

# Guide du programmeur

Antony Leclerc

Groupe 2231

Sciences Informatiques et Mathématiques, Cégep de Sherbrooke

Rapport remis à l'enseignant Régis Lamothe, le 12 mai 2023

## Introduction

## Parties scientifiques

### *Lois fondamentales newtonienne*

#### A. Mise en situation

Toute simulation physique qui se respecte doit être capable de représenter adéquatement les lois fondamentales de Newton. Effectivement, ces lois arrivent à représenter avec un haut niveau de fidélité les comportements de corps physique dans l'espace lorsqu'ils sont soumis à des forces. L'objectif de cette partie scientifique est donc de réussir à créer les fondations de notre simulation dans *Unity* afin d'être en mesure d'utiliser les lois de Newton.

#### B. Théorie

##### I. Il y a trois lois fondamentales de Newton :

**1 : Le principe d'inertie** : Si la force résultante sur un corps est nulle, l'objet gardera une vitesse constante. Ainsi, la force résultante ( $F_r^{\rightarrow}$ ) doit respecter cette équation :

$$F_r^{\rightarrow} = \Sigma F^{\rightarrow} = 0^{\rightarrow}$$

**2 : Le principe d'accélération** : L'accélération ( $a^{\rightarrow}$ ) d'un corps est parallèle et directement proportionnelle à la force résultante ( $F_r^{\rightarrow}$ ) et inversement proportionnelle à la masse ( $m$ ) du corps. Cette loi est généralement représentée comme ceci :

$$F_r^{\rightarrow} = ma^{\rightarrow}$$

**3 : Le principe d'action réaction** : Si le premier corps exerce une force  $F_1^{\rightarrow}$  sur le deuxième corps, alors le deuxième corps exerce une force  $F_2^{\rightarrow}$  égale et opposée sur le premier corps :

$$F_1^{\rightarrow} = - F_2^{\rightarrow}$$

Référence des trois lois de Newtons :

[https://media4.obspm.fr/public/ressources\\_lu/pages\\_forces/force-gravitation-masse-exercices-1.html](https://media4.obspm.fr/public/ressources_lu/pages_forces/force-gravitation-masse-exercices-1.html)

<http://www2.ift.ulaval.ca/~dupuis/Modelisation%20et%20animation%20par%20ordinateur%20IFT-66819%20et%20IFT-22726/Simulation%20de%20corps%20rigides/Simulation%20de%20corps%20rigides.pdf>

II. Algorithme théorique de l'implémentation des lois newtoniennes :

**Procédure générale pour gérer un mouvement dynamique dans le cas de particules isolées :**

**1<sup>er</sup> cas :** La force nette est constante dans le temps

- 1.** Déterminez les forces (orientation et grandeur) agissant sur un objet.
- 2.** Calculez la force nette  $\mathbf{F}$  (orientation et grandeur) agissant sur un objet.
- 3.** Calculez l'accélération  $\mathbf{a}$  due à la force nette  $\mathbf{F}$  de l'objet.
- 4.** Tenir compte de l'accélération pour déterminer la vitesse au temps  $t$ :  
$$\mathbf{v}(t) = \mathbf{v}_0 + \mathbf{a} (t - t_0) \quad \text{où } \mathbf{v}_0 \text{ est la vitesse de l'objet au temps } t_0 \text{ avant l'application de } \mathbf{F}.$$
- 5.** Déterminer la position au temps  $t$  de l'objet :  
$$\mathbf{P}(t) = \mathbf{P}_0 + \mathbf{v}_0 (t - t_0) + \frac{1}{2} \mathbf{a} (t - t_0)^2 \quad \text{où } \mathbf{P}_0 \text{ est un point de l'objet.}$$

**Note :** Un choix judicieux des axes peut simplifier les calculs de la force nette.

**2<sup>ème</sup> cas :** La force nette évolue de façon discrète aux temps  $t_1, t_2, \dots, t_n, \dots$

La force nette est constante à l'intérieur d'un intervalle.

Soit  $t \in (t_i, t_{i+1}]$ ,

- 1.** Déterminez les forces (orientation et grandeur) agissant sur l'objet à  $t_i$ .
- 2.** Calculez la force nette  $\mathbf{F}_i$  (orientation et grandeur) agissant sur l'objet à  $t_i$ .
- 3.** Calculez l'accélération  $\mathbf{a}_i$  due à la force nette  $\mathbf{F}_i$  de l'objet.
- 4.** Tenir compte de l'accélération pour déterminer la vitesse au temps  $t$ :  
$$\mathbf{v}(t) = \mathbf{v}_i + \mathbf{a}_i (t - t_i) \quad \text{où } \mathbf{v}_i \text{ est la vitesse de l'objet à } t_i.$$
- 5.** Déterminer la position au temps  $t$  de chaque objet :  
$$\mathbf{P}(t) = \mathbf{P}_i + \mathbf{v}_i (t - t_i) + \frac{1}{2} \mathbf{a}_i (t - t_i)^2 \quad \text{où } \mathbf{P}_i \text{ est un point de l'objet à } t_i.$$

**3<sup>ème</sup> cas :** La force nette évolue de façon continue dans le temps

On doit opter pour une méthode approximative comme par exemple, la méthode d'EULER :

Pour chaque valeur de  $t \equiv t_i$  où  $t_i = t_{i-1} + h$ ,

- 1.** Déterminez les forces (orientation et grandeur) agissant sur l'objet à  $t_i$ .
- 2.** Calculez la force nette  $\mathbf{F}_i$  (orientation et grandeur) agissant sur l'objet à  $t_i$ .
- 3.** Calculez l'accélération  $\mathbf{a}_i$  due à la force nette  $\mathbf{F}_i$  de l'objet à  $t_i$ .
- 4.** Déterminer une vitesse approximative  $\mathbf{v}(t_{i+1})$  à  $t_{i+1}$ :  
$$\mathbf{v}(t_{i+1}) = \mathbf{v}(t_i) + h \mathbf{a}_i$$
- 5.** Déterminer la position au temps  $t_{i+1}$  de chaque objet :  
$$\mathbf{P}(t) = \mathbf{P}_i + \frac{1}{2} (\mathbf{v}(t_i) + \mathbf{v}(t_{i+1})) h \quad \text{où } \mathbf{P}_i \text{ est un point de l'objet à } t_i.$$

Référence des trois lois de Newtons :

<http://www2.ift.ulaval.ca/~dupuis/Modelisation%20et%20animation%20par%20ordinateur%20IFT-66819%20et%20IFT-22726/Simulation%20de%20corps%20rigides/Simulation%20de%20corps%20rigides.pdf>

### C. Compréhension

- I. Tout cégépien qui se respecte ayant reçu des cours de physique dynamique comprend assez bien les lois fondamentales de Newtons. Ce sont des lois très utiles qui permettent de prédire le comportement de différents corps dans les systèmes de forces. La première loi (la loi de l'inertie) est la plus simple et la plus facile à imaginer. Effectivement, en l'absence de force sur un objet, l'objet ne va simplement pas changer son comportement. Par exemple, en plaçant une pomme immobile dans le vide de l'espace, cette pomme ne bougera pas. Elle ne sera soumise à aucune force, donc son comportement restera exactement le même. Cependant, si on lui donne un petit coup, on lui aura appliqué une force pendant un court instant de temps ce qui changera sa vitesse. Si on arrête de la toucher, la pomme va simplement conserver sa dernière vitesse en poursuivant sa route dans le vide de l'espace. La deuxième loi est aussi assez facile à imaginer. Effectivement, en appliquant une force sur un objet, on change sa vitesse en l'accélérant et les objets plus lourds sont plus difficiles à accélérer. Par exemple, il est plus difficile de lancer une voiture qu'une balle de baseball car la voiture est plus massive (la balle de baseball ira beaucoup plus loin!). La troisième loi de Newton est la moins intuitive. Elle stipule que toute action engendre une réaction égale et opposée. Par exemple, si on frappe un mur de brique avec une batte de baseball en aluminium, le mur recevra le coup, mais nous ressentiront aussi le contrecoup dans notre bras, car le mur aura appliqué une force égale et opposée à notre coup. Ce sont ces trois lois qui forment la base de la compréhension de la physique dynamique .

- II. Bien que les lois soient assez simples, il y a de nombreuses problématiques qui émergent au moment où il faut commencer à les simuler. Effectivement, le plus gros problème provient de l'échelle de temps. Dans notre réalité, le temps est divisible à l'infini : nous sommes sur une échelle réelle où il est possible d'analyser une fraction, d'une fraction, d'une fraction d'une seconde :

*Entre 0 et 1 il y a une infinité de nombres : 0... 0, 00001... 0, 0001... 0. 01... 1*

Hors, la puissance de calculs d'un ordinateur est limitée. Il est tout simplement impossible d'effectuer une infinité d'opérations et de calculs à chaque seconde. Ainsi, il faut se limiter à une échelle de temps *discrète*. C'est une échelle de temps fixe qui avance à chaque "tick". Par

exemple, si on calcule 60 "tick" par seconde, chaque "tick" représente une intervalle d'environ 17 milliseconde. Avec ces "tick"s il est possible d'évaluer les effets des lois de Newton :

Nous savons que l'accélération correspond à la variation de vitesse dans le temps ( $\frac{m}{s^2}$  ou  $\frac{v}{s}$ ).

Ainsi, en connaissant la valeur temps ( $t$ ) du "tick" il est possible d'approximer la variation de vitesse tel que  $\Delta v = at$  et donc le nouvelle vitesse du corps en question à chaque "tick" est :

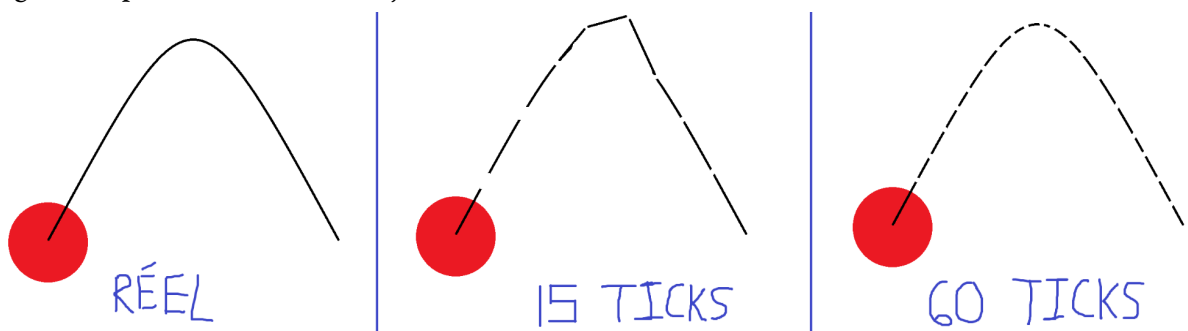
$$v_f = v_i + at$$

Aussi, nous savons que la vitesse correspond à la variation de la position dans le temps ( $\frac{m}{s}$ ).

