INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

Informatique Graphique - TP 5

Animation procédurale

GRAINDORGE Amance

VERNET Hector

Université de Rennes

I. INTRODUCTION

Ce TP a pour objectif d'utiliser un système de simulation simple pour animer une scène. Ce système se base sur l'application de différents types de force à des particules (gravité, ressort, collision). Nous commencerons par mettre en place le calcul du déplacement des particules — qui sera la base de notre simulation — puis nous ajouterons de nouvelles interactions entre particules, les ressorts et les collisions. Enfin, nous observerons l'utilisation de la simulation dans une scène plus complexe avant de l'appliquer à notre projet.

II. EXERCICE 1

Dans ce premier exercice, nous allons écrire la méthode `EulerExplicitSolver::do_solve(…)` qui calcule la vitesse et la position de toutes les particules après un temps `dt` en fonction de son état actuel, en utilisant les équations de mouvement. La voici complétée :

```
1   void EulerExplicitSolver::do_solve(
2     const float& dt,
3     std::vector<ParticlePtr>& particles
4   )
5   {
6     for (ParticlePtr p : particles)
7     {
8       if (!p->isFixed())
9       {
10        // Update particle velocity
11        p->setVelocity(p->getVelocity() + (1 / p->getMass()) * dt * p->getForce());
12        // Update particle position
13        p->setPosition(p->getPosition() + dt * p->getVelocity());
14      }
15    }
16  }
```

Une fois la méthode fonctionnelle, on obtient le résultat suivant :

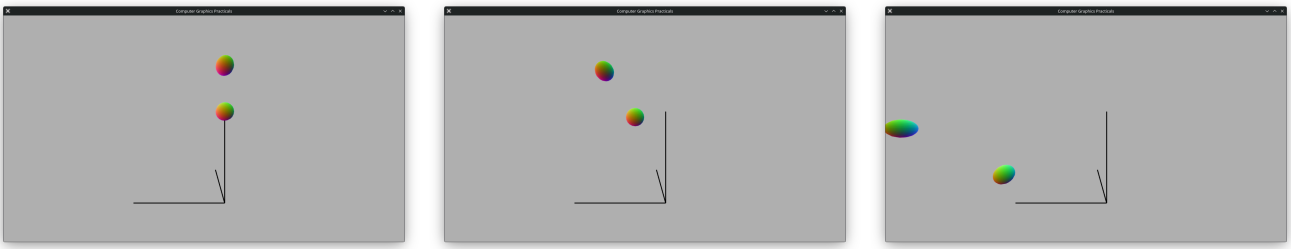

Figure 1 : Implémentation du calcul du mouvement des particules

III. EXERCICE 2

Ce deuxième exercice a pour objectif de simuler un objet composé de plusieurs particules. Pour cela, chaque particule composant l'objet sera consdérée comme liée au autres par un ressort, et subira des forces en conséquence. Nous avons donc écrit la méthode `SpringForceField::do_addForce()`, qui est chargée de calculer les forces à appliquer aux deux extrémités de chaque ressort :

```
1   void SpringForceField::do_addForce()
```

```
2   {
3     // Compute displacement vector
4     glm::vec3 displacement = m_p1->getPosition() - m_p2->getPosition();
5
6     // Compute displacement length
7     float displacementLength = glm::length(displacement);
8
9     // Compute spring force corresponding to the displacement
10    // If the displacement is measurable by the computer (otherwise no force)
11    if (displacementLength > std::numeric_limits<float>::epsilon())
12    {
13      glm::vec3 normalizedDisplacement = glm::normalize(displacement);
14      glm::vec3 idealForce = -m_stiffness * (displacementLength - m_equilibriumLength) *
    normalizedDisplacement;
15      glm::vec3 relativeVelocity = m_p1->getVelocity() - m_p2->getVelocity();
16      float dampingFactor = glm::dot(relativeVelocity, normalizedDisplacement);
17      glm::vec3 dampedForce = -m_damping * dampingFactor * normalizedDisplacement;
18      glm::vec3 totalForce = idealForce + dampedForce;
19
20      m_p1->setForce(m_p1->getForce() + totalForce);
21      m_p2->setForce(m_p2->getForce() - totalForce);
22    }
23  }
```

L'exemple de cet exercice est un filet composé de particules reliées entre elles par des ressorts. Lorsque l'on lance la simulation une première fois sans changer de paramètre, on constate que le filet reste constament en mouvement, ce qui n'est pas réaliste. On change donc les valeurs de l'amortissement (`damping`) global (→10) et celui des ressort (→ 50). L'amortissement simule des pertes d'énergie présentes dans la réalité, comme les frottements de l'air (global) ou l'échauffement (ressort), ce qui va permettre à notre système de se stabiliser après un certain temps :

