UNIVERSITÉ DE SHERBROOKE

**Faculté des sciences**
**Département d'informatique**
http://www.usherbrooke.ca/informatique

# IMN504 – animation et rendu temps réel - TP 1

Guillaume GILET - guillaume.gilet@usherbrooke.ca

Arthur DELON – arthur.delon@usherbrooke.ca

## 1. Consignes

- Travail à faire par groupe seul.
- Le travail sera à rendre avant le 16 février 2026
- Livrables : Une archive de la solution **nettoyée** (sans les fichiers de compilation intermédiaires). A rendre sur turnin.dinf.usherbrooke.ca

## 2. Objectif

Pour ce TP, nous disposons d'un squelette d'application volontairement très simple pour automatiser certaines tâches. L'objectif est d'animer un drapeau animé par système de masse-ressorts.

## 3. Travail demandé

Pour le rendu final, ce projet devra contenir les éléments suivants :

- Un drapeau animé par système masse-ressort. Ce drapeau sera défini par une grille de points (les masses) « attachées » virtuellement par un ensemble de ressorts. Le drapeau devra être soumis aux forces de gravité et du vent, et être attaché à un cylindre qui sera manipulé (le repère du drapeau est ici attaché au repère du cylindre).
- Bonus : N'hésitez pas à rajouter des effets supplémentaires (amortissement, collisions…)

Le code devra être rendu sous la forme d'une archive du projet (pensez à nettoyer le projet avant le rendu afin de limiter la taille de l'archive). Une attention particulière sera portée au code produit et a ses qualités (commentaires, optimisation raisonnable, pas de code inutile, bonne séparation initialisation et boucle de rendu...). Des points bonus seront ajoutés pour les effets supplémentaires (collisions…)

## 4. Code et application

La scène fournie contient un sol, un objet drapeau **Drape** et un objet cylindrique **Pole.** Le repère de l'objet drapeau étant attaché à l'objet cylindrique, c'est ce dernier qu'il faudra manipuler pour tester les différentes conditions de notre simulation.

### Définir les ressorts

Dans le code fourni, la définition du modèle masse-ressort du drapeau sera fait dans la classe **CustomModelGL** Celle-ci génère déjà la géométrie du drapeau (une grille de points). L'idée est de compléter le constructeur fourni pour remplir un tableau de ressorts (*springs*). Un ressort (*spring*) est une structure qui contient :

- Les identifiants des sommets aux extrémités.
- Une longueur au repos

- Un multiplicateur des coefficients de rigidité. Pour plus de simplicité et de contrôle, une rigidité globale sera définie via l'interface. Cette dernière sera multipliée par le multiplicateur lors de l'évaluation des forces

Il va falloir dans cette classe définir des ressorts pour les voisins directs (orthogonaux) de chaque point de la grille, ainsi que pour des voisins en diagonale. Pour augmenter la stabilité de la simulation, expérimentez également l'ajout de ressorts a de plus longue distance (par exemple, un ressort de plus faible rigidité entre des éléments situés à 2 « cases » de distance.

## Mettre à jour la simulation

La simulation en elle-même se fait dans le matériau **MassSpringMaterial**, associé à l'objet **Drape.** Les paramètres de la simulation seront accessibles dans l'interface du matériau au cours de l'exécution du code **(Onglet *SceneInformation – Nodes - Drape – BaseDrape*).** Ces paramètres physiques vont contrôler la simulation. Ils sont rangés dans une structure contenant:

- *mass* : la masse associée à chaque sommet
- *deltaTime* : le pas de temps de notre simulation
- ks*_Stiffness* : la rigidité globale des ressorts de la simulation
- kd*_dampening* : un facteur global d'amortissement (réduit les forces de tous les sommets en fonction de la vitesse)
- *wind* : la force du vent
- *windFriction* : La friction causée par l'interaction du tissu avec l'air

Dans la classe **MassSpringMaterial,** la fonction *animate*() s'effectue à chaque pas de temps, appellera une fonction *computeMassSpringAnimation*(), qui calculera les forces, vitesses et déplacements des sommets, avant de mettre à jour les positions des sommets sur le GPU