Alors, il est possible d'approximer la variation de position tel que  $\Delta x = vt$  et donc la nouvelle position du corps en question à chaque "tick" est :

$$x_f = x_i + vt$$

Cette méthode est une approximation pure mais plus la quantité de "tick" par seconde s'agrandit, plus la simulation est juste :



*Exemple d'une simulation d'un lancer d'une balle selon le nombre de "tick"s*

La méthode de simulation est donc la suivante à chaque "tick" :

1. Faire une sommation des forces afin de trouver la force résultante
2. Déterminer l'accélération en divisant la force résultante avec la masse de l'objet en question
3. Déterminer la variation de vitesse en évaluant l'accélération selon le "tick" et faire changer la vitesse
4. Déplacer l'objet selon sa vitesse et le "tick"

#### D. Implémentation

Premièrement, nous avons créé une structure de données renfermant les propriétés propre à chaque corps :

```

/*
 * Filename : BasicPhysicsObjects
 *
 * Goal : Encapsulates and calculates physics properties of the object
 *
 * Requirements : Attach this script to physical objects of the Scene
 */
Unity Script (14 asset references) | 38 references
public class BasicPhysicsObject : MonoBehaviour
{
    //Force résultante
    Vector3 resultingForce;

    //Vitesse actuelle
    Vector3 velocity;

```

Chaque corps possède une variable de sa vitesse actuelle, une variable de son accumulation de force actuelle. Chaque corps possède aussi une référence à sa position dans l'espace. C'est cependant une propriété qui est inutile à mettre dans la classe, car Unity la définit déjà comme le **transform.position**.

```

1 reference
public void UpdateState(float timeStep)
{
    if (isStatic) { velocity = Vector3.zero; angularVelocity = 0; resultingForce = Vector3.zero; torque = 0; return; }
    else if (lockRoll) { torque = 0; angularVelocity = 0; }

    Vector3 acceleration = resultingForce / collider.GetMass(); //Détermine la valeur de l'accélération
    float angularAcceleration = torque / collider.GetInertia();

    velocity += acceleration * timeStep; //Ajoute la variation de vitesse selon l'accélération et le timestep
    angularVelocity += angularAcceleration * timeStep;

    transform.position += velocity * timeStep; //Change la position selon le timestep et la vitesse
    transform.Rotate(Vector3.forward * angularVelocity * Mathf.Rad2Deg * timeStep);

    //Reset les forces pour le next updateCall
    resultingForce = Vector3.zero;
    torque = 0;
}

```

Chaque corps possède aussi la fonction *UpdateState* qui va changer sa vitesse et sa position dépendamment de l'accélération et du "tick".

```

1 reference
public void ApplyForceGravity()
{
    //F = mg

    resultingForce += Vector3.down * UniversalVariable.GetGravity() * collider.GetMass();
}

```

Chaque corps possède aussi la fonction *ApplyForceGravity* qui va ajouter à la force résultante, la force gravitationnelle de la simulation selon le "tick".

Deuxièmement nous avons créé le *Loop* continu des calculations physiques :

```

/*
 * Filename : PhysicsManager
 *
 * Goal : Central authority of the physic calculations. The PhysicsManager holds references to every physics object in the scene to allow interactions
 *
 * Requirements : Put a single instance in a Scene attached to an empty GameObject
 */
@ Unity Script (1 asset reference) | 1 reference
public class PhysicsManager : MonoBehaviour
{
    //Change the variable numberOfStepsPerSecond to change the timerate calculations

    private int numberOfStepsPerSecond = 100; //Quantité de tick par seconde
    private float stepLength; //Longueur des ticks en sec
    private float numberOfUpdateCounter = 0; //Compteur de seconde réel afin d'effectuer les tick au bon moment
}

```

```

//Update the physics objects on a fixed time rate
@ Unity Message | 0 references
public void Update()
{
    numberOfUpdateCounter += UniversalVariable.GetTime() * Time.deltaTime / stepLength;

    while (numberOfUpdateCounter > 1)
    {
        PhysicCalculations();

        JointPhysicCalculations();

        numberOfUpdateCounter--;
    }
}

```

```

//Simulate all the physics behaviours
1 reference
private void PhysicCalculations()
{
    //ApplyForces
    for (int i = 0; i < objects.Count; i++)
    {
        physicObjects[i].UpdateState(stepLength);
meshColliders[i].UpdateColliderOrientation();
        physicObjects[i].ApplyForceGravity();
    }
}

```

Le Loop est appelé le *PhysicManager* et appelle essentiellement la fonction *PhysicCalculations* à chaque "tick" afin de faire avancer la simulation dans le temps. La fonction *PhysicCalculations* appelle la fonction *UpdateState* et *ApplyForceGravity* des objets.

#### E. Énumérations des problèmes et des solutions

- I. Il n'y a qu'un seul problème pertinent résultant de notre implémentation des lois newtoniennes et c'est le problème de performance. Effectivement, par la nature de notre projet, nous souhaitons représenter fidèlement le comportement des corps dans l'espace. Il faut donc que ces comportements soient réalistes. Si on veut que les comportements soient réalistes, il faut mettre **beaucoup** de "tick" par seconde (au minimum 60). Hors, des calculs d'interactions entre des dizaines de corps au moins 60 fois par secondes est très taxant pour n'importe quel processeur. Il arrive souvent que le framerate du jeu s'en retrouve affecté. Il est donc nécessaire de trouver un équilibre entre fidélité de la réalité et performance de la simulation.
- II. Il n'y a pas réellement de solution possible. La seule manière d'atténuer cette problématique est d'optimiser le code qui se répète à chaque "tick" afin de minimiser les pertes de performances.

#### F. Retour sur les objectifs personnels et professionnels

Mon objectif professionnel était l'approfondissement de mes connaissances et surtout de ma compréhension des lois fondamentales Newtoniennes afin d'avoir une longueur d'avance dans mes études universitaires et je considère cela comme étant mieux réussi puisque je peux schématiser et imaginer des systèmes de forces plus complexes par moi même. Mon objectif personnel d'apprendre à utiliser la plateforme Unity à cause de tous les avantages qu'elle possède a aussi été atteint car maintenant je suis capable d'utiliser les outils de base, de debugger et je connais une base de C#, le langage de programmation de Unity.



## *Dynamique rotationnelle*

### A. Mise en situation

Les lois newtoniennes forment une bonne base de la physique dynamique. Cependant, elles oublient une couche de complexité qui est inhérente à notre réalité : les objets peuvent tourner. C'est cette particularité que permet la dynamique rotationnelle permet de simuler. L'objectif de l'implémentation de cette partie scientifique est donc de coder une simulation qui permet de faire tourner des objets physiques selon les lois régissant la physique classique. Les objets physiques simulés seront limités à des polygones convexes de tous genres et des cercles.

### B. Théorie

#### I. Les lois de dynamique rotationnelles sont homologues aux lois de Newton :

Le moment de force ( $\tau$ ), connu aussi sous le nom de torque est l'homologue à la force en physique newtonienne. C'est le facteur appliqué à l'objet qui va induire la rotation.

Le moment d'inertie ( $I$ ) est l'homologue à la masse en physique newtonienne. C'est le facteur des corps qui va s'opposer à la rotation.

L'accélération angulaire ( $\alpha$ ), la vitesse angulaire ( $\omega$ ) et la position angulaire ( $\theta$ ) sont les homologues respectifs de l'accélération, la vitesse et la position.

Il n'y a que les deux premières lois de Newton qui peuvent se transcrire en dynamique rotationnelle :

**1 : Le principe d'inertie :** Un objet va conserver sa vitesse angulaire si le torque résultant appliqué sur l'objet est nul. Il est possible de représenter l'équation si le torque résultant est nul de cette manière :

$$\tau_r = \Sigma \tau = 0$$

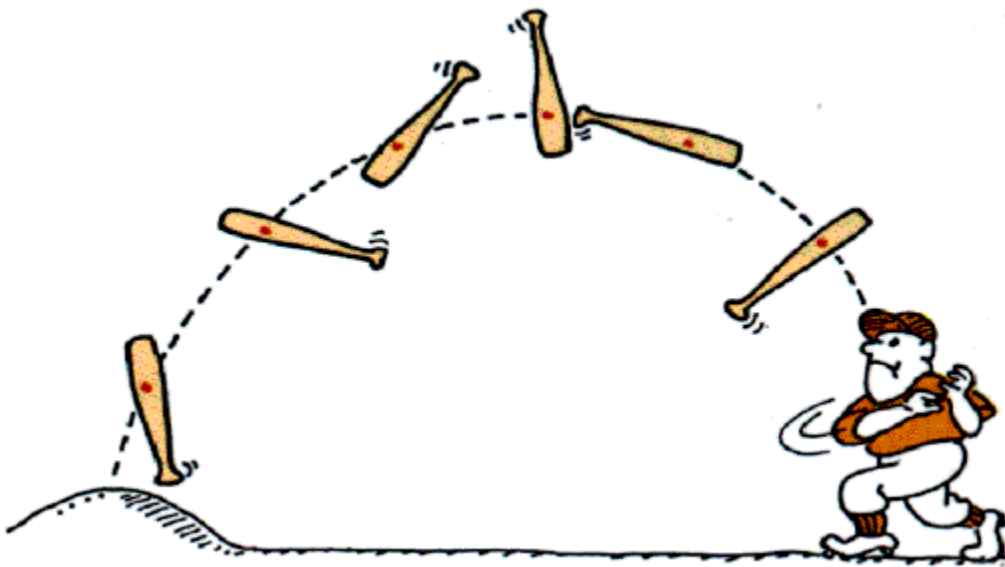
**2: Le principe d'accélération :** Un objet va subir une accélération angulaire proportionnelle au torque appliqué dessus et inversement proportionnel à son moment d'inertie. Cette loi est représentée de cette manière :

$$\tau_r = I\alpha$$

Il y a cependant un problème car en physique classique, il n'est pas possible d'appliquer directement du torque sur un objet. Il faut appliquer une force qui va ensuite appliquer du torque sur l'objet. Il est donc nécessaire d'évaluer la relation entre le torque et une force. Selon les lois sur la dynamique rotationnelle, le torque correspond au produit scalaire entre la distance du centre de masse ( $r^{\rightarrow}$ ) et la force appliqué ( $F^{\rightarrow}$ ) :

$$\vec{\tau} = \vec{r} \times \vec{F}$$

Nous avons introduit le principe de centre de masse sans le définir. Le centre de masse est simplement le point de pivot de tout objet. Il est le point d'un objet où toute force appliquée dessus va résulter en une translation et aucune rotation.



