Projet de Modélisation et Ingénierie du vivant

Quentin Augrain, Florent Mallard

INSA de Rennes
5INFO

22 janvier 2016

1 Tutoriel

Question 1 : Comme indiqué dans les commentaires de la fonction, il y a deux fonctions dans la boucle :

— Bouger le monde ;

— Afficher le monde.

Cela signifie que la fonction va tout d'abord calculer le prochain mouvement, pour ensuite afficher la nouvelle position. Pour le moment, la phase de calcul n'est pas implémentée et rien ne bouge dans la simulation.

Question 2 : Nous ajoutons la gravité au « force_accumulator », et obtenons le résultat montré à la Figure 1, c'est-à-dire que rien ne se passe. En effet même si nous calculons maintenant les forces, nous ne les appliquons pas au mesh, ce qui résulte en un mesh toujours fixe. Il nous faut donc appliquer ces forces à chaque particule du mesh afin qu'un effet se fasse ressentir.

Question 3 : Maintenant que nous appliquons les forces, le mesh tombe et finit par disparaître de l'écran. Il semble qu'il n'y ait pas de limite physique à sa chute. On peut donc envisager un moyen afin que même si aucune autre interaction que la gravité ne s'applique au mesh, il reste dans le champ de l'écran.

Question 4 : Une autre solution a été proposée, il s'agit de fixer le premier rang des particules. Ainsi, le mesh reste visible et a un effet dit « de rideau ». Pour ce faire, il nous faut nullifier les forces de ces particules. Nous avons compté dix particules sur ce rang, nous avons donc ajouté une boucle comme montré sur la Figure 2. Cette boucle met simplement les forces appliquées à ces dix particules à zéro, empêchant ainsi tout mouvement.

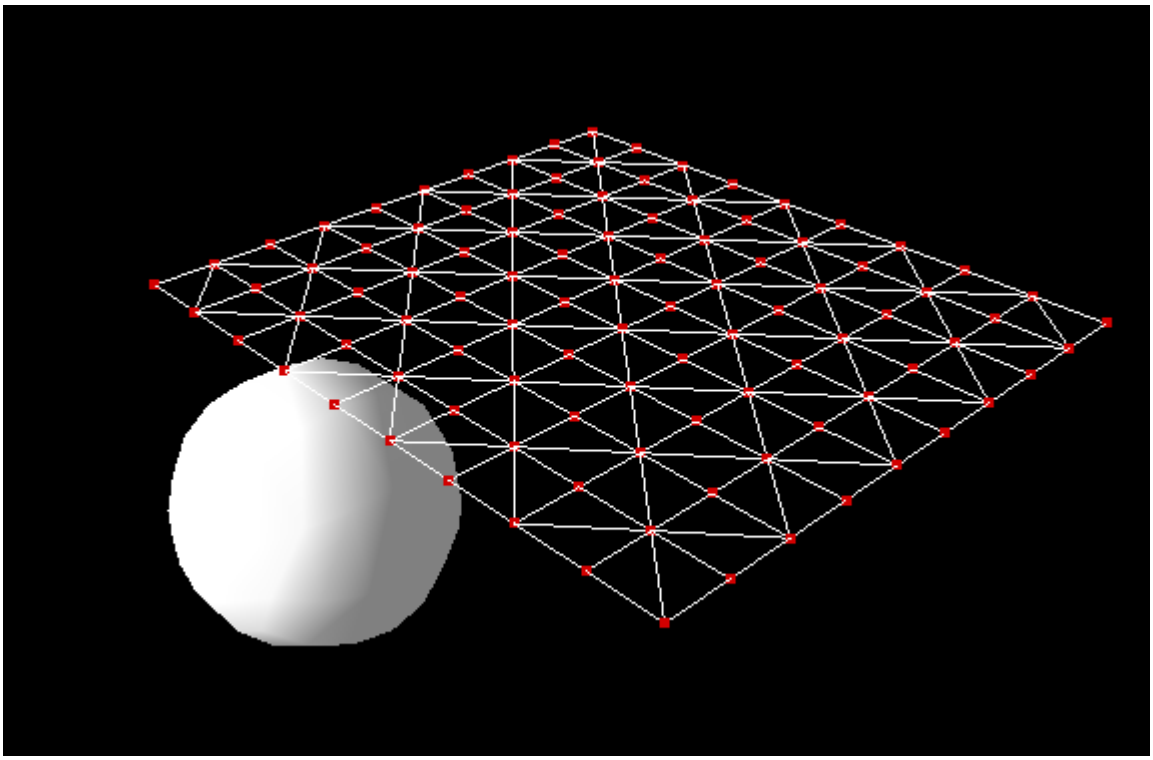

FIGURE 1 – Aucun changement n'est visible même si les forces changent.

```
void Simulator::Update()
{
    // Define the simulation loop (the methods are not in order)
    //ApplyVelocityDamping( ... )
    ComputeForces();

    for (int i = 0; i < 10; i++)
    {
        m_Mesh->particles[i].force_accumulator = Maths::Vector3(0, 0, 0);
    }

    Integrate();
    //UpdateManipulator ( ... )
}
```

FIGURE 2 – Le premier rang reste fixe.

```
for (unsigned int p = 0; p < m_Mesh->particles.size(); p++)
{
    // gravity
    m_Mesh->particles[p].force_accumulator = Maths::Vector3(0, -GRAVITY_CONSTANT, 0);

    // neighbors forces
    Particle *particle = &(m_Mesh->particles[p]);
    for (unsigned int n = 0; n < particle->neighbors.size(); n++)
    {
        Particle *neighbor = particle->neighbors[n];

        float dij = particle->pos.distance(neighbor->pos);
        float dij_init = particle->init_pos.distance(neighbor->init_pos);

        Maths::Vector3 diff_pos = neighbor->pos - particle->pos;
        diff_pos.normalise();

        particle->force_accumulator += diff_pos * K *  (dij - dij_init);
    }
}
```

FIGURE 3 – Mise à jour des voisins pour chaque particule.

