Antoine Pizon

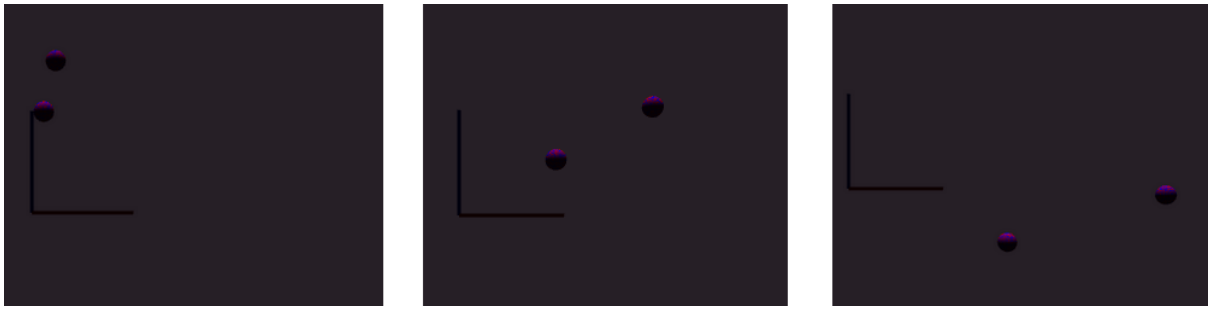
Informatique Graphique : TP6

Partie I : Simple particules

À chaque frame, on doit mettre à jour la position des particules en fonction du temps. Pour ce faire, il faut tout d'abord calculer la vitesse mise à jour, on intègre ensuite cette vitesse pour avoir la position.

La vitesse est tout simplement l'accélération intégrée, on calcule l'accélération d'après la loi de Newton selon laquelle la somme des forces est égale à la masse multipliée par l'accélération. On a donc l'accélération qui est égale à la somme des forces (accessible via `p->getForce()`) divisée par la masse. On ajoute cette accélération à la vitesse (multipliée par le temps écoulé depuis le dernier calcul), puis on ajoute la vitesse à la position (toujours multipliée par l'intervalle de temps).

```
p->setVelocity(p->getVelocity() + (p->getForce() / p->getMass()) * dt);
p->setPosition(p->getPosition() + p->getVelocity() * dt);
```



Résultat obtenu avec deux particules

Partie II : Ressort

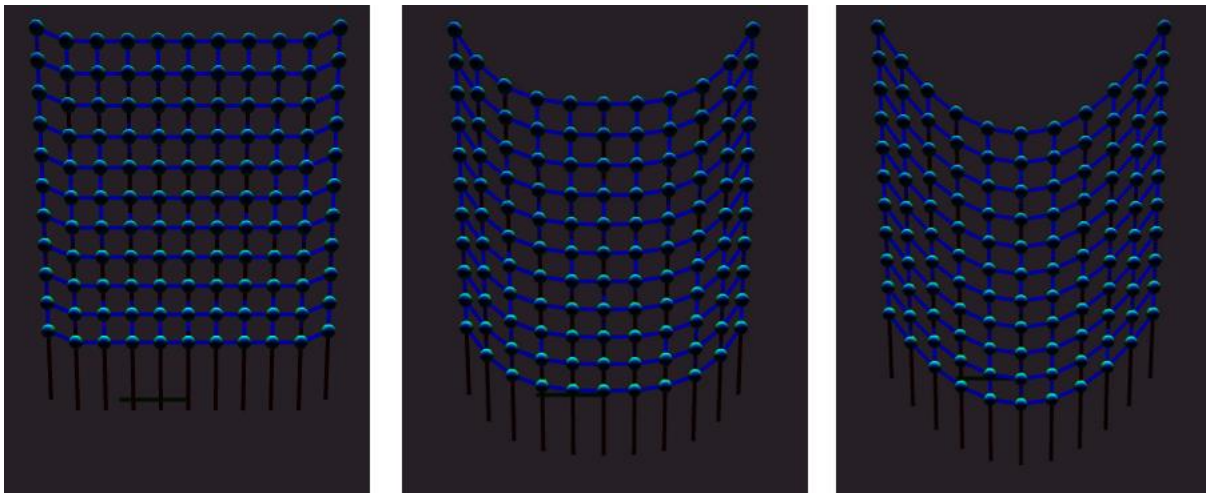
On veut ici simuler un tissu ou une autre surface souple, pour ce faire on suppose que chaque particule composant cette surface est liée aux particules adjacentes par un ressort. Pour chaque duo de particule, on commence par calculer la distance entre ces deux particules (avec `glm ::length`) puis le vecteur normalisé (grâce à `glm ::normalize`).

```
float length = glm::length(m_p1->getPosition() - m_p2->getPosition());
glm::vec3 norm = glm::normalize(m_p1->getPosition() - m_p2->getPosition());
```