*Excellente illustration qui démontre que la batte de baseball va pivoter autour de son centre de masse qui est le point rouge*

La position du centre de masse ( $\vec{x}$ ) d'un objet quelconque peut être calculée en prenant la moyenne de la sommation de tous les points massiques ponctuels ( $m_i$ ) proportionnels à leur distance de l'origine ( $x_i$ ) :

$$\vec{x} = \left( \frac{1}{\sum_{i=1}^n m_i} \right) \cdot \sum_{i=1}^n m_i x_i$$

Références de la théorie de la dynamique rotationnelle :

[https://en.wikipedia.org/wiki/List\\_of\\_moments\\_of\\_inertia](https://en.wikipedia.org/wiki/List_of_moments_of_inertia)

<https://en.wikipedia.org/wiki/Torque>

<http://www2.ift.ulaval.ca/~dupuis/Modelisation%20et%20animation%20par%20ordinateur%20IFT-66819%20et%20IFT-22726/Simulation%20de%20corps%20rigides/Simulation%20de%20corps%20rigides.pdf>

## II. Algorithmes théoriques des lois de la dynamique rotationnelle :

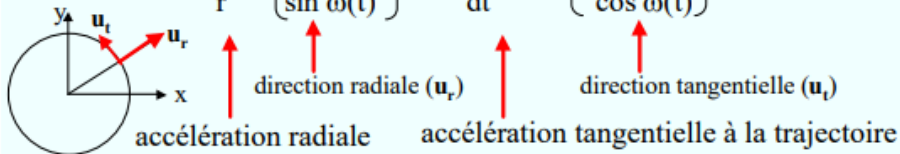
### Mouvement circulaire non uniforme et force centripète :

$$\mathbf{C}(t) = \begin{pmatrix} r \cos \omega(t) \\ r \sin \omega(t) \end{pmatrix} \quad \text{avec } \|\mathbf{C}(t)\| = r$$

$$\Rightarrow \mathbf{V}(t) = \begin{pmatrix} -\omega'(t) r \sin \omega(t) \\ \omega'(t) r \cos \omega(t) \end{pmatrix} \quad \text{avec } \|\mathbf{V}(t)\| = \omega'(t) r = v(t)$$

La vitesse peut varier en grandeur et en direction.

$$\Rightarrow \mathbf{A}(t) = -\frac{v^2(t)}{r} \begin{pmatrix} \cos \omega(t) \\ \sin \omega(t) \end{pmatrix} + \frac{dv(t)}{dt} \begin{pmatrix} -\sin \omega(t) \\ \cos \omega(t) \end{pmatrix}$$



Si  $v(t) = v \Rightarrow$  accélération tangentielle nulle

$$\|\mathbf{A}(t)\| = v^2 / r$$

$\Rightarrow$  force centripète orientée vers le centre du cercle  
de grandeur :  $m v^2 / r$ .

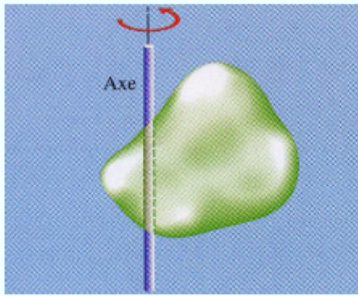
### Mouvement d'un objet quelconque faisant aussi intervenir une rotation autour d'un axe fixe

- ❖ Jusqu'à maintenant, nous avons limité le mouvement d'un corps à une translation pure comme s'il s'agissait d'une particule.
- ❖ Si la force résultante, appliquée à un objet, n'est pas alignée avec son centre de masse, alors l'objet subit aussi un mouvement de rotation autour d'un axe.
- ❖ L'étude générale du mouvement de rotation est assez complexe.

Nous allons limiter notre étude au mouvement de rotation d'un corps rigide autour d'un axe fixe.

Un axe qui reste fixe p/r au corps et dont la direction est fixe p/r à un référentiel d'inertie.

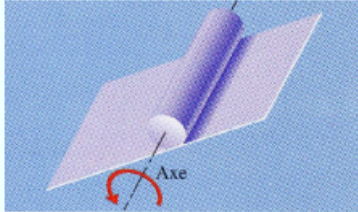
La forme et les dimensions de l'objet sont fixes (non déformables).



Axe de rotation fixe en **position** et en **direction**

Le corps est soumis à un **mouvement de rotation pur** (aucune translation).

Toutes les particules du corps suivent des trajectoires circulaires centrées sur l'axe de rotation.



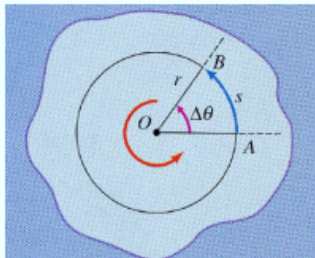
Axe de rotation fixe en **direction** seulement.

Ex. : une boule de billard qui roule sur la surface de la table.

❖ Avec un mouvement de translation, nous avons retenu les notions de **position linéaire**, **vitesse linéaire** et **accélération linéaire**. <sup>30</sup>

### Introduction de variables dites **angulaires**

❖ Puisque tous les points d'un corps en rotation n'ont pas la même vitesse linéaire ni la même accélération linéaire, nous allons introduire de nouvelles variables dites **angulaires** pour représenter le mouvement de rotation autour d'un axe fixe.



**Rotation d'un angle  $\Delta\theta$  autour d'un axe fixe en O.**

déplacement angulaire  $\equiv \Delta\theta$

position angulaire  $\equiv$

si la position angulaire est  $\theta_A$  en A  
alors cela est  $\theta_A + \Delta\theta$  en B.

(analogue à la position linéaire en translation)

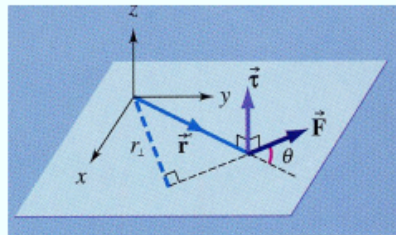
$$s = r \Delta\theta \quad \text{où } s = \text{longueur d'arc parcourue.}$$

## 1e cas : Axe de rotation fixe en position et direction

### Définition du moment de force

- ❖ Analogue d'une force dans le cas d'une rotation :  
une force produit une accélération linéaire et  
un moment de force produit une accélération angulaire.
- ❖ Capacité qu'a une force d'imprimer une rotation à un corps autour d'un axe :  
$$\tau = \mathbf{r} \times \mathbf{F}$$
- ❖ Le moment de force dépend à la fois de la direction de la force  $\mathbf{F}$  et de la position  $\mathbf{r}$  de son point d'application p/r à un point d'origine O.

Il ne s'agit pas de la distance  $\perp$  à l'axe de rotation.



### Procédure générale pour gérer un mouvement dynamique dans le cas d'objets quelconques limités à une rotation pure :

À l'aide du moment d'inertie  $I$  et du moment de force  $\tau$ , on obtient :

$$\tau = I \alpha \quad \text{ou encore,} \quad \alpha = \tau / I$$

où  $\tau$  est le moment de force extérieur résultant sur le corps,  
 $\tau$  et  $\alpha$  sont dans la direction de l'axe de rotation.

**1<sup>er</sup> cas :** Le moment de force net est constant dans le temps

1. Calcul du moment d'inertie ( $I$ ) p/r à l'axe de rotation.
2. Calcul du moment de force externe résultant sur le corps ( $\tau$ ).
3. Calcul de  $\alpha = \tau / I$ .
4. Tenir compte de  $\alpha$  pour déterminer la vitesse angulaire au temps  $t$  :  
$$\omega(t) = \omega_0 + \alpha (t - t_0) \quad \text{où } \omega_0 \text{ est la vitesse angulaire du corps au temps } t_0 \text{ avant l'application de } \mathbf{F}.$$
5. Déterminer la position angulaire au temps  $t$  de chaque objet :  
$$\theta(t) = \theta_0 + \omega_0(t - t_0) + \frac{1}{2} \alpha (t - t_0)^2 \quad \text{où } \theta_0 \text{ est la position angulaire à } t_0.$$
6. Appliquez une rotation d'un angle  $\theta(t)$  autour de l'axe de rotation.<sup>41</sup>

**2<sup>ième</sup> cas :** Le moment de force net évolue de façon discrète aux temps  $t_1, t_2, \dots, t_n, \dots$

Le moment de force net est constant à l'intérieur d'un intervalle.

Référence de l'implémentation de l'algorithme théorique de la dynamique rotationnel :

<https://www.toptal.com/game/video-game-physics-part-i-an-introduction-to-rigid-body-dynamics>


<http://www2.ift.ulaval.ca/~dupuis/Modelisation%20et%20animation%20par%20ordinateur%20IFT-66819%20et%20IFT-22726/Simulation%20de%20corps%20rigides/Simulation%20de%20corps%20rigides.pdf>

**Calculating the area and centroid of a polygon**

Written by Paul Bourke  
July 1988  
See also: [centroid.pdf](#) by Robert Nurmberg

**Area**

The problem of determining the area of a polygon seems at best messy but the final formula is particularly simple. The result and sample source code (C) will be presented here. Consider a polygon made up of line segments between  $N$  vertices  $(x_i, y_i)$ ,  $i=0$  to  $N-1$ . The last vertex  $(x_N, y_N)$  is assumed to be the same as the first, ie: the polygon is closed.

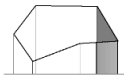


The area is given by


$$A = \frac{1}{2} \sum_{i=0}^{N-1} (x_i y_{i+1} - x_{i+1} y_i)$$

Note for polygons with holes. The holes are usually defined by ordering the vertices of the enclosing polygon in the opposite direction to those of the holes. This algorithm still works except that the absolute value should be taken after adding the polygon area to the area of all the holes. That is, the holes areas will be of opposite sign to the bounding polygon area. The sign of the area expression above (without the absolute value) can be used to determine the ordering of the vertices of the polygon. If the sign is positive then the polygon vertices are ordered counter clockwise about the normal, otherwise clockwise.

To derive this solution, project lines from each vertex to some horizontal line below the lowest part of the polygon. The enclosed region from each line segment is made up of a triangle and rectangle. Sum these areas together noting that the areas outside the polygon eventually cancel as the polygon loops around to the beginning.



The only restriction that will be placed on the polygon for this technique to work is that the polygon must not be self intersecting, for example the solution will fail in the following cases.