Question 5 : Le comportement est pour le moins étrange, car les particules oscillent de plus en plus jusqu'à disparaître. Pour le moment la simulation n'est pas stable, car les forces appliquées aux particules ne se compensent pas. Ceci aboutit, plus ou moins rapidement en fonction des paramètres de simulation choisis, à un écran contenant uniquement le premier rang de particules, demeuré fixe comme précisé à la question précédente. La FIGURE 3 montre notre implémentation.

Question 6 : Le paramètre « dt » définit la vitesse de la simulation à travers le nombre d'itérations à chaque fois. Une valeur plus petite correspond donc à une simulation plus rapide, car on calcule la position et les forces après un instant plus grand, le mouvement a donc été plus important. K définit la rigidité. Celle-ci va aider à obtenir l'effet de rideau désiré. Sachant cela, nous n'avons pas réussi à stabiliser la simulation. En effet même si le rideau est considéré comme plus rigide, les forces des particules ne se compensent pas et donnent le même résultat que précédemment.

Question 7 : Nous ajoutons la boucle demandée qui effectue n appels à « Update » avec un timestep désormais divisé par n. Le but de cette manipulation est d'avoir un mouvement plus petit, car calculé sur un plus petit pas de temps. Grâce à cela les interactions calculées sont plus fines et donc plus réalistes. Bien entendu, cela implique aussi plus de calculs, et donc un coût plus important pour le processeur. Le CPU est bien plus actif après que nous ayons implémenté cette modification.

```
//fix first row of particules to simulate hanging
for (int i = 0; i < 10; i++)
{
    m_Mesh->particles[i].force_accumulator = Maths::Vector3(0, 0, 0);
}
```

FIGURE 4 – La boucle fixant la première rangée.

Question 8 : En amortissant la chute des particules, nous arrivons à stabiliser la simulation. L'amortissement simule l'interaction des particules avec l'air ambiant. C'est lui qui va réussir à compenser les interactions des particules entre elles et empêcher une oscillation de plus en plus importante comme observée précédemment. TODO Image(s) de la chute et/ou du résultat

Question 9 : Avec dt = 1/200, K = 300, et n = 20, nous arrivons à une simulation stable. Mais ces paramètres sont très coûteux en ressources car avec dt = 1 / 200 et n = 20, cela signifie que nous effectuons quatre mille fois la boucle « Update » par seconde. Plus tard, et sur des ordinateurs moins puissants, nous préférerons utiliser un dt de 1/50. Cela n'affecte pas trop la simulation mais permet de diminuer significativement le coût en ressources.

Question 10 : Le modèle de Mass-Spring-Damper est assez simple à utiliser dès lors que les formules à appliquer sont à disposition. En revanche, pour obtenir une simulation stable, le moindre écart dans les paramètres peut créer une simulation tout à fait exotique. En effet nous avons vu que diminuer la rigidité du mesh pouvait conduire à une simulation qui n'était pas réaliste, car le mesh se comportait plus comme une plaque pivotante que comme un rideau. Appliquer un amortissement trop fort rend la simulation trop longue pour être réaliste.

2 TP2

Question 1 : Baisser la valeur d'amortissement résulte en une accélération de la chute et en une baisse de la stabilité. En effet, l'amortissement sert à compenser les interactions entre les particules, qui font osciller le mesh, et à simuler une interaction avec l'air. Diminuer ces « frottements » accélère donc la chute du rideau, et les particules ont des interactions plus fortes qui déstabilisent la simulation. Après quelques recherches, la vitesse relative est simplement une soustraction des vecteurs vitesses du voisin par rapport à la particule.

Question 2 : Afin de simuler le sol, nous implémentons une limite de hauteur en dessous de laquelle les particules ne peuvent descendre, ayant

