Nicolas Luciani
7/03/2023
Master 1 IMAGINE Université de Montpellier

HAI819I : Moteurs de jeux
TP 5
Compte-rendu

---

1 Introduction

L'objectif de ces TPs est d'implémenter des premieres intégrations numériques pour implanter de la physique dans notre moteur.

2 Corps rigide

Pour pouvoir calculer les effets de la physique sur nos objets, nous avons décidé de créer une classe RigidBody qui contiendra tous les calculs nécessaires ainsi qu'une fonction de mise à jour de la position du Transform de l'entité à laquelle il est associé. Cette fonction est dépendante du temps.

```
#define EARTH_G 9.81
class RigidBody
{
public:
        RigidBody(float _mass=1.0):mass(_mass){};
        ~RigidBody(){};
        void computeForces(float deltaTime)
        {
                glm::vec3 F = glm::vec3(0.0, 0.0, 0.0);
                for (int i = 0; i < others.size(); ++i)
                {
                        F += others[i];
                }
                glm::vec3 sumForces = glm::vec3(0.0, -EARTH_G, 0.0) + F;
                float invMass = mass > 0.0f ? (1.0/mass) : 0.0f;
                glm::vec3 a = invMass * sumForces;
                setAcceleration(a);
                glm::vec3 a_t = deltaTime * a;
                glm::vec3 v_t = speed + a_t;
                setSpeed(v_t);
                others = std::vector<glm::vec3>();
        }
        void applyForce(glm::vec3 force){others.push_back(force);}
        glm::vec3 getForcesDirection()
        {
                glm::vec3 F = glm::vec3(0.0, 0.0, 0.0);
                for (int i = 0; i < others.size(); ++i)
```

```
                {
                        F += others[i];
                }
                F += glm::vec3(0.0, -0.1, 0.0);
                return glm::normalize(F);
        }

    /* Getters & Setters */
        glm::vec3 computeRebound(glm::vec3 normal) { return glm::reflect(speed, normal);}
private:
        glm::vec3 speed;
        glm::vec3 accel;
        std::vector<glm::vec3> others;

        float mass;
};
```

3 Collision

On calcule alors les collisions entre notre objet et le terrain de façon très simple pour le moment, en ne vérifiant uniquement si la position de l'objet en y est supérieure à 0. Malheureusement, cela pose quelques problèmes vis à vis de la fréquence à laquelle sont calculées ces collisions. Il arrive parfois que le cube s'enfonce dnas le terrain avant que les collisions soient résolues.

```
bool computeCollisions(Entity &e)
{
    return (e.transform.getLocalPosition() - glm::vec3(0.1, 0.1, 0.1))[1] <= 0;
}
```

4 Contrôles au clavier

Nous avons ajouté la commande de saut au clavier.

```
if(key == GLFW_KEY_SPACE && action == GLFW_PRESS)
{
    isJumping = true;
}

/* in main loop */
if(isJumping)
{
    cube.getRigidBody()->applyForce(glm::vec3(0.0, 7.0, -3.0));
    isJumping = false;
}
```

5 Résultats

On peut alors implémenter le rebond dans notre boucle principale

```
glm::vec3 noSpeed = glm::vec3(0.0, 0.0, 0.0);
if(computeCollisions(cube))
{
    glm::vec3 reboundVec = cube.getRigidBody()->computeRebound(glm::vec3(0.0, 1.0, 0.0));
    reboundVec = rebound * reboundVec;
    cube.getRigidBody()->setSpeed(reboundVec);
    if(glm::length(cube.getRigidBody()->getSpeed()) <= 0.01f)
    {
        cube.getRigidBody()->applyForce(glm::vec3(0.0, 0.5, 0.0));
        cube.getRigidBody()->setSpeed(noSpeed);
    }
}
```

On obtient alors les résultats en vidéo, en suivant le lien suivant. https://youtu.be/h8axK43Xa94