On calcule ensuite les deux forces qui s'exerce sur la particule, tout d'abord la force équivalente à un système « idéal » puis celle correspondant à l'amortissement. On ajoute ensuite ces forces à la première particule et leur opposé à la seconde. Toutes ces opérations ne sont réalisées que si la distance entre les deux particules est supérieure à une certaine valeur pour éviter les calculs inutiles.

```
if (length > std::numeric_limits<float>::epsilon()) {
	glm::vec3 idealForce = -m_stiffness * norm
		* (length - m_equilibriumLength);
	glm::vec3 damperForce = -m_damping * norm
		* glm::dot(m_p1->getVelocity() - m_p2->getVelocity(), norm);

	m_p1->incrForce(idealForce + damperForce);
	m_p2->incrForce(-idealForce - damperForce);
}
```



Résultat obtenu avec une grille de particule de taille 10 x 10

Partie III : Collisions

Pour gérer les collisions on va devoir compléter deux méthodes, tout d'abord la méthode `testParticlePlane`, qui test si une particule et un plan sont entrés en collisions, puis la méthode `solveCollision` qui va mettre à jour la vitesse et la position de cette particule suite à cette collision.

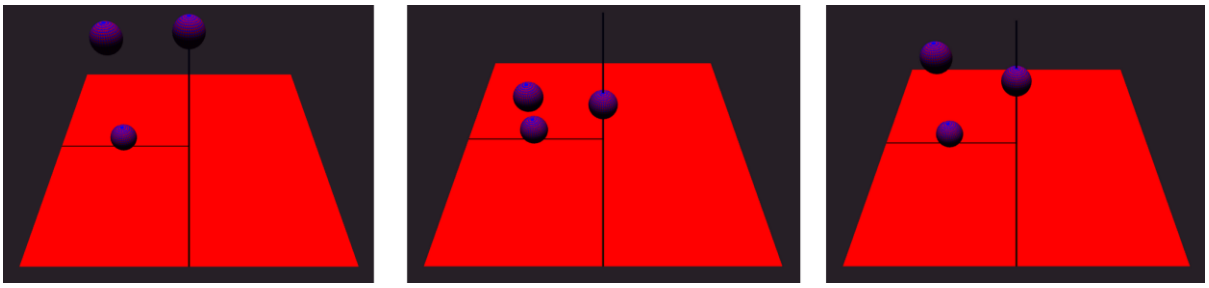
Pour la méthode `testParticlePlane`, on se contente de calculer la distance entre le plan et la particule et de voir si celle-ci est supérieur ou non au rayon de la particule.

```
float dist = glm::dot(particle->getPosition()
	- glm::vec3(0, 0, plane->distanceToOrigin()), plane->normal());

return dist < particle->getRadius();
```

On complète ensuite la partie `do_solveCollision()` qui est chargée de faire réagir la particule en cas de collision. Cette méthode est en trois partie, tout d'abord on calcule la distance entre le plan et la particule, puis on adapte sa position à la nouvelle position et enfin on adapte sa vitesse.

```
float dist = glm::dot(m_particle->getPosition()
	- glm::vec3(0, 0, m_plane->distanceToOrigin()), m_plane->normal());
m_particle->setPosition(m_particle->getPosition()
	- (dist - m_particle->getRadius()) * m_plane->normal());
m_particle->setVelocity(m_particle->getVelocity()- (1 + m_restitution)
	* glm::dot(m_particle->getVelocity(), m_plane->normal()) * m_plane->normal());
```



Résultat obtenu avec deux particules lachées d'une certaine hauteur. Celle de gauche rebondi sur la particule présente sur le plan, celle de droite rebondi directement sur le plan.