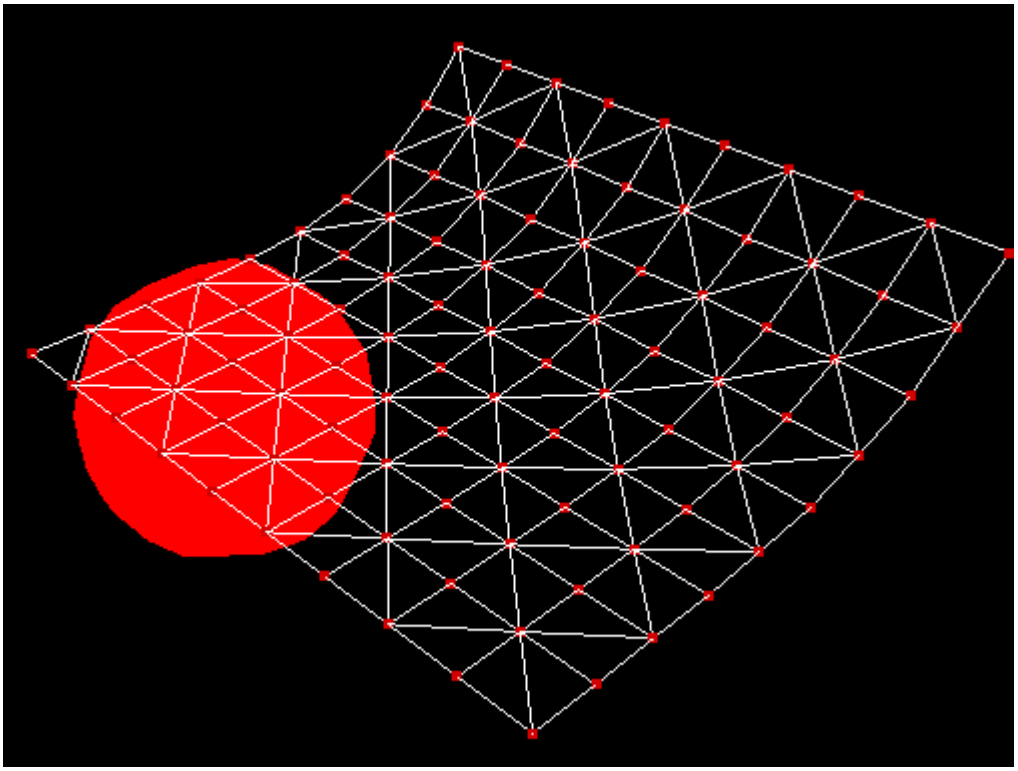

FIGURE 5 – Le mesh pendant la chute.

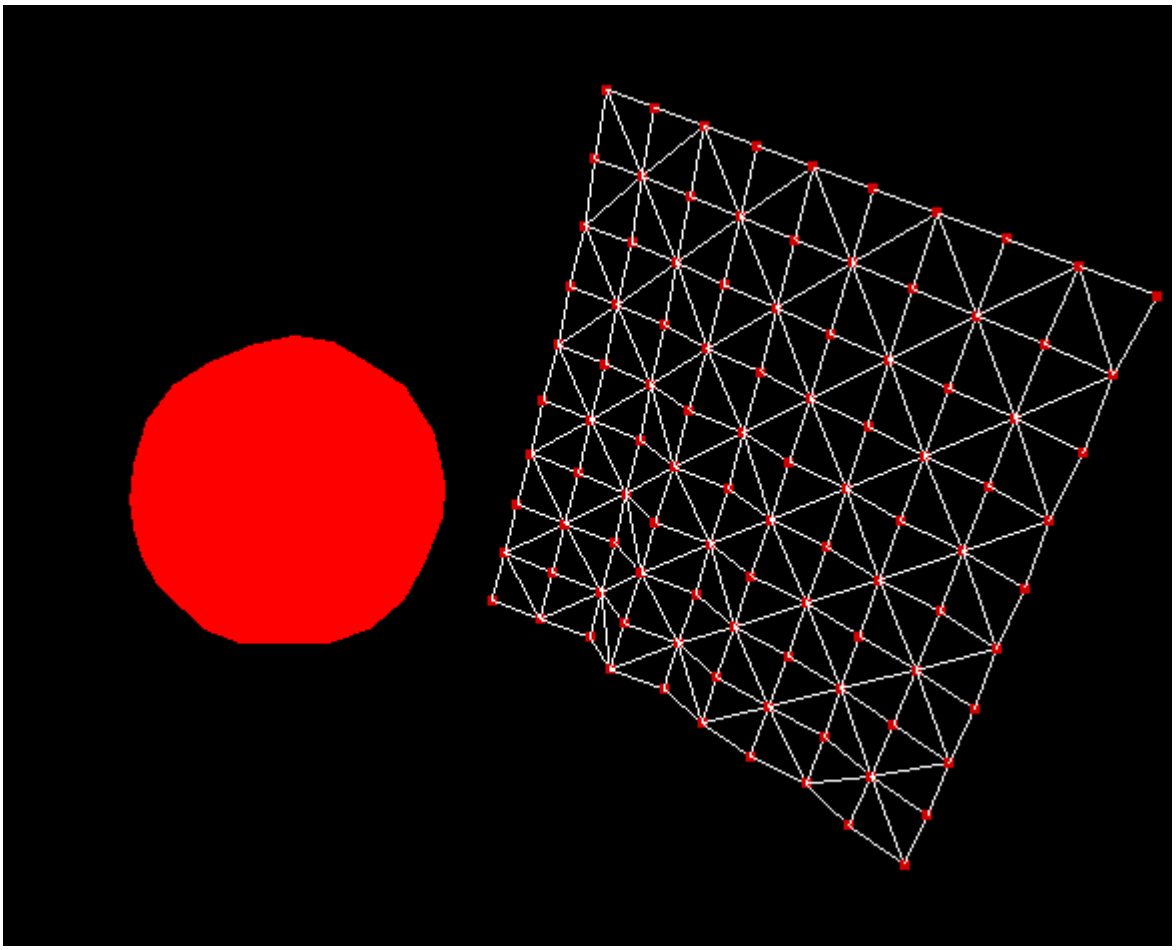

FIGURE 6 – Le mesh stabilisé.

```
// remove all forces on y axis starting y = 0 and below to simulate floor
for (unsigned int p = 0; p < m_Mesh->particles.size(); p++)
{
    if (m_Mesh->particles[p].pos.y <= 0)
    {
        m_Mesh->particles[p].force_accumulator.y = 0;
        m_Mesh->particles[p].vel.y = 0;
    }
}
```

FIGURE 7 – Implémentation du sol.

pour effet de simuler un plan. Cette méthode nous a posé un petit problème lorsque les particules descendaient un peu plus bas que le seuil fixé, car nous supprimions alors toutes les forces sur les particules, et le manipulateur n'interagissait plus avec. TODO image sol + code

Question 3 : Afin de ne pas passer trop de temps sur une question qui ne nous semblait pas essentielle, nous avons simplement changé la couleur de notre manipulator, que nous montrons ci-dessous : TODO

Question 4 : Le coefficient C, s'il est négatif, créé une attraction entre le manipulator et le mesh. En revanche, s'il est positif, les deux objets se repoussent. Celui-ci peut permettre de simuler les propriétés adhérentes ou répulsives (à cause du contact) de différents outils qui pourraient être utilisés en chirurgie.

