( 18.0 * tempsDeVieRestant, nlutins ) ); // 18 Hz
texCoord.s = ( texCoord.s + num ) / nlutins;
```

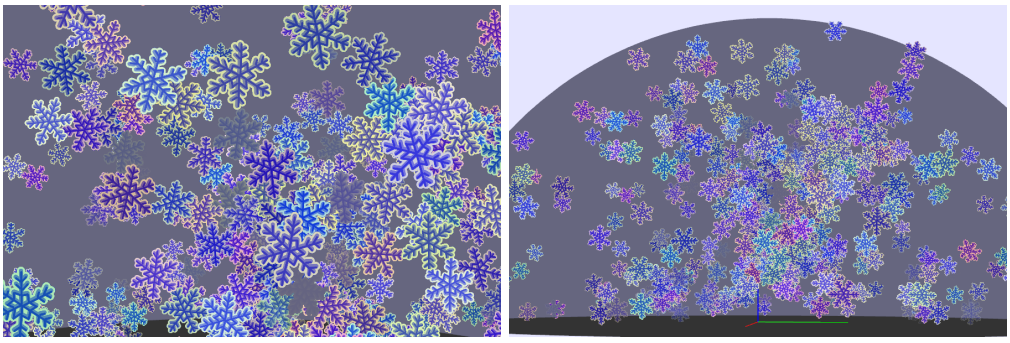

FIGURE 4 – Système de particules avec les flocons en lutins (deux points de vue différents)

Tournoiement du flocon : De même, on fera tourner le flocon autour de son centre (figure 4) en construisant une une matrice de rotation, avec un angle de rotation de « `6.0 * tempsDeVieRestant` » (6 rad/s), qui sera appliquée aux coordonnées des sommets [1]

1. Pourquoi ne pas simplement faire tourner les coordonnées de texture `s` et `t` ? (Remplacer la texture « flocon.bmp » par la texture « echiquier.bmp » peut permettre de mieux voir l'effet.)

Partie 3 : Affiner le rendu

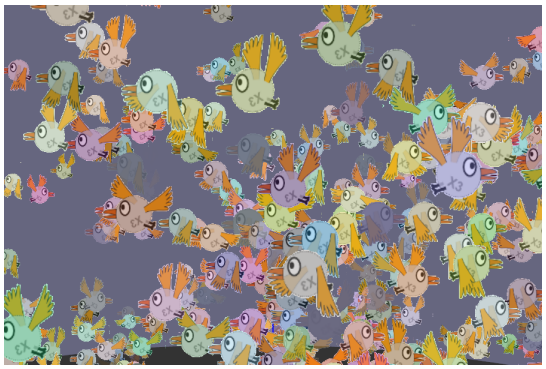

FIGURE 5 – Sens du vol des oiseaux (vers la gauche ou vers la droite)

Pour que le rendu soit plus cohérent, on s'assurera que les oiseaux volent toujours vers l'avant plutôt que de quelquefois voler en « marche arrière ». On corrigera ainsi le vol des oiseaux vers la gauche ou vers la droite, selon l'axe des X. (Indice : utilisez la fonction `sign()`.)
Les flocons peuvent aussi être affectés, mais ce n'est pas nécessaire.

Après chaque collision avec la paroi ou avec le sol, la vitesse des particules sera divisée par 2 afin de les ralentir un peu. Les collisions étant alors inélastiques, les particules iront de plus en plus lentement et s'accumuleront un peu plus au sol.

Lorsque la particule est une hauteur de moins de « `hauteurInerte = 8.0` » unité de hauteur, elle deviendra inerte : le flocon arrêtera de tourner et l'oiseau arrêtera son vol (même si la particule se déplace encore un peu au sol).

Enfin, pour une simulation plus « belle » (en particulier pour les flocons !), on voudra que les lutins disparaissent en « fondant » lorsqu'en fin de vie. Dans le nuanceur de géométrie, on modifiera donc la valeur alpha de la couleur du panneau afin qu'il soit moins opaque lorsque le lutin se rapproche de sa fin de vie (« `AttribsOut.couleur.a = ...` »).