**Centroid**


The centroid is also known as the "centre of gravity" or the "center of mass". The position of the centroid assuming the polygon to be made of a material of uniform density is given below.  $A_0$  in the calculation of the area above,  $A_0$  is assumed to be  $A_0$ , in other words the polygon is closed.

$$C_x = \frac{1}{6A} \sum_{i=0}^{N-1} (x_i + x_{i+1}) (x_i y_{i+1} - x_{i+1} y_i)$$

$$C_y = \frac{1}{6A} \sum_{i=0}^{N-1} (y_i + y_{i+1}) (x_i y_{i+1} - x_{i+1} y_i)$$

**Centroid of a 3D shell described by 3 vertex facets**

The centroid  $C$  of a 3D object made up of a collection of  $N$  triangular faces with vertices  $(x_i, y_i, z_i)$  is given below.  $R_i$  is the average of the vertices of the  $i$ th face and  $A_i$  is twice the area of the  $i$ th face. Note the faces are assumed to be thin sheets of uniform mass, they need not be connected or form a solid object. This reduces to the equations above for a 2D 3 vertex polygon.

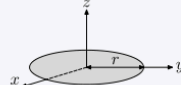
$$C = \frac{\sum_{i=0}^{N-1} A_i R_i}{\sum_{i=0}^{N-1} A_i}$$


$$R_i = (x_i + x_j + x_k) / 3$$

$$A_i = \frac{1}{2} \| (x_j - x_i) \times (x_k - x_i) \|$$

Référence de l'implémentation du calcul du centre de masse :

<http://paulbourke.net/geometry/polygonmesh/>

<p>Thin, solid disk of radius <math>r</math> and mass <math>m</math>.</p> <p>This is a special case of the solid cylinder, with <math>h = 0</math>. That</p> <p><math>I_x = I_y = \frac{I_z}{2}</math> is a consequence of the <a href="#">perpendicular axis theorem</a>.</p>		$I_z = \frac{1}{2} m r^2$ $I_x = I_y = \frac{1}{4} m r^2$
--	---	---

A polygon can be decomposed into multiple triangles. Suppose you want to find the MMOI value about a reference point (the coordinate origin) and about the center of mass. Split the polygon into multiple triangles and follow the following algorithm:

1. Loop through triangles, each with three vertices  $\mathbf{A} = (A_x, A_y)$ ,  $\mathbf{B} = (B_x, B_y)$ ,  $\mathbf{C} = (C_x, C_y)$
2. Calculated the area of the triangle

$$\text{area}(i) = \frac{1}{2}(\mathbf{A} \times \mathbf{B} + \mathbf{B} \times \mathbf{C} + \mathbf{C} \times \mathbf{A}) \quad (1)$$

where  $\mathbf{A} \times \mathbf{B} = A_x B_y - A_y B_x$  and so on with the remaining cross products.

3. Calculate the centroid of the triangle

$$\text{cen}(i) = \frac{1}{3}(\mathbf{A} + \mathbf{B} + \mathbf{C}) \quad (2)$$

4. Calculate the mass moment if inertia of the triangle (per unit mass)

$$\text{mmoi}(i) = \frac{1}{6}(\mathbf{A} \cdot \mathbf{A} + \mathbf{B} \cdot \mathbf{B} + \mathbf{C} \cdot \mathbf{C} + \mathbf{A} \cdot \mathbf{B} + \mathbf{B} \cdot \mathbf{C} + \mathbf{C} \cdot \mathbf{A}) \quad (3)$$

where  $\mathbf{A} \cdot \mathbf{B} = A_x B_x + A_y B_y$  and the same for the remaining dot products.

Références de l'implémentation du calcul du moment d'inertie:

[https://en.wikipedia.org/wiki/List\\_of\\_moments\\_of\\_inertia](https://en.wikipedia.org/wiki/List_of_moments_of_inertia)

<https://physics.stackexchange.com/questions/708936/how-to-calculate-the-moment-of-inertia-of-convex-polygon-two-dimensions>

## C. Compréhension

- I. La dynamique rotationnelle ajoute une couche de complexité à notre compréhension de la manière dont des corps évoluent dans leur environnement. Effectivement, on ne peut plus simplement traiter l'ajout de force comme de simples translations. Malgré tout, cette partie scientifique reste compréhensible. Le mouvement rotationnel d'un objet peut être défini par la variation de son angle dans le temps (vitesse angulaire ( $\omega$ )). Cette vitesse angulaire résulte d'une accélération angulaire ( $\alpha$ ) appliquée dans le temps. Cette accélération angulaire, quant à elle, résulte d'une force appliquée sur l'objet et va varier selon l'orientation et la position de la force. Par exemple, pour faire tourner une tige de métal, il vaut mieux appliquer la force perpendiculaire à la tige de métal et à son extrémité. Dans le cas où on applique la force trop proche du milieu de la tige, la tige va tourner avec moins de facilité. Par ailleurs, la facilité de tourner un objet dépend de sa forme (un cube sera plus difficile à tourner qu'une tige) et de sa masse (un objet plus lourd demandera plus de force pour une même accélération angulaire). Ces deux types d'impédance sont représentés par le concept de moment d'inertie ( $I$ ). La théorie de cette partie scientifique est donc compréhensible.

## II. Compréhension de l'algorithme

Au niveau du fonctionnement de l'algorithme, il y a beaucoup de points de similitudes avec la physique newtonienne. En théorie, la physique rotationnelle devrait se faire en continue à chaque millième de millième de seconde. Hors, un ordinateur ne possède aucunement la capacité de faire autant d'opérations par secondes. Il est donc nécessaire d'effectuer la simulation à l'aide d'intervalles de temps discrets. Cet intervalle de temps devrait être suffisamment court afin qu'il y ait au moins une soixante opérations par seconde ce qui permet d'approximer le comportement réel des objets. Ainsi, nous allons effectuer un loop à chaque intervalle de temps fixe dans lequel nous allons faire évoluer les propriétés rotationnelles des objets. Voici ce à quoi le loop doit ressembler pour chaque objet :

1. Évaluer le torque de chaque force appliqué sur l'objet selon la formule :
$$\vec{\tau} = \vec{r} \times \vec{F}$$
2. Faire la sommation de tous les torques pour obtenir le torque résultant
3. Calculer l'accélération angulaire selon la formule :  $\tau_r = I\alpha$
4. Calculer la variation de vitesse angulaire associé à l'accélération angulaire dans le temps (temps entre chaque intervalle ( $t$ )) selon cette formule :  $\Delta\omega = \alpha t$
5. Calculer la nouvelle vitesse angulaire en additionnant la variation :
$$\omega_f = \omega_i + \Delta\omega$$
6. Calculer la variation de position angulaire associé à la vitesse angulaire dans le temps ( $t$ ) selon cette formule :  $\Delta\theta = \omega t$
7. Calculer la nouvelle position angulaire en additionnant la variation :
$$\theta_f = \theta_i + \Delta\theta$$
8. Changer la rotation de l'objet selon sa position angulaire
9. Patienter pendant l'intervalle de temps, puis retour à l'étape 1

Il y a cependant un problème associé à cet algorithme. Effectivement nous ne connaissons pas le moment d'inertie ( $I$ ) pour chaque objet. Il est facile d'attribuer une masse arbitraire à un objet quelconque, mais le moment d'inertie dépend de la forme générale de l'objet en question. Pour les cercles, c'est simple, leur moment d'inertie dépend de leur masse( $m$ ) et de leur rayon( $r$ ):  $I = \frac{1}{2}mr^2$ . Cependant, pour les polygones complexes, il n'y a pas de formule absolue. Il faut donc développer un algorithme qui calcule le moment d'inertie d'un objet de n'importe quelle forme. Voici l'algorithme pour déterminer le moment d'inertie pour les polygones :



1. Diviser le polygone convexe en triangles selon ses points
2. Trouver la masse de chaque triangle
3. Trouver le moment d'inertie de chaque triangle par unité de masse selon cette formule ( $A^{\rightarrow}$  est le point 1 et  $B^{\rightarrow}$  est le point 2 du triangle, le point 3 est par définition (0,0) donc inutile) :  

$$I_{triangleParUnitéDeMass} = (A^{\rightarrow} \cdot A^{\rightarrow} + B^{\rightarrow} \cdot B^{\rightarrow} + A^{\rightarrow} \cdot B^{\rightarrow})/6$$
4. Multiplier le moment d'inertie par unité de masse par la masse
5. Faire la sommation du moment d'inertie de tous les triangles

Par ailleurs, un autre problème est qu'il est nécessaire d'identifier le centre de masse ( $C_m$ ) d'un objet afin de déterminer son point de pivot. Il faut donc développer un algorithme qui permet de déterminer le centre de masse d'un objet de n'importe quelle forme. Voici l'algorithme pour déterminer la position du centre de masse :

Nous connaissons tous les points relatifs de chaque objet ( $x^{\rightarrow}$ ). Aussi, chaque objet possède une masse répartie équitablement sur toute sa surface ce qui évite de devoir faire des intégrales de variation de masse. Ainsi, pour obtenir le centre de masse de chaque objet :

1. Trouver l'aire du polygone quelconque ( $A$ )
2. Trouver la position  $X$  du centre de masse qui correspond à :  

$$C_x = \frac{1}{6A} \sum_{i=0}^{N-1} (x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$$
3. Trouver la position  $Y$  du centre de masse qui correspond à :  

$$C_y = \frac{1}{6A} \sum_{i=0}^{N-1} (y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$$
4. Déplacer le point de pivot du polygone afin qu'il tourne autour de son centre de masse

## D. Implémentation

### I. Loop de dynamique rotationnelle

Les propriétés rotationnelles sont encapsulées dans un objet du nom de *BasicPhysicObject*. Cet objet contient la position angulaire actuelle, la vitesse angulaire actuelle et le torque actuel. À chaque intervalle de temps de la simulation, chaque *BasicPhysicObject* met à jour son état grâce à la fonction *UpdateState* :

```

public void UpdateState(float timeStep)
{
    if (isStatic) { velocity = Vector3.zero; angularVelocity = 0; resultingForce = Vector3.zero; torque = 0; return; }
    else if (lockRoll) { torque = 0; angularVelocity = 0; }

    Vector3 acceleration = resultingForce / collider.GetMass(); //Détermine la valeur de l'accélération
    float angularAcceleration = torque / collider.GetInertia(); //Détermine la valeur du torque

    velocity += acceleration * timeStep; //Ajoute la variation de vitesse selon l'accélération et le timestep
    angularVelocity += angularAcceleration * timeStep; //Ajoute la variation de vitesse angulaire selon l'accélération angulaire et le timestep

    transform.position += velocity * timeStep; //Change la position selon le timestep et la vélocité
    transform.Rotate(Vector3.forward * angularVelocity * Mathf.Rad2Deg * timeStep); //Rotation de l'objet selon sa variation de position angulaire

    //Reset les forces pour le next updateCall
    resultingForce = Vector3.zero;
    torque = 0;
}