Question 5 : Les modifications demandées afin de rendre possibles les manipulations avec le client haptique. Ainsi, nous avons choisi d'associer la position du manipulateur à celle du client haptique. De cette manière, les mouvements du manipulateur sont limités, car le bras articulé l'est lui-même, et on a un contrôle très instinctif. TODO code

Question 6 : Pour améliorer le réalisme, un retour de force est rendu possible par le client haptique. Cette fonction se révèle très utile afin de simuler la résistance que l'on peut rencontrer lors d'une manipulation, surtout dans un domaine comme la chirurgie. TODO code

Question 7 : Le client haptique dispose de quatre boutons, pouvant être associés aux fonctions de notre choix. Pour pouvoir utiliser ces boutons, une fonction « HapticButtonClicked » a été mise à notre disposition afin de la compléter. Son rôle se résume pour le moment à détecter lorsqu'on appuie sur un des boutons du client haptique. Cette détection se fait grâce au code de la FIGURE **??**.

Question 8 :TODO code

Question 9 : Après changement du mesh, l'aspect de la simulation est

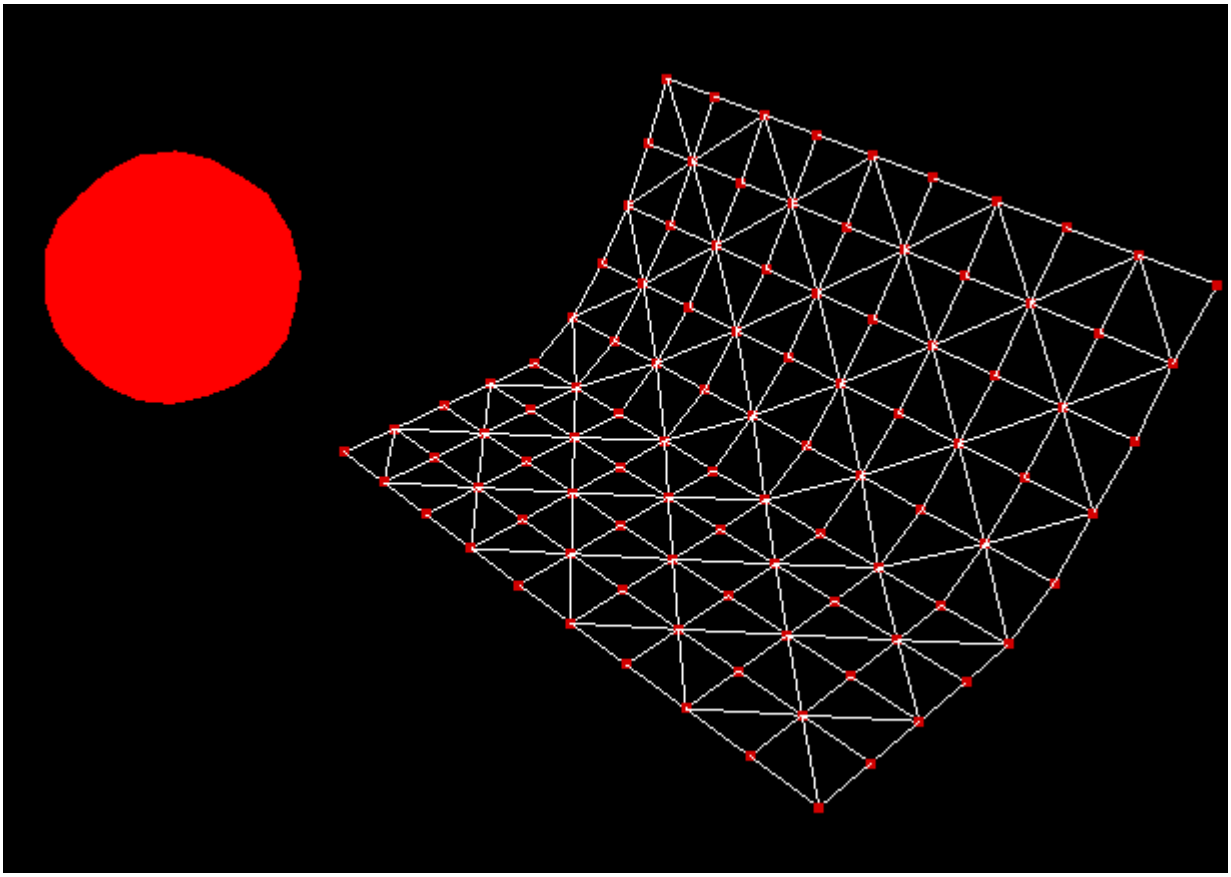

FIGURE 8 – Rideau tombé sur le sol.