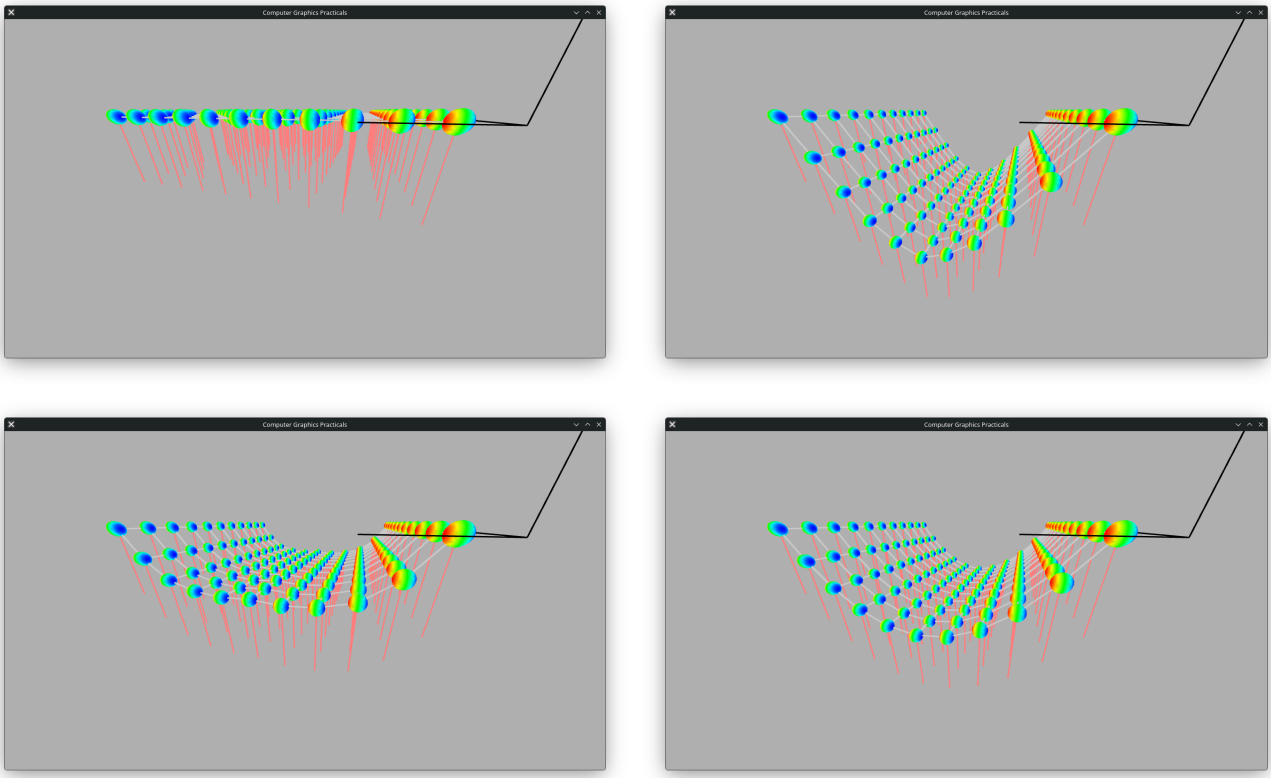

Figure 2 : Simulation de filet avec ressorts

IV. EXERCICE 3

L'objectif de ce troisième exercice est d'implémenter les collisions entre plusieurs particules. Les collisions entre deux particules étant déjà fonctionnelles, nous nous sommes concentrés sur l'interaction entre une particule et un plan. Nous avons donc complété la méthode `ParticlePlaneCollision::do_solveCollision()` qui calcule la nouvelle position et vitesse d'une particule après une collision avec un plan infini. Voici le code complété :

```
1   void ParticlePlaneCollision::do_solveCollision()
2   {
3     // Don't process fixed particles (Let's assume that the ground plane is fixed)
4     if (m_particle->isFixed())
5       return;
6
7     // Compute interpenetration distance
8     float planeParticleDist = glm::dot(
9           m_particle->getPosition(),
10          m_plane->normal()
11    ) - m_plane->distanceToOrigin();
12    float interpenetrationDist = m_particle->getRadius() - planeParticleDist;
13
14    // No collision
15    if (interpenetrationDist <= 0)
16      return;
17
18    // Project the particle on the plane
19    glm::vec3 proj = m_plane->projectOnPlane(m_particle->getPosition());
20    m_particle->setPosition(proj + m_particle->getRadius() * m_plane->normal());
```

```
21
22      // Compute post-collision velocity
23      m_particle->setVelocity(
24        m_particle->getVelocity()
25        - (1.0f + m_restitution) * glm::dot(m_particle->getVelocity(),
26        m_plane->normal()) * (m_plane->normal())
27      );
28    }
```