```

Une autre fonction importante est la fonction qui permet d'appliquer une force sur l'objet qui porte le nom de *ApplyForce*. L'application de la force va faire varier le torque dépendamment de l'endroit où elle est appliquée :

```

//Apply force at another point other than the center of mass where :
//force is the force applied
//r is the vector from the center of mass towards the position of where the force is applied
0 references
public void ApplyForce(Vector3 force, Vector3 r)
{
    resultingForce += force;

    torque += (r.x * force.y) - (r.y * force.x);
}

```

## II. Calcul du moment d'inertie

Les propriétés physiques des objets comme la masse et le moment d'inertie sont encapsulés dans un objet du nom de *MeshColliderScript* qui contient toutes les informations sur les meshes (objets/polygones) de notre simulation. Lorsque les objets sont créés, l'instance du *MeshColliderScript* va calculer son inertie et son centre de masse.

```

//Calculate the moment of inertia of the given polygon
float density = mass / -area;
inertia = 0;
for (int i = 0; i < modelPoints.Count; i++)
{
    int j = (i + 1) % modelPoints.Count;
    //Calculate the area of the current checking triangle to get his mass
    float massTriangle = 0.5f * density * Vector3.Cross(modelPoints[i], modelPoints[j]).magnitude;

    //Calculate the inertia of this specific triangle
    float inertiaTriangle = massTriangle * (modelPoints[i].sqrMagnitude + modelPoints[j].sqrMagnitude + Vector3.Dot(modelPoints[i], modelPoints[j])) / 6;

    //Adds the inertia value to the mesh
    inertia += inertiaTriangle;
}

```

### III. Calcul du centre de masse

```

//Calculate the center of mass of a given polygon
//Code adapted from the source document of Paul Bourke 1988
//http://paulbourke.net/geometry/polygonmesh/

//Get the area of a custom polygon
area = 0;
for (int i = 0; i < modelPoints.Count; i++)
{
    int j = (i + 1) % modelPoints.Count;
    area += (modelPoints[i].x * modelPoints[j].y) - (modelPoints[j].x * modelPoints[i].y);
}
area /= 2;

//Calculate the position of the center of mass
Vector3 centerOfMass = Vector3.zero;
for (int i = 0; i < modelPoints.Count; i++)
{
    int j = (i + 1) % modelPoints.Count;
    centerOfMass.x += (modelPoints[i].x + modelPoints[j].x) * ((modelPoints[i].x * modelPoints[j].y) - (modelPoints[j].x * modelPoints[i].y));
    centerOfMass.y += (modelPoints[i].y + modelPoints[j].y) * ((modelPoints[i].x * modelPoints[j].y) - (modelPoints[j].x * modelPoints[i].y));
}
centerOfMass /= 6 * area;

//Offset the position of the points to match the center of mass
for (int i = 0; i < modelPoints.Count; i++)
{
    modelPoints[i] -= centerOfMass;
}
transform.position += centerOfMass;

```

#### E. Énumérations des problèmes et des solutions

Le code effectué est très robuste et ne semble avoir aucun problème pertinent. Cependant, le seul hic est le fait qu'il est entièrement limité à des objets convexes. Effectivement, des objets non-convexes auraient rajouté beaucoup de complexité à notre simulation pour l'ensemble des parties scientifiques. Nous nous sommes donc limités à des objets simples. Malgré tout, si nous aurions voulu implémenter des objets non-convexes, il aurait fallu adapter l'ensemble les algorithmes de centre de masse et du calcul d'inertie.

## *Systèmes de friction*

### A. Mise en situation

Pour avoir une simulation physique le plus fidèle possible plusieurs éléments doivent être implémenter et un d'entre eux est la friction. Les objets dans le moteur physique doivent pouvoir interagir entre par la perte ou le transfert d'énergie créé par la collision et donc la friction de deux objets.

### B. Théorie

On peut séparer la friction en deux équation :

1. La première équation nous dit : Un objet immobile est plus difficile à bouger qu'un objet en mouvement puisque la force de friction statique doit suivre l'équation :

$$F_f = \mu_s * F_n$$

Le  $\mu_s$  de tout objet est plus grand que le  $\mu_c$  ce qui veut donc dire que la force de friction statique sera toujours plus grande que la force de friction cinétique qui est calculé par l'équation:

$$F_f = \mu_c * F_n$$

Dans cette équation la force normale reste la même.

2. La deuxième équation qui compose la friction est l'équation de la friction de l'air. Pour avoir le résultat le plus réaliste possible il faut respecter cette équation:

$$F_D = \frac{1}{2} * C * \rho * A * v^2$$

source utilisé pour la friction de l'air

<https://courses.lumenlearning.com/suny-physics/chapter/5-2-drag-forces/>

### C. Compréhension

La première équation qui est calculé deux fois est plutôt simple à comprendre puisque premièrement le coefficient  $\mu$  est une valeur donnée dépendamment du type de matériaux qui consiste l'objet. La force normale elle c'est la force qu'une surface qui est en contact avec notre objet exerce dessus. Pour la deuxième équation c'est un peu plus difficile puisque l'on veut que notre objet soit affecté par une friction qui serait créé par le déplacement des particules d'air. Cependant simuler des particules d'air serait extrêmement demandant pour l'ordinateur qui veut tester la simulation. Donc à la place on calcule une force qui essaye au mieux de reproduire ce que les particules d'air auraient créé comme résistance.

### D. Implémentation

Pour implémenter la force de friction il faut premièrement trouver la tangente de l'objet pour pouvoir appliquer la force dans le bon sens .

```
//Friction
Vector3 tangent = relativeVelocity - (Vector3.Dot(relativeVelocity, normal) * normal);
tangent = tangent.normalized;
```

après avoir le sens de notre force de friction on calcule la force de chaque impulsion nécessaire pour appliquer la force de friction à chaque instant de la simulation pour ce faire

il faut calculer  $j = \frac{-(1+e)((V^B - V^A) \cdot t)}{\frac{1}{mass^A} + \frac{1}{mass^B}}$

```
float momentOfInertiaObjectImpulseInhibitorFRICTION = Mathf.Pow(Vector3.Dot(perpVectorObject, tangent), 2) / objectInertia;
float momentOfInertiaOtherObjectImpulseInhibitorFRICTION = Mathf.Pow(Vector3.Dot(perpVectorOtherObject, tangent), 2) / otherObjectInertia;
float speedAlongTangent = Vector3.Dot(relativeVelocity, tangent);
float jt = -(0.2f)*Vector3.Dot(relativeVelocity, tangent);
jt = jt / (massImpulseInhibitor + momentOfInertiaObjectImpulseInhibitorFRICTION + momentOfInertiaOtherObjectImpulseInhibitorFRICTION);
```

Puis, on calcule la valeur du coefficient de friction de l'objet.

```
float mu = Mathf.Sqrt(friction1Static* friction1Static + friction2Static* friction2Static);
```

Par la suite, on vérifie si l'objet est en mouvement pour savoir si nous devons calculer la friction cinétique ou statique puis on calcule la force que doit avoir l'impulsion.

```
Vector3 frictionImpulse;
if (Mathf.Abs(jt) <= j * mu)
{
    frictionImpulse = jt * tangent * mu;
}
else
{
    float dynamicFriction = Mathf.Sqrt(friction1Dynamic * friction1Dynamic + friction2Dynamic * friction2Dynamic);
    frictionImpulse = -j * tangent * dynamicFriction;
}
```

Pour finir, on applique la force de friction.

```
//ApplyFriction
newVelocity += (1f / objectMass) * frictionImpulse;
newOtherVelocity -= (1f / otherObjectMass) * frictionImpulse;

newAngularVelocity += (Vector3.Dot(perpVectorObject, -frictionImpulse) / objectInertia);
newOtherAngularVelocity += (Vector3.Dot(perpVectorOtherObject, frictionImpulse) / otherObjectInertia);
```

Pour la force de friction de l'air on commence par trouver les points qui constituent les délimitations de l'objet.

```
Vector3[] p = new Vector3[4];
p[0] = new Vector3(bond.position.x, bond.position.y, 0.0f);
p[1] = new Vector3(bond.position.x + bond.width, bond.position.y, 0.0f);
p[2] = new Vector3(bond.position.x, bond.position.y + bond.height, 0.0f);
p[3] = new Vector3(bond.position.x + bond.width, bond.position.y + bond.height, 0.0f);
```

Par la suite, on trouve la normale de la vitesse de notre objet

```
Vector3 normalisedVel = velocity.normalized;
```

Pour trouver l'aire de notre surface en contact avec l'air il faut faire

```
float min1 = Mathf.Infinity;
float max1 = -Mathf.Infinity;
for (int i = 0; i < p.Length; i++)
{
    float temp = Vector3.Dot(normalisedVel, p[i]);
    min1 = Mathf.Min(temp, min1);
    max1 = Mathf.Max(temp, max1);
}

float A = Mathf.Abs(min1 - max1);
```

Ensuite, on calcule la norme du vecteur de la force de friction de l'air.

```
float AirForce;
float P = UniversalVariable.GetAirDensity();
float C = 0.4f;
float speed = velocity.magnitude;

AirForce = 0.5f * C * P * A * (speed * speed);
```

Pour finir, on calcule la force et on l'applique sur notre objet

```
drag = (AirForce * -normalisedVel);
ApplyForceAtCenterOfMass(drag);

physicObjects[i].ApplyAirDensity();
```

#### E. Énumérations des problèmes et des solutions

La force de friction entre les objets pouvait créer du mouvement ce qui dans la réalité n'est pas possible. Pour régler ce problème et optimiser la friction j'ai changé la façon dont la friction était calculée pour la faire en impulsion et implémenter une fonction qui vérifie que la force de friction ne créera pas de force par elle-même. Pour la friction de l'air le plus gros problème rencontré a été de savoir comment calculer l'aire de l'objet qui est en contact avec l'air de l'environnement. Comme solution j'ai utilisé la classe AABB qui crée un rectangle

tout autour de l'objet et qui s'adapte à l'orientation de l'objet ce qui veut donc dire que l'aire calculer ne sera pas parfait mais beaucoup plus simple.