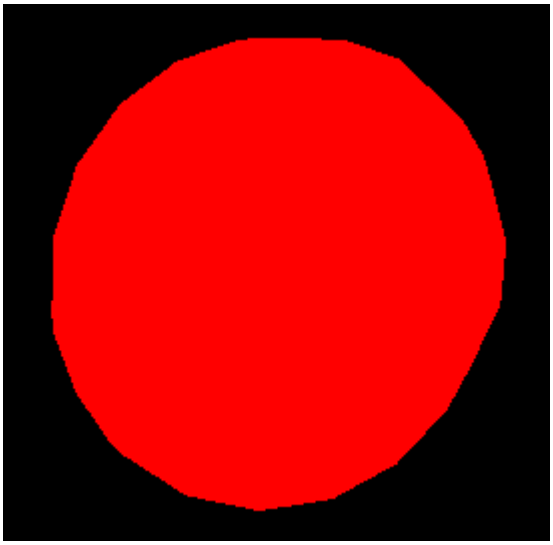

FIGURE 9 – Le manipulateur est maintenant de couleur rouge.

```
manipulator.setPosition(haptic_client.getPosition());
```

FIGURE 10 – Ajout du client haptique.

```
//Update haptic simulation here!!
for (unsigned int p = 0; p < mesh.particles.size(); p++)
{
    if (mesh.particles[p].pos.distance(manipulator.getPosition()) < manipulator.getRadius())
    {
        retour += mesh.particles[p].pos - manipulator.getPosition();
    }
}

retour = -retour;
haptic_client.setForce(retour/5);
```

FIGURE 11 – Ajout de la force en retour.

donné dans la FIGURE **??**. Après quelques changements dans les paramètres de simulation, nous décidons que la simulation est stable avec les mêmes valeurs que précédemment. Pour rappel, celles-ci étaient : dt = 1/50, K = 300, et n = 20.

Question 10 : Dans un premier temps, nous avons fixé tous les points en contact avec le manipulateur, à l'appui sur la touche « f ». Dans un second temps, nous avons cherché les positions des sphères qui avaient un rendu que nous avons jugé correct au lancement de la simulation. Dans un souci de visibilité, les sphères « de fixation » n'ont pas été dessinées. TODO scrshot avec les positions des sphères

Question 11 : Afin d'atteindre les objectifs fixés, nous avons fixé une dizaine de points, un time step de 1/50e de seconde, et 20 boucles avant d'afficher le résultat. En somme nous avons gardé les paramètres trouvés auparavant.

3 TP3

Question 1 :

Question 2 : Afin de mieux repérer les particules fixées, nous avons décidé de les colorer en bleu. Le résultat de cette modification est montré dans la FIGURE 15. Nous avons choisi le bleu car, selon nous, c'est une couleur qui se voyait suffisamment facilement et qui n'était pas associée à un état négatif. Image points fixés

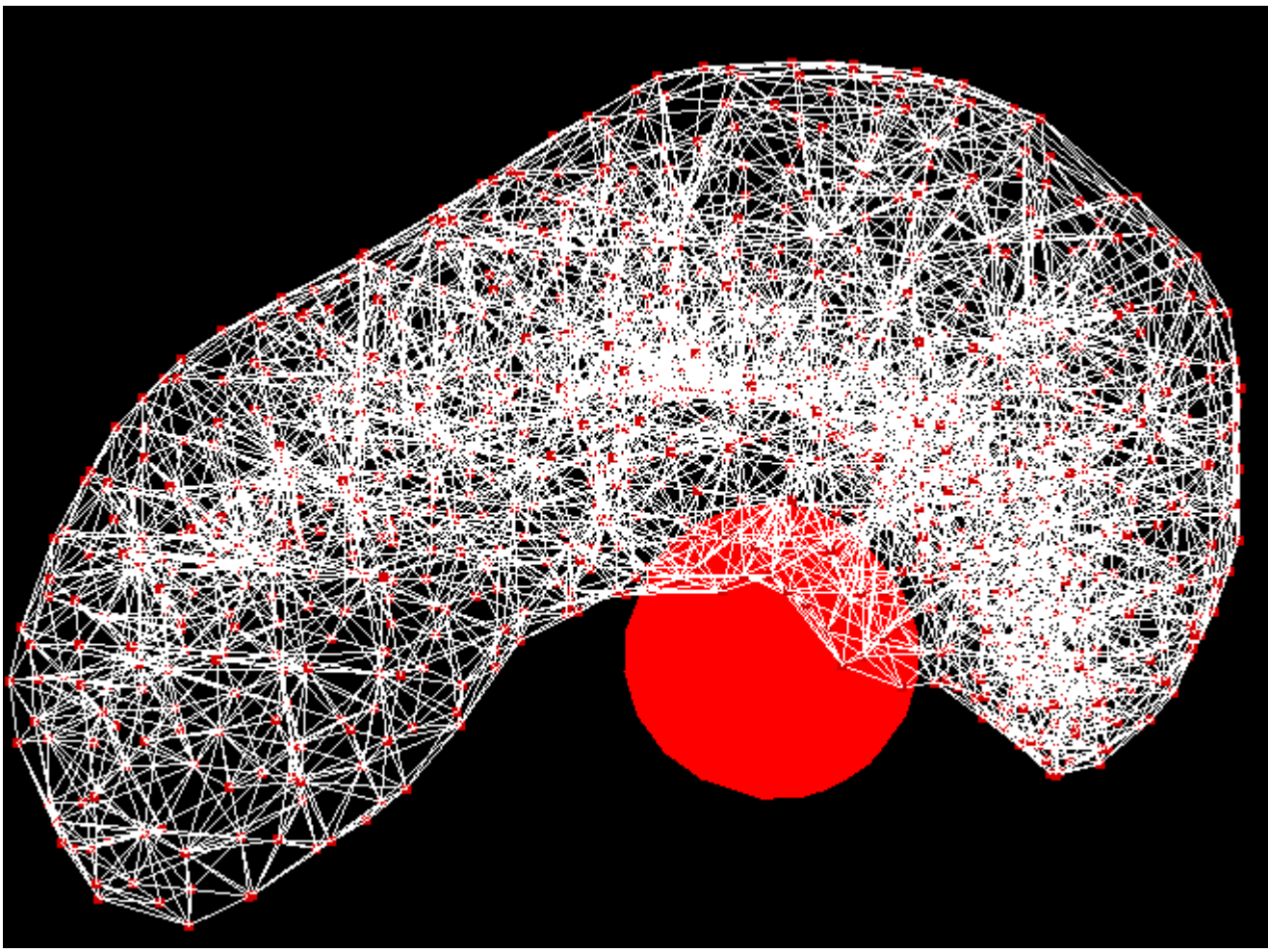

FIGURE 12 – Mesh représentant un foie.

