

# Rapport TP Animation par modèle physique



Émilie Vernay

git : <https://github.com/Emilie-V-pro/TTengine-2.0>  
(le code spécifique au tp se trouve dans src/sceneV2,  
le code de la physique se trouve dans src/sceneV2/  
animatic/simulation)

## Introduction

Ce TP d'animation par modèle physique consiste à simuler la déformation d'un tissu en fonction de force externe qui lui sont appliqués et des collisions avec une scène.

## I - Boucle de simulation pour l'animation d'un objet

### Les particules

Simuler chaque atome d'un objet pour des calculs de physique demanderait un temps et une mémoire quasiment infinie. Une solution pour effectuer ces calculs consiste à simplifier la représentation des objets en le discrétilisant en un nombre fini de particules. Chaque particule possède une position  $p$  et une masse  $m$ . Nous pouvons aussi leur associer une force  $F$ , une accélération  $a$  et une vitesse  $v$ .

### Calcul des forces exercées sur la particule au temps $t_0$

La première étape de la boucle consiste à faire une somme des différentes forces à chaque particule. Cela peut être la gravité, le vent, la réaction d'un support, le frottement entre la particule et le support.

### Calcul de l'accélération au temps $t_0$

Selon les lois de Newton

- Soit un objet de masse constante  $m$ , accélération  $a$ , force  $F$ .
- Équation du mouvement :  $F = ma$

nous pouvons donc déduire l'accélération au temps  $t_0$  correspond à  $a = \frac{F}{m}$

### Intégration de l'accélération et de la vitesse

Nous pouvons ensuite obtenir la vitesse en intégrant l'accélération, puis la position en intégrant la vitesse. Une des façons les plus simples de la calculer est d'utiliser l'intégration semi-implicite d'Euler. De plus, cette méthode permet d'obtenir des résultats assez stables.

$$\begin{cases} x'(t + dt) = x'(t) + dt x''(t) \\ x(t + dt) = x(t) + dt x'(t + dt) \end{cases}$$

Cette méthode consiste à calculer la vitesse au temps  $t + dt$  à partir de la vitesse actuel, de l'accélération et du pas de temps. Nous pouvons faire la même chose avec la position à partir de la position précédente, de la nouvelle vitesse et donc obtenir la nouvelle position.

## Transfert pour l'affichage

Il ne reste plus qu'à transférer les nouvelles positions au vertex buffer sur le GPU. Si la simulation n'est pas synchronisée avec l'affichage. Certaines optimisations sont possibles pour ne pas avoir à afficher les nouvelles données

## II - Système masse ressorts

Un système masse ressort est en ensemble de particule reliée entre elle par des ressorts. Ces systèmes peuvent former des courbes, des surfaces et des volumes en fonction de la topologie des liaisons. Dans notre cas, un système en 2D permet de simuler simplement un tissus.

Les ressorts appliquent des forces de tension et de compressions aux particules selon cette équation :

$$F = \sum_{j \in A} (\text{élasticité} + \text{viscosité}) \vec{u}_{ij}$$

$$\text{élasticité} = -k_{ij} (\|x_i - x_j\| - I_{ij})$$

$$\text{viscosité} = \nu_{ij} \left( (x')_i - (x')_j \right) \cdot \vec{u}_{ij}$$

avec :

- $A$  l'ensemble des particule relié à i
- $k_{ij}$  la raideur du ressort (la vitesse à la quelle le ressort retrouve sa longueur original)
- $I_{ij}$  la longueur du ressort au repos
- $\nu_{ij}$  la viscosité du ressort
- $\vec{u}_{ij}$  le vecteur du ressort

Nous avons juste à prendre ces forces en compte pour que ce système fonctionne avec la boucle de simulation. La topologie peut être modifiée pour représenter certain phénomène comme la déchirure.