F. Retour sur les objectifs personnels et professionnels

### *Détection de collisions*

A. Mise en situation

Parce que nous n'utilisons aucune fonctionnalité de base de Unity, nous devons développer un algorithme qui permet de détecter lorsque des formes quelconque entrent en collision.

B. Théorie

C. Compréhension

D. Implémentation

E. Énumérations des problèmes et des solutions

F. Retour sur les objectifs personnels et professionnels

### *Résolution de collisions*

A. Mise en situation

B. Théorie

C. Compréhension

D. Implémentation

E. Énumérations des problèmes et des solutions

F. Retour sur les objectifs personnels et professionnels

### *Système de joints et objets déformables*

A. Mise en situation

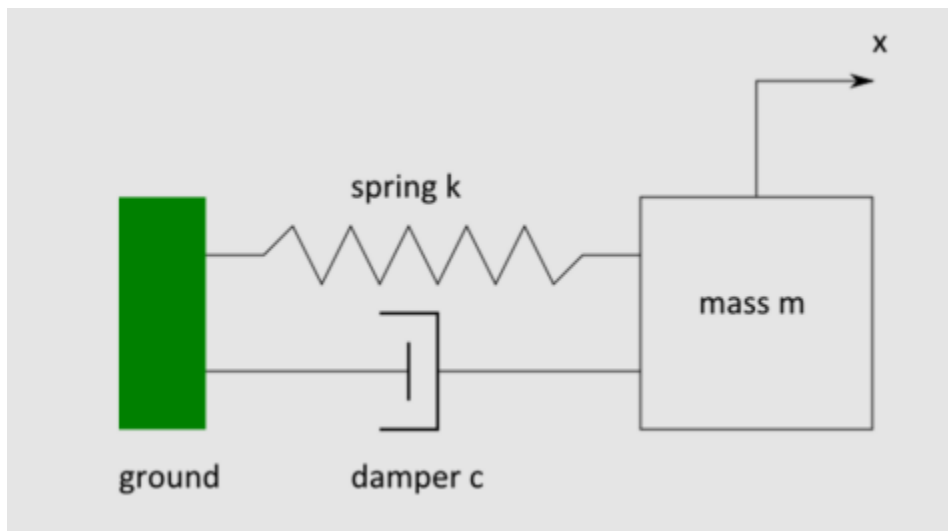
L'objectif de ma partie scientifique est d'ajouter des composantes dynamiques à notre sandbox pour former des objets plus complexes que des objets durs. Le principe sera d'ajouter un joint, ce qui est un objet qui ajoute un comportement choisi entre deux objets typiquement. Le but est d'ajouter une joint qui est stable, facilement ajustable et programmable par l'utilisateur. Dans notre cas, j'ai aussi créé un joint pour saisir des objets dans le sandbox et des exemples d'objets mous qui sont un arrangement d'objets liés par des joints.

## B. Théorie

i. Du point de vue théorique, il y a 3 éléments principaux dans mes implémentations. La première est la façon dont les contraintes sont manipulées dans l'implémentation des joints pour être malléable et stable et les deux autres sont les formules utilisées pour avoir un effet distinct pour chaque joints implémenter. Le joint de prise ne sera pas expliqué, car ces contraintes sont seulement de faire un inverse matricielle. Je vais plutôt prendre mon temps pour expliquer le joint de distance parce que c'est le même principe mais plus compliqué.

### 1 : Le principe de contraintes souples


De base, on peut imaginer des contraintes simplement en imaginant la contrainte  $k$  de ressort par exemple. Le problème avec cette intégration est simplement qu'elle n'est aucunement stable pour toutes masses et inerties dans chacun des contextes. Elle doit être adaptée pour chaque scénario à chaque seconde de la simulation ... Dans ce cas, j'ai trouver une méthode plus optimisé ici :





En posant les contraintes du ressort comme un oscillateur harmonique.  
Cette intégration à une équation différentielle connue.



$$m \frac{d^2 x}{dt^2} + c \frac{dx}{dt} + kx = 0$$

  
 acceleration

  
 velocity

  
 position

Et ensuite, on introduit le concept d'oscillateur dans l'amortisseur(c) et le ressort(k)

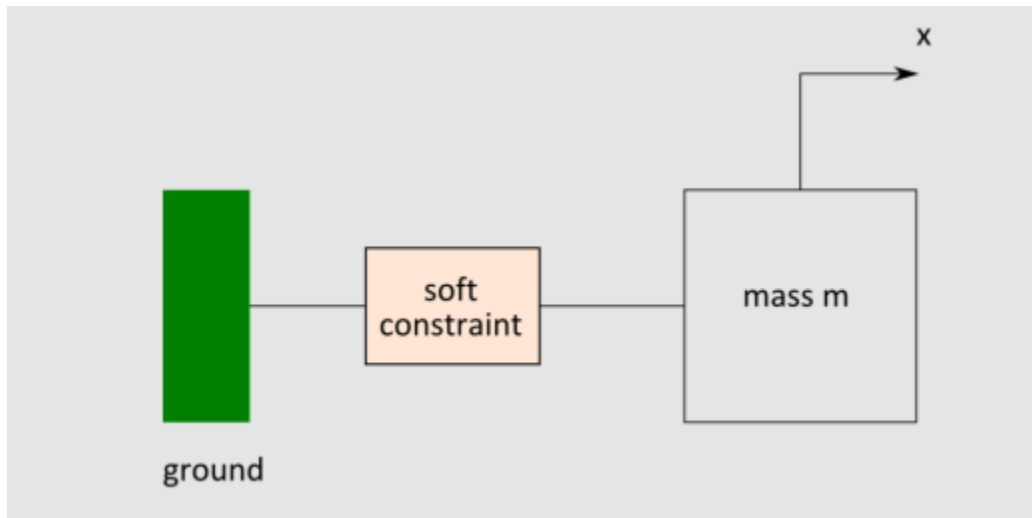
$$\frac{d^2 x}{dt^2} + 2\zeta\omega \frac{dx}{dt} + \omega^2 x = 0$$

$$2\zeta\omega = \frac{c}{m} \quad \omega^2 = \frac{k}{m}$$

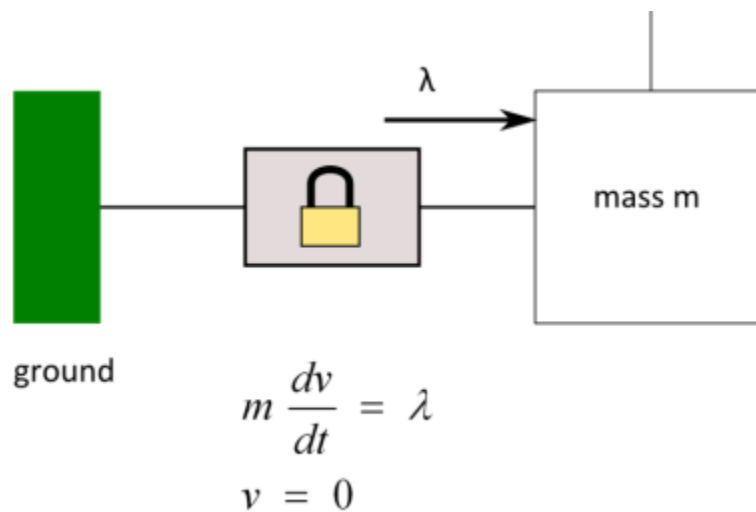
où  $\zeta$  = rapport d'amortissement  $\omega$  = fréquence angulaire

Ce qui est très bien ajustable pour l'utilisation. Maintenant vient le problème de résoudre cette équation intelligemment. On doit mettre de côté cette équation un peu et imaginer le

problème plus simplement. Posons cette boîte noir pour l'instant comme le fait ma source.



On peut alors commencer à définir cette boîte avec des contraintes dures simples.



Et on peut ajouter deux variables a cette équation a cause du 0 :

$$m \frac{dv}{dt} = \lambda$$

$$v + \frac{\beta}{h} x + \gamma \lambda = 0$$

On définit ici bêta et gamma pour rendre les contraintes souples. On peut ensuite déduire les équations.

$$v_2 = v_1 + \frac{h}{m} \lambda$$

$$x_2 = x_1 + h v_2$$

$$v_2 + \frac{\beta}{h} x_1 + \gamma \lambda = 0$$

On peut alors observer la similarité entre le système spring-amortisseur et comparer les types d'intégration de chaque.

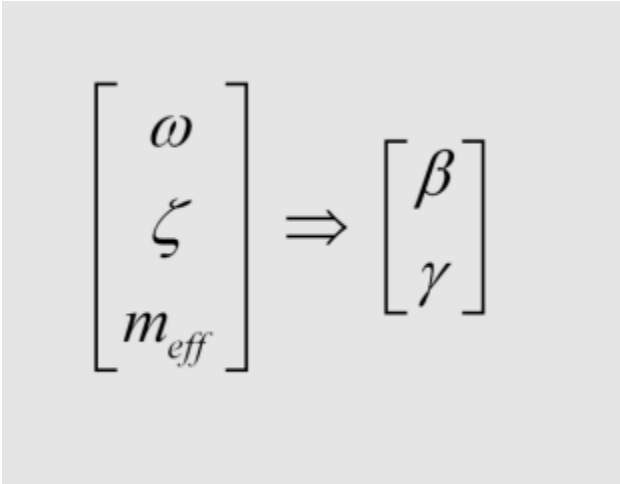
<div style="background-color: #f4a460; padding: 2px; display: inline-block;">Spring-Damper</div>	<div style="background-color: #f4a460; padding: 2px; display: inline-block;">Soft Constraint</div>
$v_2 = \frac{v_1 - \frac{hk}{m} x_1}{1 + \frac{hc}{m} + \frac{h^2 k}{m}}$	$v_2 = \frac{v_1 - \frac{\beta}{m\gamma} x_1}{1 + \frac{h}{m\gamma}}$
<div style="background-color: #90ee90; padding: 2px; display: inline-block;">Implicit Euler</div>	<div style="background-color: #90ee90; padding: 2px; display: inline-block;">Semi-implicit Euler</div>

En adaptant les deux solutions on a un mélange entre Euler implicite et Semi-implicite.