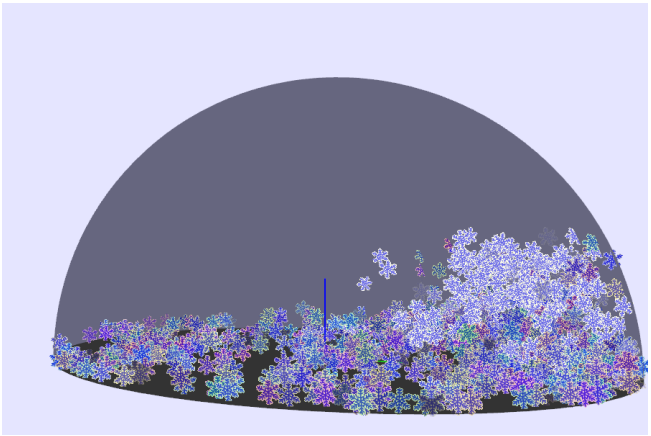

FIGURE 6 – Les lutins s'accumulent au sol et fondent

3 Exigences

3.1 Exigences fonctionnelles

Partie 1 :

E1. Le mode de rétroaction est bien utilisé pour avancer les particules. [3 pts]

E2. Les particules (re)naissent à la position du puits, avec une direction aléatoire de départ, une durée de vie aléatoire entre 0 et `tempsDeVieMax` secondes et avec une couleur aléatoire entre COULMIN et COULMAX. [3 pts]

E3. La gravité est implémentée correctement, donnant une trajectoire de parabole aux particules. [2 pts]

E4. Les particules rebondissent par collision avec les parois et au sol. [2 pts]

Partie 2 :

E5. Les particules sont représentées avec des lutins de taille « `gl_PointSize` ». [2 pts]

E6. La couleur finale du lutin dépend de la couleur du panneau et de celle du texel. [1 pt]

E7. La transparence des texels des lutins est prise en compte lorsque `texel.a < 0.1`. [1 pt]

E8. Les oiseaux volent. (Les lutins varient correctement en fonction du temps.) [2 pts]

E9. Les flocons tournent autour de leur centre. [2 pts]

E10. Le vol des oiseaux et les variations des flocons dépendent du temps de vie restant de chaque particule. [2 pts]

E11. Les particules rebondissent par collision au sol à `hauteurPlancher` du sol. [1 pt]

Partie 3 :

E12. Les oiseaux volent toujours dans le bon sens et non quelquefois vers l'arrière. [2 pts]

E13. Après une collision, la vitesse des particules est divisée par 2. [1 pt]

E14. Les oiseaux ou flocons à moins de `hauteurInerte` ne volent ou ne tournoient plus. [2 pts]

E15. Les lutins « fondent » en fin de temps de vie et deviennent transparents. [2 pts]

3.2 Exigences non fonctionnelles

De façon générale, le code que vous ajouterez sera de bonne qualité. Évitez les énoncés superflus (qui montrent que vous ne comprenez pas bien ce que vous faites !), les commentaires erronés ou simplement absents, les mauvaises indentations, etc. [2 pts]