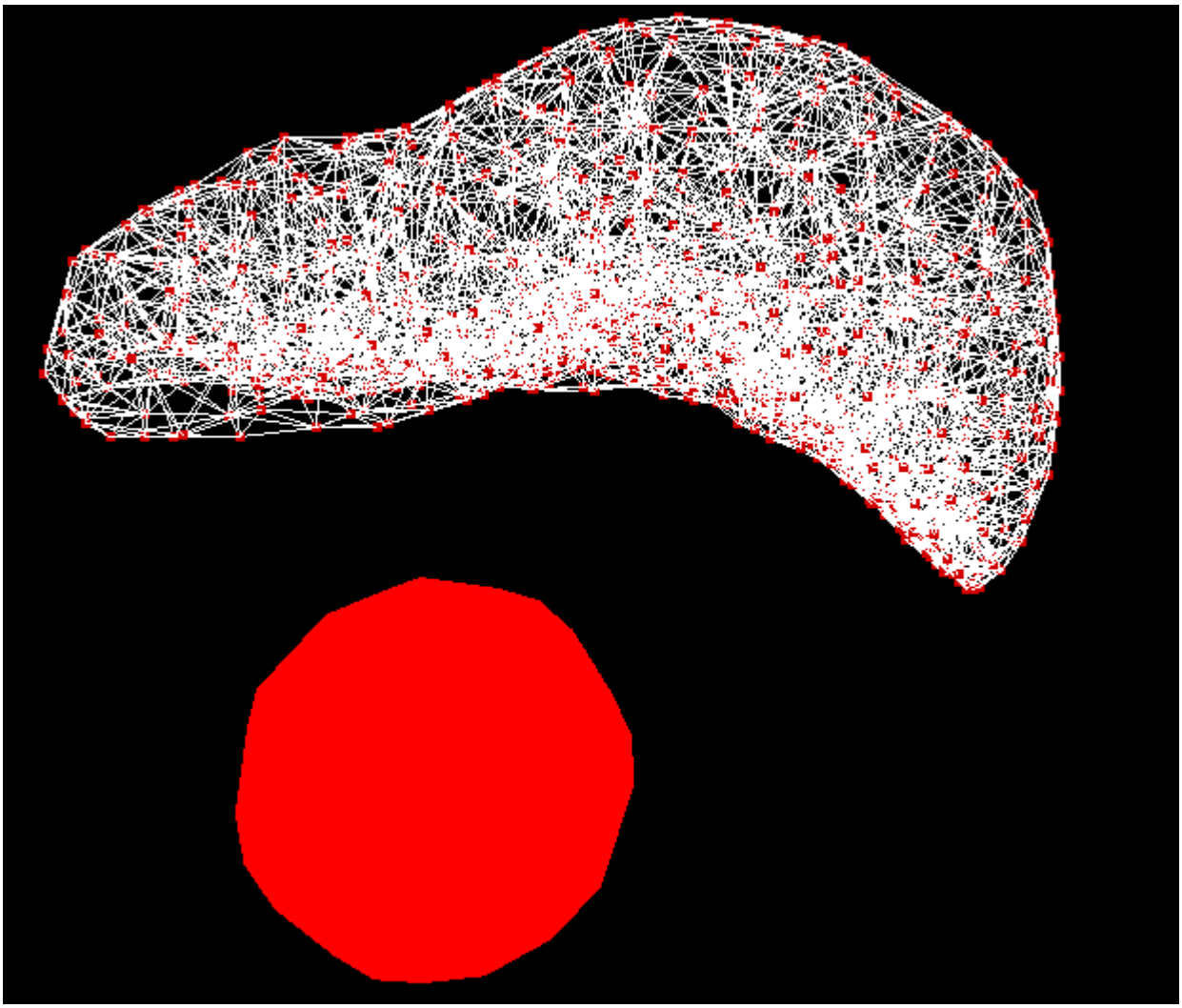

FIGURE 13 – Le mesh du foie fixé.

```
void Simulator::fixParticlesinSphere(RigidSphere* sphere)
{
    Maths::Vector3 sphere_pos = sphere->getPosition();
    Maths::Real sphere_radius = sphere->getRadius();

    for (unsigned int p = 0; p < m_Mesh->particles.size(); p++)
    {
        if (m_Mesh->particles[p].pos.distance(sphere_pos) < sphere_radius)
        {
            m_Mesh->particles[p].fixed = true;
        }
    }
}
```

FIGURE 14 – Implémentation de la fixation des points.

Question 3 : Nous avons choisi de fixer des points à l'intérieur du foie, afin que la simulation soit plus réaliste. En effet, de cette manière nous préservons la « texture » du foie en autorisant la partie supérieure à bouger, comme si le foie était posé. Nous avons également souhaité conserver les lobes du foie afin de pouvoir les soulever. Ceci rend la manipulation facile tout en gardant un réalisme dans la simulation, ce qui est le but final de ce projet.