Nous avons également complété la fonction de test de collision `testParticlePlane()` :

```
1   bool testParticlePlane(const ParticlePtr& particle, const PlanePtr& plane)
2   {
3     float particlePlaneDist = glm::dot(
4       particle->getPosition(), plane->normal()
5     ) - plane->distanceToOrigin();
6     return std::abs(particlePlaneDist) <= particle->getRadius();
7   }
```

On teste ensuite notre implémentation avec une scène simple où une particule rebondit sur une particule fixe, et une autre particule rebondit sur un plan fixe. On ajoute également une composante horizontale à la vitesse initiale de l'une des particules pour vérifier le réalisme des interactions. Voici le résultat obtenu :

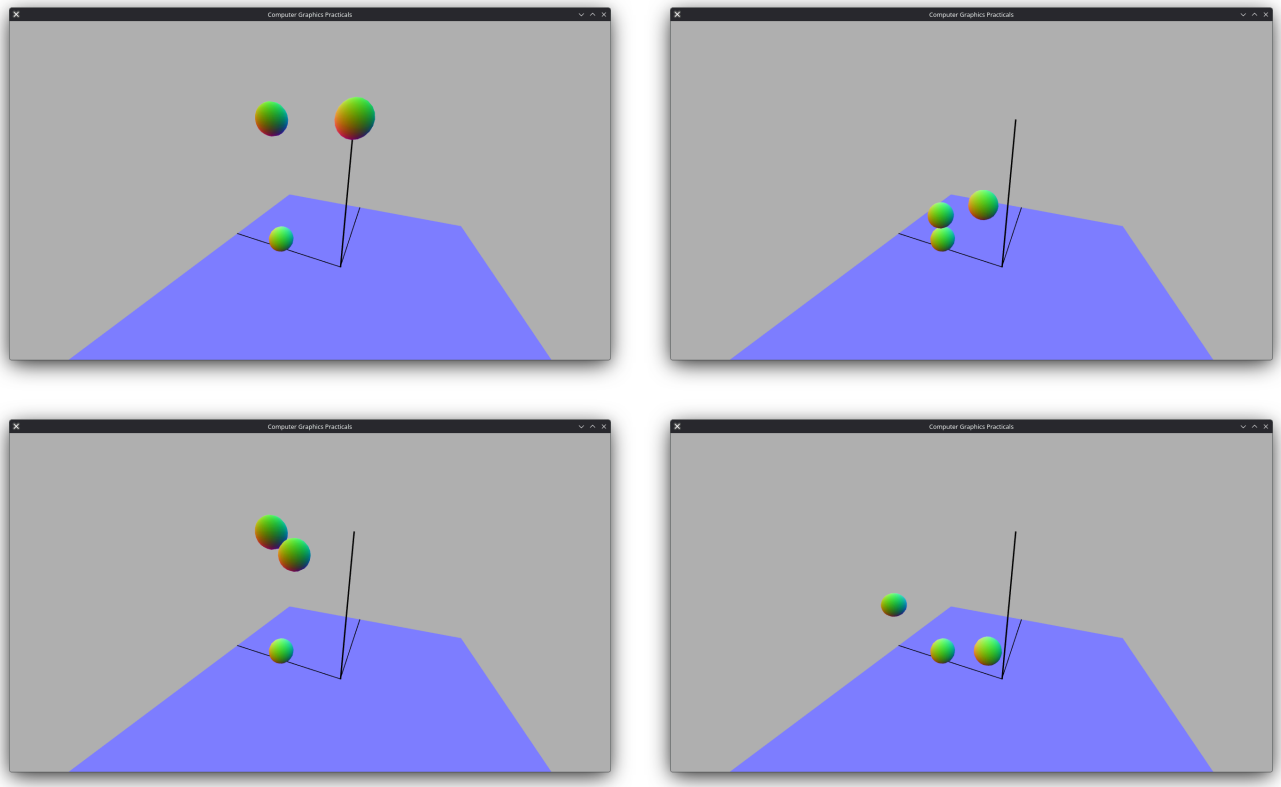

Figure 3 : Simulation de collisions entre particules et plan

On voit que les collisions sont bien gérées, avec un rebond réaliste des particules. Cependant, dans le cas d'un grand nombre d'interactions, des problèmes d'interpenetration peuvent subvenir, car la position des particules n'est corrigée qu'une seule fois par itération. Des particules peuvent donc se retrouver à l'intérieur d'autres particules *après* leur calcul de collision.