ANNEXES

A Liste des commandes

Touche	Description
`q`	Quitter l'application
`x`	Activer/désactiver l'affichage des axes
`v`	Recharger les fichiers des nuanceurs et recréer le programme
`j`	Incrémenter le nombre de particules
`u`	Décrémenter le nombre de particules
`i`	Imprimer les caractéristiques des 10 premières particules
`PAGEPREC`	Augmenter la dimension de la boîte en X
`PAGESUIV`	Diminuer la dimension de la boîte en X
`DROITE`	Augmenter la dimension de la boîte en Y
`GAUCHE`	Diminuer la dimension de la boîte en Y
`HAUT`	Augmenter la dimension de la boîte en Z
`BAS`	Diminuer la dimension de la boîte en Z
`0`	Remettre le puits à la position initiale
`PLUS`	Avancer la caméra
`MOINS`	Reculer la caméra
`b`	Incrémenter la gravité
`h`	Décrémenter la gravité
`l`	Incrémenter la durée de vie maximale
`k`	Décrémenter la durée de vie maximale
`f`	Incrémenter la taille des particules
`d`	Décrémenter la taille des particules
`t`	Changer la texture utilisée : 0-aucune, 1-oiseau, 2-flocon
`a`	Boucler sur les textures automatiquement
`g`	Permuter l'affichage en fil de fer ou plein
`ESPACE`	Mettre en pause ou reprendre l'animation
`BOUTON GAUCHE`	Manipuler la caméra
`BOUTON DROIT`	Déplacer le puits
`Molette`	Changer la distance de la caméra

B Textures utilisées

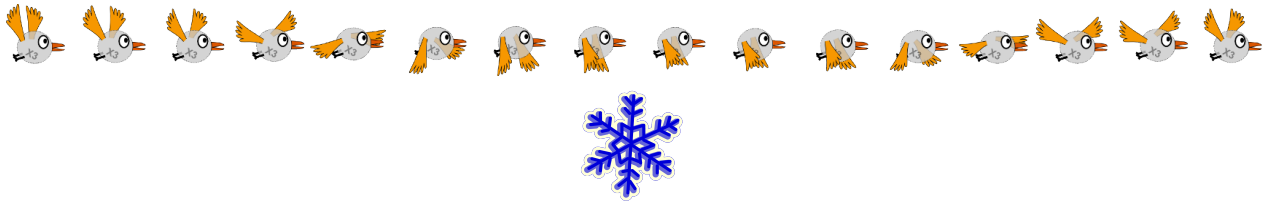

FIGURE 7 – Les textures fournies

C Figures supplémentaires

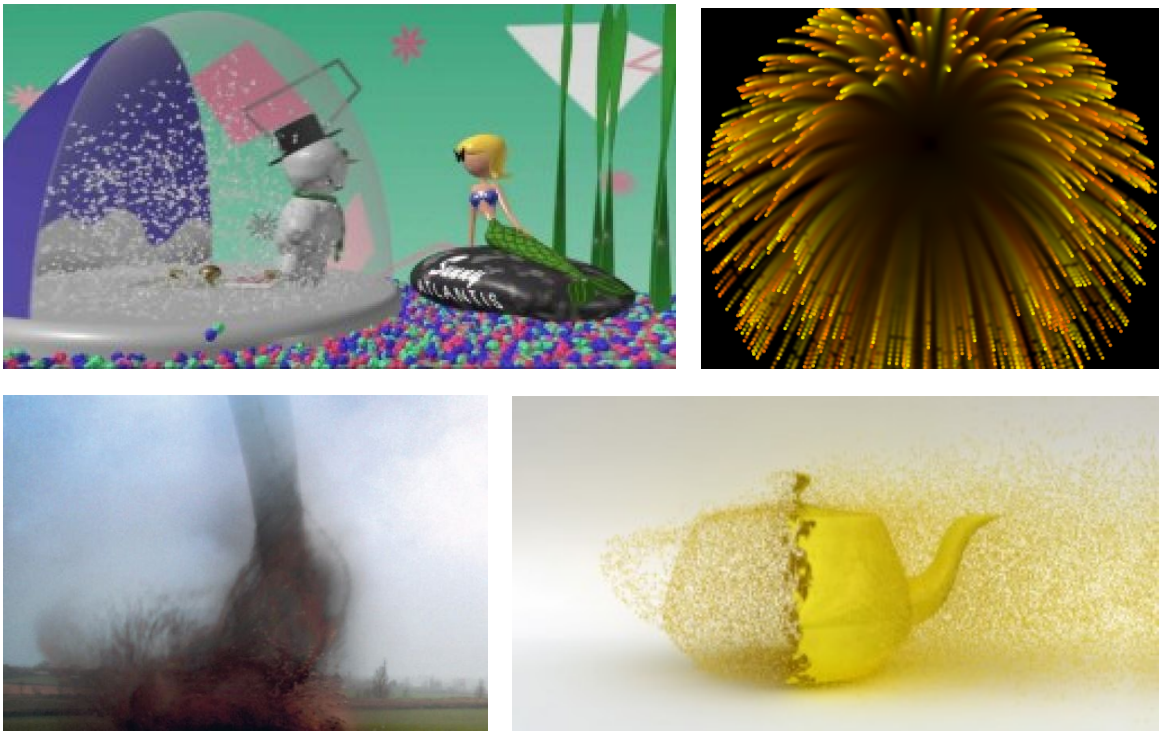

FIGURE 8 – Divers systèmes de particules

D Apprentissage supplémentaire

1. Amortir le mouvement des particules lors de chaque collision.
2. Gérer les collisions sur des parois orientées différemment.
3. Gérer les collisions sur des objets quelconques dans la scène.
4. Lorsque le nombre de collisions est impair, inverser la couleur de la particule dans le nuanceur de sommets.
5. Utiliser une couleur différente selon le nombre de collisions et faire que la particule meure après un certain nombre de collisions.