On a :

$$\gamma = \frac{1}{c + hk}$$
$$\beta = \frac{hk}{c + hk}$$

On peut alors sortir d'une dimension pour aller voir le système qui a été intégré dans le jeu. Parce que pour l'instant, nous avons réussi à introduire les variables bêta et gamma au système initial mais nous n'avons pas encore concilier la partie oscillateur harmonique. Cette partie est importante car elle est responsable pour le calcul de la masse. On peut facilement mettre en relation ces deux équations et la masse.


$$\begin{bmatrix} \omega \\ \zeta \\ m_{eff} \end{bmatrix} \Rightarrow \begin{bmatrix} \beta \\ \gamma \end{bmatrix}$$

On les met en relation comme on a fait précédemment en notant k et c avec le système harmonique.

$$k = m_{eff} \omega^2$$

$$c = 2m_{eff} \zeta \omega$$

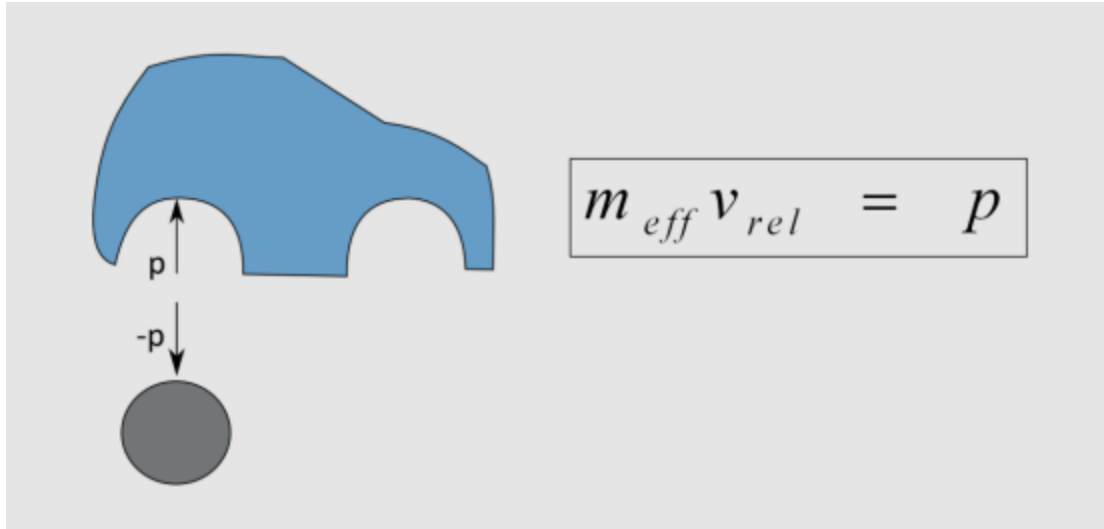
On a alors ,

$$\gamma = \frac{1}{m_{eff} \omega (\omega + 2h \zeta)}$$

$$\beta = \frac{h \omega}{2 m_{eff}} + 1$$

où h est le pas de temps entre chaque cadre de calcul.

On peut alors définir la masse effective pour avoir des contraintes encore plus stables. Elle permet de varier la tension dans les contraintes selon la vitesse relative entre deux objets.



Qui est calculer dans mon programme comme cela,

$$m_{eff} = \frac{1}{JM^{-1}J^T}$$

Avec les contraintes bêta et gamma, on obtient des paramètres souples et la masse effective rend l'intégration adaptative et beaucoup plus stable. Les calculs de ces éléments dépendent de l'intégration voulue, alors je vais illustrer et calculer ces éléments dans la deuxième partie.

## 2 : Le principe de joint de distance

Les joints de distance sont des joints qui tentent de garder tout le temps la même distance entre deux objets, tous les axes sont libres même la rotation. La résolution des contraintes se fait en 3 étapes. La première est de contraindre les équations de positions. On doit alors prendre cette équation et la faire dériver pour avoir les contraintes de vitesse. La troisième est d'isoler les vitesses. On a alors les outils pour calculer les contraintes de vitesse linéaire et angulaire. La dernière étape est de calculer le travail pour l'appliquer sur les vitesses initiales de chaque objet.

## 1. Contraindre les équations de positions

On commence avec C qui est la contrainte de distance :

$$C = d - l$$

Ici, d est la distance en temps réel et l la distance désirée en temps réel, on veut alors que l soit zéro pour avoir un équilibre. On doit alors définir d pour faire la dériver en respect du temps.

$$C = \|\vec{p}_a - \vec{p}_b\| - l$$

Cela est la norme de la distance entre les deux points ou p est le point ou le joint est attachée en additionnant les coordonnées globales du centre de masse et le décalage du centre de masse.

En utilisant la magnitude carrée de cette distance pour faire la racine carrée du produit scalaire. On substitue aussi la soustraction de vecteur par le vecteur u.

de sort que :

$$\vec{u} = \vec{p}_a - \vec{p}_b$$

Alors on obtient,

$$C = \sqrt{\vec{u} \cdot \vec{u}} - l$$

## 2. Les contraintes de vélocité

En prenant la dérivée de la contrainte C on obtient :

$$\frac{d(C)}{dt} = \vec{n} \cdot \frac{d(\vec{u})}{dt}$$

ou n est égale à :  $\frac{\vec{u}}{\sqrt{\vec{u} \cdot \vec{u}}}$  qui signifie la racine de la normale du vecteur u.

On peut alors ajouter l'élément de rotation(ra,rb) dans l'équation et leur masse(ma,mb).

$$\begin{aligned}\vec{p}_a &= \vec{c}_a m_a + \vec{r}_a R_a \\ \vec{p}_b &= \vec{c}_b m_b + \vec{r}_b R_b\end{aligned}$$

ou leur dérivées donne :

$$\begin{aligned}\frac{d(\vec{p}_a)}{dt} &= \vec{v}_a + w_a \times \vec{r}_a \\ \frac{d(\vec{p}_b)}{dt} &= \vec{v}_b + w_b \times \vec{r}_b\end{aligned}$$

Alors on peut introduire ces définitions dans l'équation dérivée de la contrainte

$$\frac{d(\vec{u})}{dt} = \frac{d(\vec{p}_a - \vec{p}_b)}{dt}$$

ce qui nous donne :

$$\frac{d(C)}{dt} = \vec{n} \cdot (\vec{v}_a + w_a \times \vec{r}_a - \vec{v}_b - w_b \times \vec{r}_b)$$

3. isoler les vitesses

On peut résoudre les produits vectoriels entre un vecteur de dimension 2 et un scalaire qui est

définie comme :  $w \times \vec{r} = \begin{bmatrix} -wr_y \\ wr_x \end{bmatrix}$  et on obtient,

$$\frac{d(C)}{dt} = \vec{n} \cdot (\vec{v}_a + R_{sa} w_a - \vec{v}_b - R_{sb} w_b)$$

On remarque ici que la partie droite est la vitesse relative entre les deux objets qui est en produit scalaire avec la racine de la magnitude du vecteur directionnel. On peut aussi sortir la jacobienne de cette équation de sorte que,



$$\frac{d(C)}{dt} = \begin{bmatrix} \dot{\vec{n}}^T & \dot{\vec{n}}^T R_{sa} & -\dot{\vec{n}}^T & -\dot{\vec{n}}^T R_{sb} \end{bmatrix} \begin{bmatrix} \vec{v}_a \\ w_a \\ \vec{v}_b \\ w_b \end{bmatrix}$$

ou,

$$J = \begin{bmatrix} \dot{\vec{n}}^T & \dot{\vec{n}}^T R_{sa} & -\dot{\vec{n}}^T & -\dot{\vec{n}}^T R_{sb} \end{bmatrix}$$

La jacobienne sera importante pour calculer la masse effective sur le joint.

#### 4. Compilation des outils finaux

On a besoin de calculer la masse effective qui nous permet de calculer  $\lambda$  qui est la grandeur de l'impulse. La masse effective est calculée dans mon programme comme cela,

$$m_{eff} = \frac{1}{J M^{-1} J^T}$$

Avec les contraintes bêta et gamma, on obtient des paramètres souples et la masse effective rend l'intégration adaptative et beaucoup plus stable. M dans ce contexte fait référence à la matrice diagonale des masses et des inerties des deux objets comme cela :

$$\begin{bmatrix} M_a^{-1} & 0 & 0 & 0 \\ 0 & I_a^{-1} & 0 & 0 \\ 0 & 0 & M_b^{-1} & 0 \\ 0 & 0 & 0 & I_b^{-1} \end{bmatrix}$$

Au final on a ,

$$J M^{-1} J^T = m_a^{-1} + m_b^{-1} + I_a^{-1} (\vec{n} \times \vec{r}_a)^2 + I_b^{-1} (\vec{n} \times \vec{r}_b)^2$$

qu'on inverse pour avoir la masse effective. Pour finaliser le calcul de lambda on doit calculer la variation des contraintes comme cela,

$$m_c \Delta \dot{C} = \lambda$$

ou la variation des contraintes s'exprime comme la vitesse relative fois la jacobienne en additionnant un biais qui permet de rendre les calculations plus stables :

$$\dot{C} = \mathbf{J}\mathbf{v} + b$$

ou le biais est une constante pour réduire des impulsions trop grandes qui se situe entre 1 et 0 pour contrôler la magnitude de l'impulse appliquer. On divise alors bêta par une petite variation de temps variation de temps pour l'obtenir.

$$\dot{C} + \frac{\beta}{h} C = 0$$

Au final on a :

$$\lambda = -m_c (\mathbf{J}\bar{\mathbf{v}} + b)$$

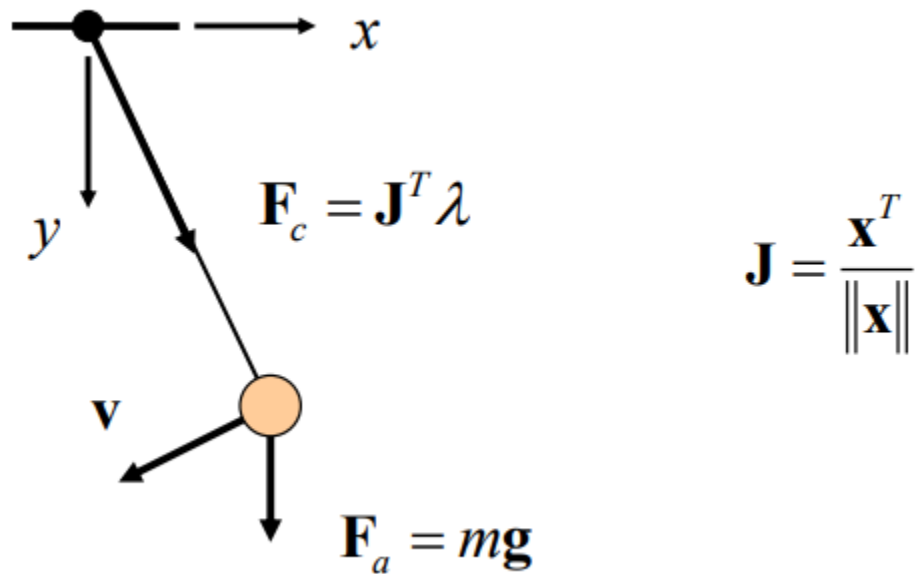
ou gamma est la variable qui influe la masse effective pour le système de contraintes souples et beta influence vitesse relatives.