Question 4 : Nous avons choisi de sauvegarder les particules fixées dans un fichier de type texte, nommé « save ». Si ce fichier existe déjà, il est chargé au lancement du programme. Celui-ci est écrasé à chaque appui sur la touche « s », ne laissant la possibilité que d'avoir une seule sauvegarde, à moins de la renommer manuellement. La FIGURE **??** montre la fonction de sauvegarde telle que nous l'avons implémentée. TODO code

Au chargement, nous cherchons ce fichier de sauvegarde. S'il n'existe pas à l'endroit où il est sauvegardé initialement, alors aucune particule n'est fixée et la simulation commence. En revanche, dans le cas contraire nous parcourons le fichier et fixons les particules dont le numéro apparaît. La FIGURE **??** montre le code remplissant ce rôle. TODO code

Question 5 : Afin de pouvoir alterner entre l'utilisation de la souris et le moniteur haptique, nous avons choisi d'ajouter une variable à notre code. Celle-ci peut prendre deux valeurs qui correspondent aux deux modes possibles. Ainsi, lors de l'appui sur la touche « m », on change le mode courant. Cet ajout s'accompagne bien sûr de vérifications de sa valeur afin d'exécuter la partie de code correspondant au bon mode. Image de code

Question 6 : Dans le but de contrôler avec le plus de précision possible le manipulateur, l'implémentation d'un facteur qui jouerait sur la sensibilité des manipulations était demandée. Ce facteur, nommé « scale », peut être

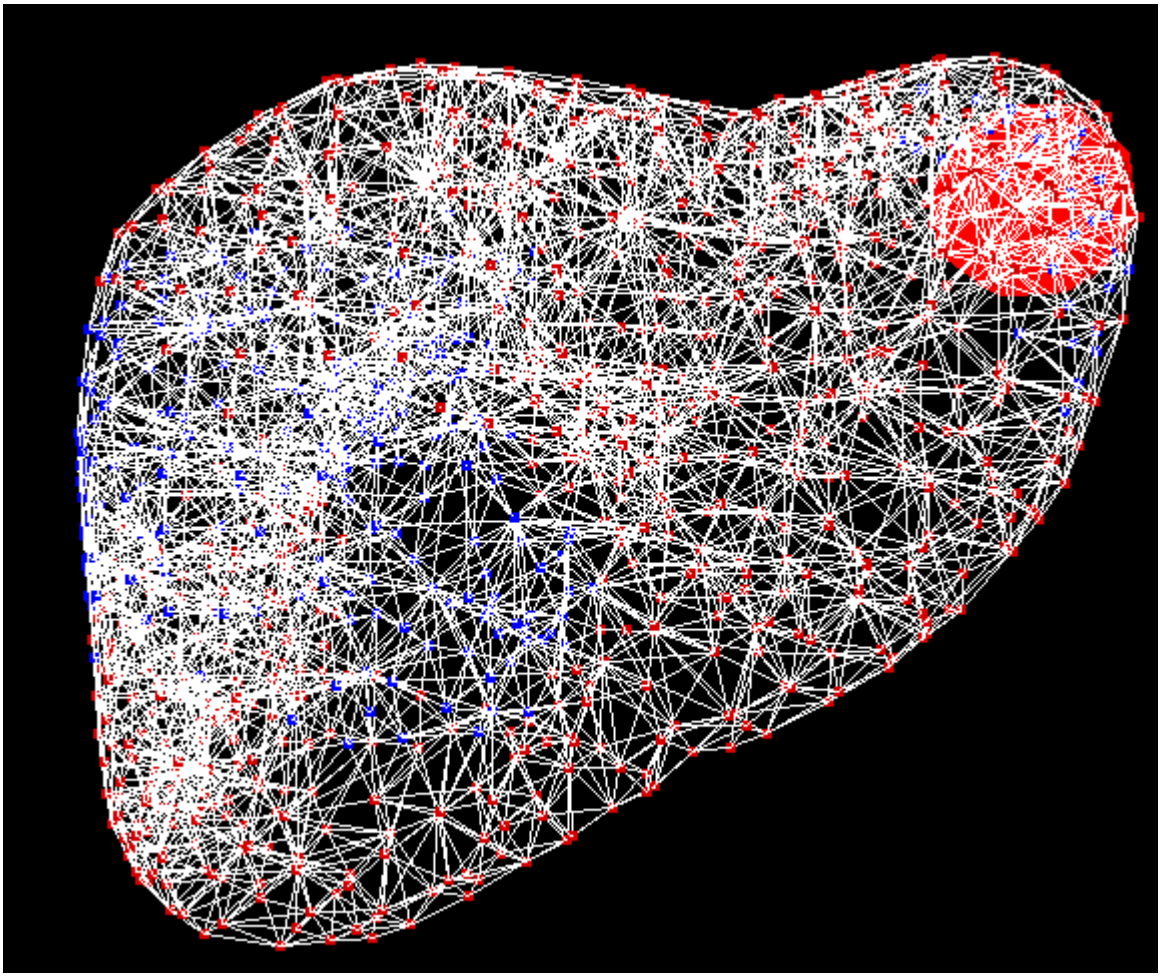

FIGURE 15 – Le mesh resemble à un rideau pendant.

```
void Simulator::saveFixedParticles()
{
    std::ofstream f;
    f.open("save.txt");

    if (f.is_open()){
        for (unsigned int p = 0; p < m_Mesh->particles.size(); p++)
        {
            if (m_Mesh->particles[p].fixed)
            {
                f << p << "\n";
            }
        }
    }
    f.close();
}
```