La première étape va être de mettre à jour la simulation en fonction du pas de temps et du schéma d'intégration semi implicite d'Euler, dans la fonction *updateSimulation().* Cela consiste pour chaque élément, à mettre à jour sa vitesse, puis sa position (cf. cours)

## Ajouter les forces

Afin d'ajouter des forces, il faudra compléter la fonction *computeMassSpringAnimation*() pour obtenir une simulation satisfaisante. Il faudra dans cette fonction ajouter les forces s'appliquant aux éléments :

- La gravité : $\vec{g} * m$
- La résistance de l'air (nous prendrons ici une approximation classique) : $kd_{dampening} * m * \vec{V}$
- La force du vent : $m * (\vec{V} - \overrightarrow{wind})$
- La force de friction (approximant la prise au vent du tissu) :

$$-windFriction * \left(\vec{N} * \left(\overrightarrow{wind} - \vec{V}\right)\right) * \vec{N}$$

- La force des ressorts (cf. cours)

Il est à noter que les formules proposées ici sont des grossières approximations des phénomènes réels. Nous vous encourageons à chercher, expérimenter et développer (en justifiant) des approximations plus poussées et réalistes, tout en gardant un temps de calcul raisonnable.

Finalement, un exemple de contraintes externes est donné pour « attacher » le drapeau au cylindre : Les forces des points accrochés au cylindre ont manuellement réduites à 0.

# 5. Rappel : Architecture du moteur

Prenez un peu de temps pour vous familiariser avec l'architecture du code fourni. Cette application est composée de plusieurs classes qui interagissent entre-elles pour automatiser le chargement des objets et l'interface utilisateur.

## Architecture Générale

Dans notre scène 3D, chaque objet 3D sera exprimé par un **nœud** : une instance de la classe ***Node***, qui est un objet c++ contenant un modèle 3D (classe ***ModelGL***), un repère 3D (classe *Frame*) et un matériau contenant une fonction de rendu et les shaders permettant l'affichage (classe ***MaterialGL***). Chaque objet nœud, matériau, et modèle sera identifié par un identifiant unique (une chaine de caractères)

Élément central de l'application, la classe ***Scene*** est un singleton contenant des pointeurs sur les éléments de la scène (Nœuds, modèles 3D, caméra...). Le rendu de la scène est effectué par la classe ***EngineGL***. La classe ***Application*** est responsable du chargement de la fenêtre graphique, des extensions OpenGL ainsi que de la gestion des actions de l'utilisateur.

## Classe Application

Cette classe contient principalement une fonction ***mainLoop*** implémentant la boucle principale. Son role est de traiter les interactions utilisateurs et d'appeler la fonction de rendu de l'objet ***EngineGL***. Elle contient également plusieurs fonctions *callback* appelée par la librairie **GLFW** lorsque l'utilisateur interagit avec l'application. C'est dans cette classe qu'il faudra traiter les interactions souris/clavier de l'utilisateur.

## Classe Scene

Cette classe est un singleton. C'est un objet unique dont on peut récupérer son unique instance depuis toute fonction du code en utilisant la fonction **Scene : :getInstance().** Elle contient un nœud ***Root***, qui sera le repère de référence de la scène 3D, auquel sera attaché une caméra (classe ***Caméra***), ainsi qu'un nœud **SceneNode**, auquel seront attachés tous les autres nœuds de la scène 3D.La gestion des nœuds s'effectue via un *manager de ressource :* La fonction ***getNode("monNode")*** renvoie un pointeur vers le nœud nommé *"monNode*" (Le nœud sera créé si il n'existe pas encore). Cela permet de retrouver tout nœud de la scène à tout moment dans le code. De la même manière, chaque modèle 3D sera créé par la fonction ***getModel("filename")*** avec *filename*, le chemin vers un fichier contenant un modèle 3D

## Classe EngineGL

C'est dans cette classe que se passent la plupart des choses en OpenGL. Elle contient une fonction ***init*** responsable de la création de la scène 3D (créer les nœuds, les modèles, les matériaux…). Ses deux autres fonctions majeures sont la fonction ***animate***, qui appelle la fonction ***animate*** de chaque nœud (afin de faire évoluer les paramètres des objets et gérer les déplacements, animations…) et la fonction ***render*** qui contient la boucle de rendu OpenGL. Cette dernière fonction nettoie la fenêtre graphique et appelle la fonction de rendu de chaque nœud, qui sera en charge de l'affichage de son modèle en fonction de son matériau.

