Département d'Informatique et de Recherche Opérationnelle
Université de Montréal

Pierre Poulin
Bureau : 2389
poulin@iro.umontreal.ca

IFT3355 : Infographie
Travail Pratique #1 : 15%

Disponible :	Jeudi 20 janvier	13 :30
Remise :	Jeudi 3 février	13 :30 au début de la démo
Equipes :	Deux étudiant(e)s	

1 Description

Pour ce premier travail pratique, on vous demande d'implémenter un petit robot dont la tâche est de traverser un labyrinthe afin de toucher une cible. Ce robot peut articuler son bras droit, lui permettant d'atteindre son objectif. Une fois la cible atteinte, elle changera de couleur comme signe de réussite. Le but de ce travail est de vous familiariser avec les diverses transformations matricielles qui sont fréquemment utilisées en infographie.

2 Détails

2.1 Matrices de transformation

Vous avez à implémenter les matrices de transformation qui calculent la rotation, la translation et le changement d'échelle des objets dans l'espace. Pour ce faire, vous devrez implanter les quatres méthodes suivantes

1. *GetMatrix* de la classe *Transformation*, qui crée la matrice de transformation locale à partir des paramètres de rotation, de translation et de changement d'échelle,
2. *GetInverseMatrix* de la classe *Transformation*, qui crée la matrice inverse de transformation locale à partir des paramètres de rotation, de translation et de changement d'échelle,
3. *GetLocalToGlobal* de la classe *Node*, qui combine les matrices de transformation locales, permettant le passage de coordonnées locales en coordonnées globales,
4. *GetGlobalToLocal* de la classe *Node*, qui combine les matrices inverses de transformation locales, permettant le passage de coordonnées globales en coordonnées locales.

Les méthodes *GetMatrix* et *GetInverseMatrix* de la classe *Transformation* nécessitent la complétion de sous-fonctions. Elles sont clairement indiquées dans le code.

2.2 Assemblage

Les objets seront construits à partir de primitives que vous aurez à assembler dans votre programme afin de constituer une scène similaire à celle du prototype. Chaque primitive est de taille déterminée par les paramètres donnés lors de sa construction. La boîte et le cylindre sont centrés en x et y et vont de 0 à S_z en z alors que la sphère est centrée à l'origine (voir figure 1).

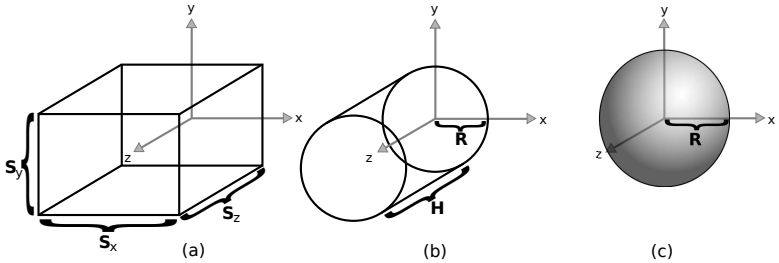

FIGURE 1 – Primitives de base dans leur repère initial (a) Boîte (b) Cylindre (c) Sphère.

2.2.1 Robot

Le robot est constitué de douze parties distinctes dont les dimensions sont les suivantes

1. Un cylindre pour le torse de rayon 0,5 et de hauteur 2,
2. Une sphère pour la tête de rayon 0.5,
3. Un cylindre pour chacun des deux bras de rayon 0,2 et de hauteur 1,
4. Un cylindre pour chacun des deux avant-bras de rayon 0,15 et de hauteur 1,
5. Une boîte pour chacune des mains d'épaisseur 0,1, de largeur 0,2 et de longueur 0,3,
6. Un cylindre pour chacune des jambes de rayon 0,2 et de hauteur 2,
7. Une boîte pour chacun des pieds de largeur 0,3, de longueur 0,7 et de hauteur 0,2.

La hiérarchie du robot est relativement simple. Celle-ci commence par le torse qui constitue sa base. À ce dernier sont attachés la tête, les bras ainsi que les jambes. Les avant-bras sont attachés aux bras, puis les mains aux avant-bras. On rattache finalement les pieds aux jambes.

Les détails de construction du robot se trouvent dans la méthode *Build* de la classe *Robot*.

2.2.2 Labyrinthe

Le labyrinthe est constitué d'un plancher et de murs. Pour cette partie, vous aurez seulement à créer et à positionner chaque mur. Pour ce faire vous devrez compléter la méthode *SegmentToBoxNode* de la classe *Labyrinth*, qui reçoit en entrée un segment de droite et qui donne en sortie une boîte dont l'orientation et les dimensions correspondent au segment donné en entrée. Chaque segment sera situé dans le plan XZ et aura donc une coordonnée y nulle.