## III - Fonctionnalité de la simulation

### Tissu fonctionnel



Figure 1: Tissu avec une texture de drap de la Sponza soumis à la gravité et la force du vent

## Interaction avec le tissu



Figure 2: Plusieurs noeuds du tissu sont attachés à des objets et sont déplacés via le clavier

## Collisions

sphère



Figure 3: Tissu en collision avec une sphère

cube

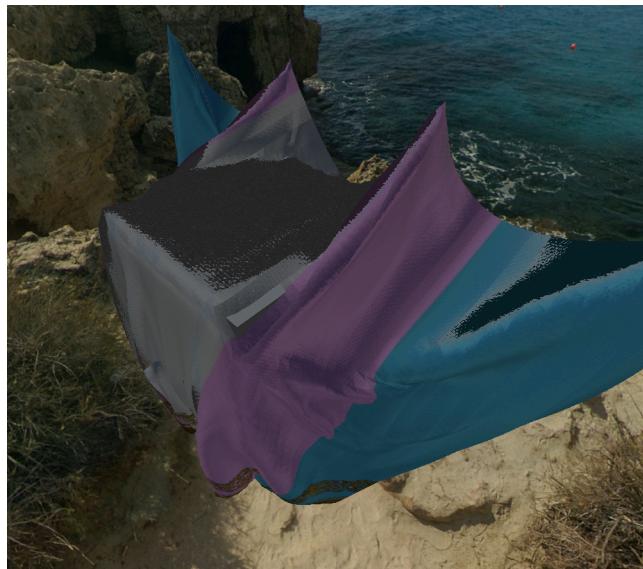


Figure 4: Tissu en collision avec un cube

## IV - Amélioration personnelle

Pour ce projet, je n'ai pas utilisé la librairie gkit mais mon propre moteur de rendu en C++ avec Vulkan et GLM. Cela m'a permis d'avoir moi-même le contrôle sur le fonctionnement des threads, l'architecture de la scène, la gestion de la mémoire du gpu...

### Modification du shader de rendu du tissu

La première amélioration que j'ai apportée au rendu tissu a été la désactivation du backface culling pour que les 2 face du tissu soient affichées. Ainsi que l'inversion de la normale si les triangles du tissu sont regardés par derrière. L'inversion de la normale peut se faire avec le code ci-dessous :

```
1 if (!gl_FrontFacing) {  
2     surfaceNormal = -surfaceNormal;  
3 }
```

glsl

La seconde amélioration est le remplacement du modèle de lumière de blinn-phong par du physical based rendering. Je me suis basé sur l'implémentation de [learnOpenGL](#). En plus de cela j'ai ajouter une version basique de l'image based lighting en utilisant la skybox comme source de lumière. Pour mettre en avant cela, j'ai repris les textures de normalmap, metal-roughness et albedo (recoloré aux couleurs trans) de la Sponza et je les ai appliqués au tissu.