V. EXERCICE 4

Ce dernier exercice nous propose de tester notre système de simulations dans une scène plus complexe, composée d'une table de billard et de deux particules. L'une des particules peut être controlée au clavier par l'utilisateur, en utilisant la classe `ControlledForceFieldRenderable`, tandis que l'autre réagit seulement aux collisions. Voici le résultat obtenu :

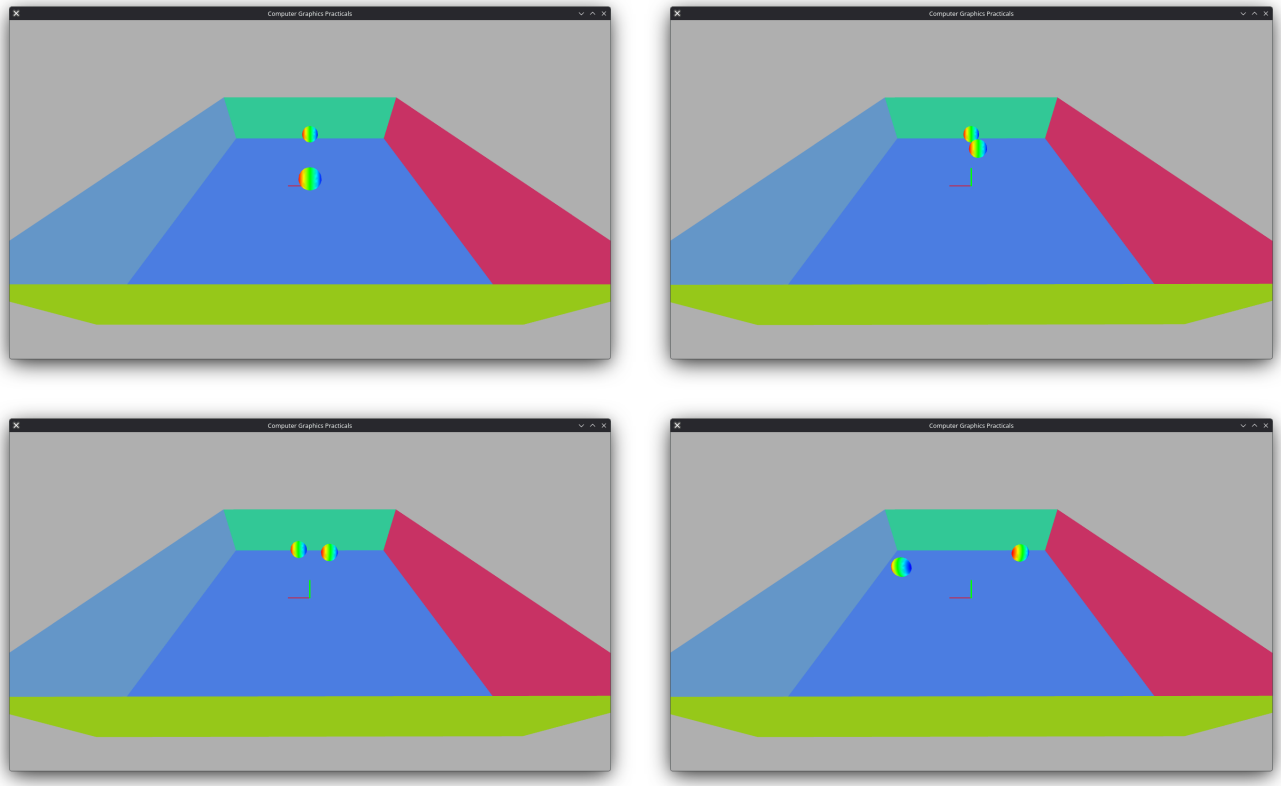

Figure 4 : Simulation d'une table de billard avec particules contrôlables

VI. CONCLUSION

Ce TP nous a permis de mettre en place un système de simulation physique basé sur l'application de forces à des particules. Nous avons implémenté le calcul du mouvement des particules, la gestion des ressorts entre particules, ainsi que les collisions entre particules et plans. Ces éléments constituent une base solide pour des simulations plus complexes, et pourront être utilisés dans notre projet final.