6. Faire en sorte que la gravité soit appliquée de façon un peu différente selon la couleur de la particule (par exemple : une rouge tombe plus vite et une bleu moins vite).
7. Ajouter des contrôles pour changer la direction de la gravité.
8. Afficher une vue en stéréo anaglyphe ou en stéréo double.
9. Rendre les particules deux fois plus transparentes à chaque collision.
10. Inverser la couleur de la particule, dans le nuanceur de sommets, lorsque le nombre de collisions est impair.
11. (*) Afficher des sphères avec un niveau de détail différent selon la distance à l'observateur.
12. (*) Faire en sorte que chaque particule laisse une trace dans l'espace.
13. (*) Faire que les particules soient des sources lumineuses, chacune avec une intensité de « `1.0/nparticules` », afin d'éclairer les parois intérieures.

* : *Pour ces éléments, il est préférable de n'afficher qu'un petit nombre de particules.*

E Formules utilisées

Intégration d'Euler pour avancer une particule

Pour avancer les particules, on peut utiliser le méthode d'Euler. Ce n'est pas la méthode la plus précise, mais elle est très simple :

```
positionMod = position + dt * vitesse;
vitesseMod = vitesse;
```

On ne tiendra pas compte ici de la masse ou de la friction dans nos calculs et on appliquera la gravité en réduisant simplement un peu la composante z de la vitesse par `gravite * dt` à chaque itération.

Collisions avec les parois

Pour gérer les collisions « rigides » (sans perte de vitesse) avec les parois de la bulle (un demi-ellipsoïde déformé), on doit vérifier que la particule demeure toujours à l'intérieur de la bulle. On peut transformer les positions et les normales vers une sphère de rayon unitaire pour faire ces vérifications de collision.

Pour transformer la position vers la sphère unitaire, il faut appliquer l'inverse de la transformation de modélisation utilisée pour déformer la sphère originale (une `FormeSphere` de rayon 1). Dans le cas présent, on sait que c'est une unique mise à l'échelle par le `vec3 bDim`. On doit ainsi *diviser* chaque composante de la position par la dimension correspondante :

```
vec3 posSphUnitaire = positionMod / bDim;
```

Par contre, pour transformation un vecteur, il faut se rappeler le calcul des normales pour l'illumination. Il appliquer *l'inverse* de la transformation ci-dessus, c'est-à-dire l'inverse de l'inverse de la mise à l'échelle. On doit ainsi *multiplier* chaque composante de la vitesse par la dimension correspondante :

```
vec3 vitSphUnitaire = vitesseMod * bDim;
```

Alors, pour vérifier si la particule est encore à l'intérieur de la bulle (=à l'intérieur de la sphère unitaire), on peut simplement vérifier si la longueur de `posSphUnitaire` est inférieure à `1`.

Si la particule est sortie de la bulle, on peut la ramener à l'intérieur avec l'approximation ci-dessous pour obtenir un effet de collision et, surtout, utiliser la fonction GLSL « `reflect(V,N)` » pour obtenir une nouvelle direction du vecteur vitesse après la collision :

```
float dist = length( posSphUnitaire );
if ( dist >= 1.0 ) // ... la particule est sortie de la bulle
{
   positionMod = ( 2.0 - dist ) * positionMod;
   vec3 N = posSphUnitaire / dist; // normaliser N
   vec3 vitReflechieSphUnitaire = reflect( vitSphUnitaire, N );
   vitesseMod = vitReflechieSphUnitaire / bDim;
}
```

(*Note* : Pour tester vos collisions, vous pouvez temporairement modifier dans votre nuanceur et donner la même direction de départ à toutes les particules. Vous pouvez alors choisir explicitement une direction pour faire quelques tests, par exemple : `vitesseMod = vec3( 0.0, 30.0, 50.0 )`.)