FIGURE 16 – Le mesh resemble à un rideau pendant.

```
void Simulator::restoreFixedParticles()
{
    std::ifstream f("save.txt");

    if (f)
    {
        std::string line;

        while (std::getline(f, line))
        {
            m_Mesh->particles[atoi(line.c_str())].fixed = true;
        }
    }
    f.close();
}
```

FIGURE 17 – Le mesh resemble à un rideau pendant.

```
#define HAPTIC 1
#define MOUSE 0
```

FIGURE 18 – Les deux modes possibles sont des entiers.

```
if (key == 'i'){
    scale_factor += 0.2;
}
if (key == 'd'){
    scale_factor -= 0.2;
}
```

FIGURE 19 – Le mesh resemble à un rideau pendant.

changé dynamiquement. Nous avons choisi les touches « i » et « d » afin d'effectuer les modifications. La FIGURE **??** montre la manière dont nous avons implémenté cette amélioration. Nos pouvons y voir que nous avons choisi un pas d'une valeur de 0.2. code

Question 7 : Voici la manière dont nous avons choisi d'implémenter une forme d'inertie à notre manipulateur lorsque le bras articulé arrive en bout de course : code

Question 8 : Nous avons choisi d'appuyer sur un des boutons du bras articulé afin de désactiver le mouvement du manipulateur. De cette manière nous pouvons remettre le bras à la position désirée avant de reprendre la manipulation. code

Question 9 : Après ajout du Leap motion, nous pouvons visualiser la représentation de notre main à l'écran. Pour le moment, certains gestes sont reconnus, mais ne sont associés à aucune action. Pour le moment, les mouvements reconnus sont :

— Le pincement entre le pouce et l'index ;

— Le balayage de la main.

Par la suite nous déciderons d'ajouter le pincement entre le pouce et le majeur.

Question 10 : Nous avons choisi d'associer au pincement entre le pouce et l'index la translation du mesh actuel. Au pincement entre le pouce et le majeur, nous effectuons une rotation du mesh, et lors du balayage de la main, nous changeons entre les deux modes présents, à savoir le contrôle à la souris ou avec le moniteur haptique. Nous avons hésité à ajouter le mode « scalpel » dans la rotation des modes engendrée par le balayage, mais nous avons trouvé que la précision des mouvements et la qualité des manipulations pourrait s'en ressentir. En effet, si lors d'un balayage le scalpel est activé, il y a un risque que l'utilisateur coupe des liens sans le vouloir.

Pour le mode « scalpel », nous avons modélisé une sphère au bout de l'index. Celle-ci ne sert que pour des raisons de facilité d'implémentation et n'apparaît en aucun cas. Lorsque la sphère se trouve entre deux points, la laissantison est rompue. TODO

```
void leapCheckPinchGesture()
{
    if (!listener.isHandVisible()) return;

    Vector3 thumb = listener.getFingerTipPosition(0);
    Vector3 index = listener.getFingerTipPosition(1);
    Vector3 middle = listener.getFingerTipPosition(2);

    float dist = thumb.distance(index);
    float dist2 = thumb.distance(middle);

    if (dist < 0.3f)
    {
        std::cout << "Pinch between thumb and index detected!!! " << dist << std::endl;

    }
    else
    {
        thumbPos = thumb;
    }
}
```

FIGURE 20 – Le pinch est reconnu par le Leap motion.

```
void leapCheckSwipeGesture()
{
    float speed;
    Vector3 direction;
    int finger;

    if (listener.isSwipe(speed, direction, finger))
    {
        std::cout << "[LeapMotion]" << " Swipe gesture(" << finger << ") : " << speed << ","
    }

}
```

FIGURE 21 – Le swipe est déjà reconnu par le Leap motion.

```cpp
void Simulator::translateMesh(Maths::Vector3 t)
{
    for (unsigned int p = 0; p < m_Mesh->particles.size(); p++)
    {
        m_Mesh->particles[p].pos -= t;
    }

}

void Simulator::rotateMesh(Maths::Vector3 prevPos, Maths::Vector3 currPos)
{
    // get rotation quaternion
    Maths::Quaternion q;
    q = prevPos.getRotationTo(currPos);

    for (unsigned int p = 0; p < m_Mesh->particles.size(); p++)
    {
        m_Mesh->particles[p].pos = q * m_Mesh->particles[p].pos;
    }
}
```

FIGURE 22 – Implémentation de la rotation et de la translation.

```
void leapCheckPinchGesture()
{
    if (!listener.isHandVisible()) return;

    Vector3 thumb = listener.getFingerTipPosition(0);
    Vector3 index = listener.getFingerTipPosition(1);
    Vector3 middle = listener.getFingerTipPosition(2);

    float dist = thumb.distance(index);
    float dist2 = thumb.distance(middle);

    if (dist < 0.3f)
    {
        std::cout << "Pinch between thumb and index detected!!! " << dist << std::endl;
        Vector3 diff = thumbPos - thumb;
        thumbPos = thumb;
        simulator.translateMesh(diff);
    }
    else if (dist2 < 0.3f)
    {
        std::cout << "Pinch between thumb and middle detected!!! " << dist << std::endl;
        simulator.rotateMesh(thumbPos, thumb);
        thumbPos = thumb;
    }
    else
    {
        thumbPos = thumb;
    }
}
```

FIGURE 23 – L'appel des fonctions précédentes.

```
void leapCheckSwipeGesture()
{
    float speed;
    Vector3 direction;
    int finger;

    if (listener.isSwipe(speed, direction, finger))
    {
        std::cout << "[LeapMotion]" << " Swipe gesture(" << finger << ") : " << speed << ","
    }

}
```

FIGURE 24 – Le swipe est déjà reconnu par le Leap motion.