Chaque mur sera d'épaisseur 0,05, de hauteur 1 et de longueur égale à la longueur du segment. Vous devrez ensuite modifier les paramètres de transformation locale pour aligner la boîte sur le segment donné, ce qui terminera la construction du mur. Vous devrez utiliser quelques règles trigonométriques pour cette partie.

Les détails de construction des murs se trouvent dans la méthode *SegmentToBoxNode* de la classe *Labyrinth*.

2.3 Contrôle du robot

Pour contrôler le robot, utilisez les touches `w` et `s` pour avancer et reculer et les touches `a` et `d` pour tourner à gauche et à droite respectivement. Cette animation devra être faite dans la méthode *Animate* de la classe *Robot*.

La classe *Robot* garde en mémoire l'orientation locale du robot qui sera utilisée pour le calcul de la translation lorsqu'il avance ou recule. On garde aussi en mémoire l'orientation de la jambe par rapport au torse qui sera utilisée pour l'animation des jambes.

Afin de garder un maximum d'uniformité, la méthode *Animate* vous donne le temps en secondes qui s'est écoulé depuis le dernier appel à cette fonction. Ce paramètre vous permettra de savoir exactement de combien d'unités vous devrez ajuster la position et l'orientation pour obtenir une animation fluide et précise.

L'orientation des jambes pourrait suivre une fonction sinusoïdale pour un minimum de réalisme. Je vous laisse par contre la liberté de créer vos propres fonctions. La translation effectuée à chaque pas de temps devra être telle que le

robot n'ait pas l'air de glisser sur le sol, ce qui est appelé le « footskating ». La translation ne sera donc pas constante et devra être calculée en fonction de la vitesse à laquelle les jambes bougent. Un peu de calcul différentiel pourrait être utile pour cette partie parce que le tâtonnement peut parfois être long et donne souvent des résultats inexacts.

Lorsque l'angle d'ouverture des jambes du robot n'est pas nul, les pieds auront tendance à flotter dans le vide. Il faudra donc appliquer une correction à la hauteur du robot afin que les pieds touchent le sol en tout temps. Une façon simple de s'en assurer est d'appliquer une translation verticale correspondant à la moyenne des hauteurs de la base de chaque pied, en considérant que la hauteur du sol est à $y = 0$. L'extrême précision n'est pas demandée, il s'agit simplement de réduire l'impression que le robot vole.

Les détails d'animation du robot se trouvent dans la méthode *Animate* de la classe *Robot*.

2.4 Contrôle du bras

Les articulations contrôlables du bras droit sont l'épaule et le coude, ce qui s'implémentera par des transformations appliquées au bras et à l'avant-bras respectivement. Utilisez les touches `r` et `f` pour contrôler le bras et les touches `t` et `g` pour l'avant-bras.

Les détails d'animation du bras se trouvent dans la méthode *Animate* de la classe *Robot*.

2.5 Collisions avec les murs

Pour éviter que le robot passe à travers les murs, vous devrez implémenter la détection et la résolution de collisions avec les murs du labyrinthe. On suppose que le robot reste en tout temps au sol, on ne portera donc pas attention aux collisions avec le plancher. Le robot sera approximé par un cercle sur le plan XZ de rayon 1,25 et les murs seront considérés comme des segments, aussi dans le plan XZ.

Cette approximation n'est pas la plus réaliste, mais simplifie beaucoup les calculs. Pour plus de précision, nous pourrions calculer le point du robot le plus éloigné de son centre sur le plan XZ et prendre ce point comme référence pour calculer le rayon de notre cercle. Cependant, en prenant un cercle de rayon fixe un peu plus grand que le rayon du torse, nous nous assurons quecelui-ci ne dépassera pas le mur. Par contre, à leur extension, les jambes traverseront légèrement le mur, c'est normal. Le but est simplement d'empêcher que le robot puisse traverser complètement.

Si un des segments intersecte le cercle du robot, vous devrez déplacer le robot de telle sorte qu'il n'y ait plus d'intersection. Une autre façon de voir le problème est de considérer que la position du robot ne doit pas être à une distance inférieure à 1,25 d'un segment (voir figure 2)

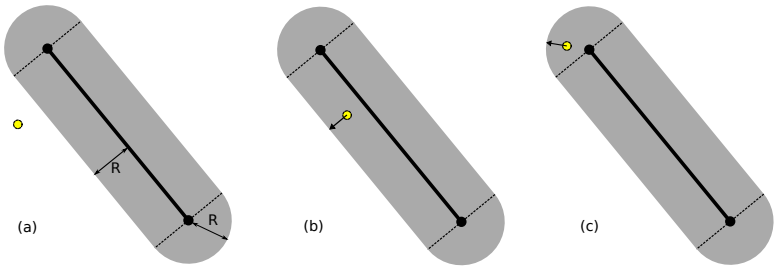

FIGURE 2 – Détection de collision entre un segment et un cercle de rayon $R = 1,25$. (a) Aucune intersection (b) Intersection intérieure (c) Intersection à l'extrémité. La correction à appliquer est indiquée par une flèche.