## Classe Node

Cette classe est principalement un conteneur modélisant un objet 3D. Chaque objet est composé d'un repère (classe ***Frame***), éventuellement d'un modèle 3D (classe ***ModelGL***) et d'un matériau (classe ***MaterialGL***). Chaque nœud se place ainsi dans une hiérarchie et peut "*adopter*" d'autres nœuds. Les repères 3D de chaque nœud sont ainsi liés dans une hiérarchie de transformations (*cf cours*). Cette classe contient une fonction ***animate*** pour gérer les animations ainsi qu'une fonction ***render*** , dédiée à l'affichage. Cette fonction appelle la fonction ***render*** du matériau associé. L'ensemble des traitements et commandes pour afficher le modèle d'un nœud sera effectuée dans la fonction ***render*** du matériau.

## Classe Frame

Cette classe modélise un repère 3D (une matrice de transformation). Chaque repère est "*lié*" à un repère parent menant jusqu'au nœud **Root** de la scène. Cette classe contient toutes les fonctions de changement de repère pour exprimer un point ou une direction d'un repère dans un autre (tous les repères doivent être attachés au repère **Root**). Une fonction ***getModelMatrix***() permet de récupérer la matrice contenant l'ensemble des transformation depuis ce repère vers le repère **Root.**

## Classe ModelGL

Cette classe modélise un modèle 3D chargé depuis un fichier. Elle contient les différents tableaux OpenGL (VBO,VA…) et est responsable de les initialiser. Une fonction ***drawGeometry()*** permet d'envoyer la géométrie dans le pipeline OpenGL.

## Classe MaterialGL

Cette classe modélise la méthode de rendu de chaque objet. Elle contient un objet ***GLProgramPipeline*** encapsulant l'objet OpenGL correspondant (le pipeline graphique). Chaque nouvelle méthode de rendu devra être une nouvelle classe héritant de celle-ci. Le **constructeur** de cette classe devra spécifier les shaders à utiliser (classe ***GLProgram)***, les lier au pipeline, valider ce dernier et gérer ses propres variables **uniform**. La fonction ***render*** devra être responsable du rendu de son modèle 3D associé. Finalement, la fonction ***animate*** sera appelée régulièrement et sera responsable des mises a jour des variables **uniform** dont la valeur change à chaque frame. (Note : ces deux fonctions reçoivent en paramètre **o**: un pointeur vers le nœud sur lequel le matériau sera appliqué). La classe **BaseMaterial** est un exemple de matériau utilisé pour ce TP. N'hésitez pas à créer vos propres matériaux (un nouveau fichier classe, héritant de **MaterialGL**) dans son propre répertoire) et à rajouter des fonctions (par exemple pour modifier les variables **uniform** depuis le reste du code).

Attention: chaque matériau contient son propre objet OpenGL ***Pipeline***, ses propres *shaders* et son propre espace mémoire GPU. Il faudra donc créer une nouvelle instance d'un matériau si plusieurs objets se partagent le même matériau.

## Classe GLProgramPipeline et GLProgram

Ces classes encapsulent les objets OpenGL ***ProgramPipeline*** et ***Program***. Ces classes s'occupent de charger et de compiler les *shaders* passés en paramètre. Elles contiennent notamment les fonctions permettant d'afficher l'état de l'objet (erreurs de compilation…) ainsi qu'une fonction *getId()* renvoyant l'identifiant OpenGL correspondant.

## Classe GLProgramPipeline et GLProgram

Cette classe modélise une caméra et doit être liée au nœud *Root*. (Cela est fait lors de son initialisation) Elle contient un repère (une ***Frame***) ainsi que les fonctions permettant de spécifier les différentes projections utilisées. Vous utiliserez notamment sa fonction *getViewMatrix()* et *getProjectionMatrix()* pour récupérer ses matrices de vue et de projection.