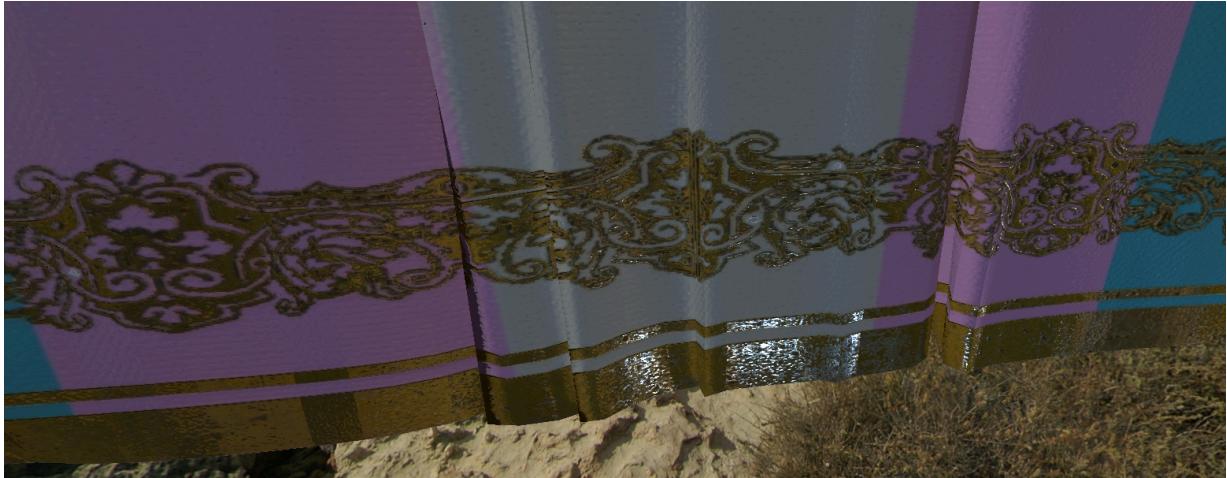


Figure 5: rendu des du tissu de la Sponza utilisant un rendu physiquement basé

### Utilisation de Vulkan pour séparer la boucle de simulation de l'affichage

Un des principaux avantages de Vulkan sur OpenGL est la possibilité de faire des appels à l'api sur plusieurs threads en parallèle et l'utilisation des queues de transfert du GPU. Même si des techniques de streaming de ressource existe sur OpenGL, celles-ci ne sont pas simples à mettre en place. Avec cela, nous pouvons simplement déléguer la partie simulation et mise à jour du vertex buffer à un thread différent du rendu (Il est aussi possible de soumettre au GPU des dispatch de compute shader si nous voulons mettre les calculs de physique sur le GPU depuis cet autre thread).

Cette séparation a permis de considérablement accéléré la boucle de simulation permettant d'avoir une physique en temps réel (sans être au ralenti). En effet, la simulation n'a pas à attendre que la boucle face des appels aux fonctions OpenGL/Vulkan qui peuvent être très lourd.

### Interaction avec un squelette

Du fait qu'une grande partie des TP d'image ont été réalisés sur le même moteur, cela m'a permis de fusionner ce TP avec celui de M Meyer. J'ai donc rajouté une cape au squelette du TP d'animation de personnage. Pour cela, j'ai d'abord attaché des noeuds du tissu aux épaules du squelette pour que la cape bouge avec le personnage. Puis j'ai un des collisions au squelette pour que le tissu interagisse correctement. Affin de minimiser les calculs de collision, j'ai décidé d'utiliser des capsules pour représenter les différents membres et j'ai gardé que les membres pour être en collision avec le tissu.



Figure 6: Collision avec un squelette



Figure 7: hit box du squelette

## Parallélisation des calculs sur CPU

Pour améliorer les performances de la simulation. J'ai décidé de paralléliser les calculs sur tous les cœurs du CPU. Pour cela, j'ai utilisé la librairie OpenMP pour gérer la création et la répartition des threads. J'ai juste eu à utiliser une instruction préprocesseur avant les boucles for comme le code ci-dessous :

```

1 #pragma omp parallel for
2 for (auto &ressort : _SystemeMasseRessort->GetRessortList()) {
3     ...
4 }
```

cpp

## Suggestion d'améliorations

D'autre amélioration du TP aurait pu être fait, comme :

- La parallélisation des calculs sur le GPU via des compute shader. Cela permet de réduire les temps de calculs, car ceux-ci sont faits en parallèle sur le GPU et ne requiert pas de transfert du vertex buffer entre la RAM et la VRAM. Il faut cependant attention à l'implémentation des algorithmes de calculs de force des ressorts qui peuvent nécessiter une synchronisation des instances du compute shader.
- L'utilisation de BVH pour calculer les collisions. Le système de collision naïve montre déjà ses faiblesses avec seulement une vingtaine de collisionneurs. Le BVH pourrait faciliter cela et diminuant grandement le nombre de tests avec des volumes englobants.