Vous devrez utiliser les matrices de transformation local→global et global→local pour convertir les coordonnées vers un référentiel commun, pour ensuite pouvoir détecter et résoudre la collision. La correction à appliquer sera différente si le robot se trouve dans zone intérieure ou dans une zone d'extrémité du segment. Dans la zone intérieure, vous devrez le déplacer dans la direction perpendiculaire au segment. Dans une zone d'extrémité, vous devrez le déplacer de telle sorte que la distance avec le point d'extrémité soit égale à 1,25.

Gardez cette partie pour la fin afin de vous laisser le temps de bien comprendre le système de hiérarchie et la manipulation des transformations. Les détails du système de collisions se trouvent dans la méthode *ResolveWorld* de la classe *Collisions*.

2.6 Cible

Pour terminer le labyrinthe, le robot devra toucher à la cible sphérique jaune avec sa main droite. Si un point de la main, que vous aurez choisi, se situe à l'intérieur de la cible, vous devrez changer la couleur de la sphère comme signe de réussite.

Les détails du test de la cible se trouvent dans la méthode *TestGoal* de la classe *Collisions*.

3 Support au développement

Un squelette du programme que vous aurez à concevoir est disponible sur le réseau du DIRO. Copiez dans votre compte l'archive tp1.tar.gz disponible dans le répertoire ∼dift3355/pub/tp/tp1. Ensuite, exécutez

```
tar -zxvf tp1.tar.gz
cd tp1
make debug
```

4 Outils disponibles

Vous avez à votre disposition un prototype qui démontre le résultat final attendu. Celui-ci peut aussi être utile pour l'assemblage des objets de la scène. Par ailleurs, nous vous fournissons les classes Matrix4 et Vector3. Ces classes seront utiles pour la construction de vos matrices de transformation. Entre autres, vous apprécierez les fonctionnalités suivantes :

1. `Matrix4::{Pre|Post}Translate( Vector3 t )` pour une translation par le vecteur t,
2. `Matrix4::{Pre|Post}Rotate( Vector3 v, θ )` pour rotation autour de l'axe v,
3. `Matrix4::operator*( Matrix4 m )` pour la multiplication matricielle,
4. `Matrix4::operator*( Vector3 v )` pour la multiplication avec le vecteur v en coordonnées homogènes, en supposant que la 4ème coordonnée est égale à 1,
5. `Vector3::operator*( Vector3 v )` pour calculer le produit scalaire.

Pour plus d'informations consultez lesdites classes.

5 Délivrables et rapport du travail

Nous aurons besoin d'une version électronique de votre programme pour l'évaluer. Pour la remise, exécutez les commandes suivantes :

```
% ssh remise
% cd <repertoire racine du projet>
% make dist # Crée l'archive à remettre
% tar zft tp1.tar.gz # Vérifie le contenu de l'archive
% remise ift3355 tp1 tp1.tar.gz
% remise -v ift3355 tp1 tp1.tar.gz # Vérifie la remise
% exit
```

Vous devez aussi écrire un rapport d'environ 3-4 pages qui contiendra :

1. L'énoncé du problème à résoudre.
2. La description des techniques utilisées. Décrivez les principes théoriques que vous avez appliqués pour obtenir les matrices de transformation. Spécifiez de quelle façon vous avez animé le robot, incluant la façon dont vous avez procédé pour vous assurer que les pieds ne glissent pas au sol. Décrivez la méthode employée pour traiter les collisions. Nous voulons ici le fondement de vos techniques, pas les détails de la disposition du code.

3. Les problèmes reliés aux techniques implémentées, suggestions pour améliorer la solution au problème, extensions possibles.
4. Références.

6 Critères d'évaluation

Section	Critères	Pondération
Code	style, clarté, efficacité, résultats	80%
Rapport	esthétique, énoncé, descriptions	10%
Analyse	problèmes, améliorations, extensions	10%

7 Bibliographie

P. Shirley et S. Marschner. Fundamentals of Computer Graphics. 3ème édition, AK Peters, 2009.