### 5. Actualisation des vitesses

Maintenant qu'on a tous les outils nécessaires, il est temps d'introduire le concept de travail pour l'appliquer à nos vitesses dans le programme. J'utilise le travail virtuel qui devrait être nul car les deux objets sont contraints de façon équivalente dans système de joint. On obtient :

$$\mathbf{P}_c = \mathbf{J}^T \lambda$$

$$\mathbf{v}_2 = \bar{\mathbf{v}}_2 + \mathbf{M}^{-1} \mathbf{P}_c$$



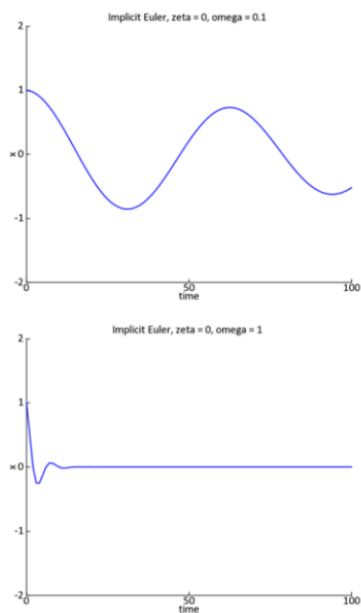
Et si  $v_2$  est positif alors l'application des forces à l'objet 1 sera négative de sorte à ce que les impulsions soient des opposés également contraints.

[https://box2d.org/files/ErinCatto\\_SoftConstraints\\_GDC2011.pdf](https://box2d.org/files/ErinCatto_SoftConstraints_GDC2011.pdf)  
[https://box2d.org/files/ErinCatto\\_ModelingAndSolvingConstraints\\_GDC2009.pdf](https://box2d.org/files/ErinCatto_ModelingAndSolvingConstraints_GDC2009.pdf)  
<https://dyn4j.org/2010/09/distance-constraint/>  
<https://www.tu-chemnitz.de/informatik/KI/edu/robotik/ws2017/vel.kin.pdf>  
<https://dyn4j.org/2010/07/equality-constraints/>

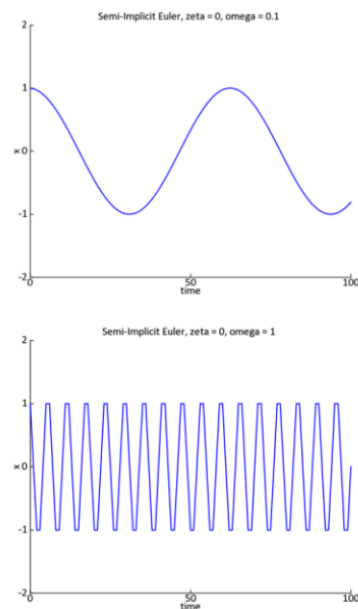
### C. Compréhension

Ma compréhension des contraintes molles est plus centrée sur l'optimisation des systèmes d'intégration d'Euler. D'un part le système amortisseur-ressort est très stable et simple à calculer mais a une complexité algorithmique plus haute. Tandis que le système oscillateur-fréquence est intégré par Euler semi-implicite qui tend à être vastement instable à haute fréquence mais est plus rapide en générale. Les contraintes molles viennent alors ajuster un bon milieu entre les deux dépendamment du contexte et des forces appliquer. Il rend la simulation dynamique et s'optimise selon le contexte de chaque image/calcul. Parce que l'intégration semi-implicite ne fonctionne que lorsque le temps entre chaque calcul est très grand, donc quand la fréquence de l'objet vient proche de celle de la simulation, les calculs deviennent vraiment instables.

Euler : Implicite



Semi implicite



On voit comment à haute fréquence, le semi implicite va créer des vagues de tremblement dans le joint. Les contraintes souples vont donc trouver le juste milieu entre ces deux objets

pour que les hautes fréquences soient plus stables et les basses aussi stables que le euler implicite.

Pour le joint de distance, les calculs ont l'air très compliqués. Mais au finale ma compréhension part du fait qu'on veut un système qui est contraint également entre deux objets qui les gardent à la même distance. En considérant que les joints sont placés dans le centre de masse, le concept n'est pas plus compliqué que ça. Pour la rotation quand le joint n'est pas placé dans le centre de l'objet, on peut imaginer que l'objet se balance sur le point de pivot et que lorsqu'il tente de monter en rotationnant sur un axe il crée de la force centripète. La partie la plus compliquée de cette intégration est de savoir quelles directions et forces ces objets doivent subir pour garder la même distance et avoir un effet rotationnelle sur l'axe alentour du point de rotation qui n'est pas le centre de masse. Et je comprends les démarches mathématiques mais je ne pourrais pas résumer simplement ces équations.

#### D. Implémentation

Pour beta et gamma, c'est assez simples on a besoin du temps , de la masse effective, de la fréquence et du ratio d'amortissement

```
private void ComputeBetaAndGamma(float timeStep)
{
    // If the frequency is less than or equal to zero, make this joint solid
    if (frequency <= 0.0f)
    {
        beta = 1.0f;
        gamma = 0.0f;
        jointMass = float.PositiveInfinity;
        frequency = 0.0f;
    }
    else
    {
        //  $\beta = hk / (c + hk)$ 
        //  $\gamma = 1 / (c + hk)$ 
        //  $k = m * \omega^2$ 
        //  $c = 2 * m * \zeta * \omega$ 
        // https://box2d.org/files/ErinCatto\_SoftConstraints\_GDC2011.pdf

        float omega = 2.0f * Mathf.PI * frequency;
        float k = 2 * jointMass * omega * omega; // Spring
        float h = timeStep;
        float c = 2.0f * jointMass * dampingRatio * omega; // Damping coefficient

        beta = h * k / (c + h * k);
        gamma = 1.0f / (c + h * k);
    }
}
```

Au début de mon algorithme , on initialise les éléments de la matrice M pour plus tard

```
// Get references to the BasicPhysicObject and MeshColliderScript components for both bodies
bpA = bo1.GetComponent<BasicPhysicObject>();
bpB = bo2.GetComponent<BasicPhysicObject>();
mcA = bo1.GetComponent<MeshColliderScript>();
mcB = bo2.GetComponent<MeshColliderScript>();
invMassA = 1.0f / mcA.GetMass();
invMassB = 1.0f / mcB.GetMass();
invInertiaA = 1.0f / mcA.GetInertia();
invInertiaB = 1.0f / mcB.GetInertia();
invInertiaSum = invInertiaA + invInertiaB;
invMassSum = invMassA + invMassB;
```

Ensuite on calcule a chaque image du programme nos distance entre le centre de masse et le point de pivot , la distance entre les deux objets comme cela :

```
Vector3 ra = (bodyA.rotation) * (offsetA);
Vector3 rb = (bodyB.rotation) * (offsetB);

// Compute the vector between the two anchor points
Vector3 pa = anchorA + ra;
Vector3 pb = anchorB + rb;
Vector3 d = (pb - pa);

// Compute the current length
float currentLength = d.magnitude;

// Compute the normalized direction vector between the two anchor points
Vector3 n = d.normalized;
```

Avec cette information on calcule la masse effective ,

```
// Compute the effective mass of the constraint
//  $M = (J \cdot M^{-1} \cdot J^T)^{-1}$ 
//  $J = [-n, -n \times r_a, n, n \times r_b]$ 
// (  $n = (anchorB - anchorA) / ||anchorB - anchorA||$  )
//  $r_a, r_b =$  vec from center of mass to offset
float crossA = Vector3.Cross(-n, ra).z;
float crossB = Vector3.Cross(n, rb).z;
float invEffectiveMass;
//first part is transitional aspect of the equation and the two second ones are relative to the rotation
invEffectiveMass = invMassSum + crossA * crossA * invInertiaA + crossB * crossB * invInertiaB ;
```

on peut ensuite calculer les valeurs de beta et gamma

```
// Compute beta and gamma values
ComputeBetaAndGamma(timeStep);
```

On peut donc calculer le biais avec beta pour le calcul de contraintes,

```
// Compute the bias term for the constraint
float bias = (currentLength - length) * beta / timeStep;
```

Maintenant est le temps de calculer votre lambda que mon code appelle impulse Mag , on doit alors calculer la vitesse relative nommée dv et jv qui sera additionnée.

```
// Compute the relative velocities
Vector3 v1 = bpA.getVelocity();
Vector3 v2 = bpB.getVelocity();
float w1 = bpA.getAngularVelocity();
float w2 = bpB.getAngularVelocity();
Vector3 raCross = new Vector3(-w1*ra.y, w1*ra.x, 0);
Vector3 rbCross = new Vector3(-w2*rb.y, w2*rb.x, 0);
//vitesse relative (dérivée de vitesse)
Vector3 dv = v2 + rbCross - v1 - raCross;
float jv = Vector3.Dot(dv, n);
// Compute Jacobian for the constraint (C = impulse Dir) here its defined by x^/sqrt(||x||)
// la racine est pour rendre les paramètres plus mous parce que ma simulation est trop rigide sans.
Vector3 J = d / d.sqrMagnitude ;
// Compute the corrective impulse
float impulseMag = -m *(jv + bias);

Vector3 impulse = impulseMag * J;
```

Pour calculer l'impulse il y a deux types d'impulse différent pour la vitesse linéaire et angulaire. Dans mon code , l'impulse angulaire est calculée dans l'équation qui additionne.

```
Vector3 impulse = impulseMag * J;
//Apply corrective impulse
if (!bpA.getIsStatic())
{
    v1 -= impulse * invMassA ;
    w1 -= Vector3.Dot(new Vector3(-ra.y*impulseMag, ra.x*impulseMag), J) * invInertiaA;
    bpA.SetVelocity(v1, w1, timeStep);
}
if (!bpB.getIsStatic())
{
    v2 += impulse * invMassB ;
    w2 += Vector3.Dot(new Vector3(-rb.y * impulseMag, rb.x * impulseMag), J) * invInertiaB;
    bpB.SetVelocity(v2, w2, timeStep);
}
```

Et fini !

- E. Énumérations des problèmes et des solutions
- F. Retour sur les objectifs personnels et professionnels

### *Simulation de fluide*

- A. Mise en situation
- B. Théorie
- C. Compréhension
- D. Implémentation



- E. Énumérations des problèmes et des solutions
- F. Retour sur les objectifs personnels et professionnels

## **Annexe**