TP3 : Animation / Interactions
AnimationPath / Picking

4. Animations

Remarque : Les constantes littérales se multipliant, il est conseillé de les regrouper dans un fichier `CONSTANTES_LITTERALES.h` *que vous inclurez à chaque fois que vous aurez besoin d'une constante.*

4.1. Véhicule

4.1.1. Déplacement automatique / animationPath:

Ajouter à la classe Démineur 3 méthodes pour déplacer le mobile d'une position à une autre.

```
void  paramétrerDéplacement(Vec3    posDep,    Vec3    posArr,    double    duree,
      AnimationPath::LoopMode repetition = AnimationPath::SWING);
void  démarerDéplacement(bool   réinitialiser = FALSE);
void  stopperDéplacement() ;
```

Ci-dessous un exemple de code pour définir une animation sur un nœud _unNoeudDeTransformation :

```
animationPathDeplacementAuto = new osg::AnimationPath();
animationPathDeplacementAuto->setLoopMode(repetition);

AnimationPath::ControlPoint cp0(posDep);
AnimationPath::ControlPoint cp1(posArr);
animationPathDeplacementAuto->insert(0         ,cp0);
animationPathDeplacementAuto->insert(duree     ,cp1);

// le callback qui va se charger de "jouer" l'animation
apcAnimationCallBack =
          new osg::AnimationPathCallback( animationPathDeplacementAuto.get() );
apcAnimationCallBack ->setPause(true);

// associer le callback à un noeud de transformation (trsf ou pat)
_unNoeudDeTransformation->setUpdateCallback( apcAnimationCallBack.get() );
```

Proposer une méthode `démarerDéplacement qui va définir` animationPath (une donnée membre de la classe) et l'affecter au bon nœud de transformation (une autre donnée membre) :

```
// déplace en [duree] secondes le demineur de la position [posDep]
// à la position [posArr]
// si le parametre [posdep] = (0,0,0) alors la position de départ est la
position actuelle   et le parametre [posArr] est un vecteur déplacement (ou
position relative)
void Demineur:: paramétrerDéplacement
                              (    Vec3                        posDep,
                                   Vec3                        posArr,
                                   double                      duree,
                                   AnimationPath::LoopMode     repetition)
```

Ci-dessous deux exemples d'appel à cette méthode :

```
// déplacement relatif (4,0,0,) par rapport à pos actuelle
monOBJ_Demineur->paramétrerDéplacement(
                        Vec3(0,0,0),
                        Vec3(4,0,0),
                        6,
                        AnimationPath::NO_LOOPING);
// déplacement absolu de  (0,3,altitude) à (0,-3, altitude )
monOBJ_Demineur->paramétrerDéplacement(
                        Vec3(0, 3,ROUE_RAYON + HAUTEUR/2),
                        Vec3(0,-3,ROUE_RAYON + HAUTEUR/2),
                        6,
                        AnimationPath::LOOP);
```

Que se passe t'il si on exécuter ces deux instructions l'une à la suite de l'autre ?
(indice : pas plus d'un callback par nœud)

Le but est maintenant de contrôler le démarrage et l'arrêt de l'animation.
Vous allez associer le callback au nœud de transformation avec `setUpdateCallback` uniquement lors du démarrage de l'animation (`démarerDéplacement`) et de le retirer avec `removeUpdateCallback` lors de l'arrêt de l'animation (`stopperDéplacement`)

Pour info :

Une autre possibilité est d'utiliser la fonction `setPause()` : la fixer à TRUE lors du paramétrage de l'animation (`paramétrerDéplacement`) et de l'arrêt de l'animation (`stopperDéplacement`) et à FALSE pour démarrer l'animation (`démarrerDéplacement`)

⇨ *A voir, le code source de l'exemple `osganimate`.*

Pour info : faire varier la vitesse:

Si on souhaite un déplacement non linéaire sur toute la durée, il suffit d'ajouter (au moins) un point de contrôle intermédiaire. La vitesse de déplacement sera toujours constante entre 2 points de contrôle (interpolation linéaire) mais pourra être différente d'un intervalle à l'autre. S'il est possible de définir plusieurs points de contrôle, la variation se fera donc par pallier.

```
      AnimationPath::ControlPoint cp0(posDep);
      animationPath->insert(0            ,cp0);
      AnimationPath::ControlPoint cp1(posDep+(posArr-posDep)/2);
      animationPath->insert(duree/4.0    ,cp1);
      AnimationPath::ControlPoint cp2(posArr);
      animationPath->insert(duree        ,cp2);
```

Aller plus loin :

L'utilisation des points de contrôle comme les clefs d'une timeline a ses limites car cela reste une interpolation linéaire par morceaux (entre deux points de contrôle). Pour rendre réaliste une animation, il faudrait définir de nombreux points de contrôle à l'aide de suite sigmoïde par exemple. Il existe une solution avec les classes `xxxMotion` de la bibliothèque `osgAnimation`. Par exemple, la classe `osganimation ::InOutCubicMotion` permettrait de simuler une accélération ou décélération plus réaliste. Ces classes pourraient être utilisées pour définir des points de contrôle d'un objet `AnimationPathCallback` ou, mieux, être utilisées dans la méthode opérateur d'un callback « maison » (voir plus loin).

⇨ *A voir, le code source de l'exemple `osganimationeasemotion`.*

4.2. Pour une simulation réaliste : des roues qui tournent

L'objectif est de faire tourner les roues … pour l'instant sans arrêt et à vitesse constante.
Il faut ajouter des `UpdateCallback` aux nœuds de transformation des roues. Ces callback modifieront ces nœuds à chaque phase update, donc à chaque frame.

⇨ Placer et Comprendre la méthode ci-dessous :

```
void rotationAutomatique(     ref_ptr<PositionAttitudeTransform>  noeud,
                              float       angle,
                              Vec3d       axe,
                              Double      duree,
                              AnimationPath::LoopMode repetition){
Vec3d       pos = noeud->getPosition();
Quat        orientationDeb= noeud->asPositionAttitudeTransform()->getAttitude();
// définir l'animation et ses 3 points de contrôle
ref_ptr<osg::AnimationPath> animationPath(new osg::AnimationPath);
animationPath->setLoopMode(repetition);
animationPath->insert(0 ,AnimationPath::ControlPoint(pos,orientationDeb));
animationPath->insert(duree/2,
            AnimationPath::ControlPoint(
                  pos,orientationDeb*Quat(DegreesToRadians(angle/2), axe)));
animationPath->insert(duree
            ,AnimationPath::ControlPoint(
                  pos,orientationDeb*Quat(DegreesToRadians(angle), axe)));
// Créer le callback
ref_ptr<AnimationPathCallback> apCallBack =
                  new osg::AnimationPathCallback( animationPath.get() );
// associer le callback au noeud de transformation
noeud->setUpdateCallback( apCallBack.get() );
}
```

⇨ Utiliser cette méthode pour faire tourner les 4 roues.

Pour info :
*La position de la roue ne varie pas mais son orientation change. Comme pour la position, le callback se charge de faire l'interpolation entre les orientations des différents points de contrôle. 3 points de contrôle sont ici suffisants : le 1er = position initiale, le 2ème = 1er * rotation de 180° , le 3ème = 2ème * rotation de 180°(=retour à l'orientation initiale)*

4.3. IHM / Picking : L'utilisateur contrôle le déplacement

L'objectif est de pouvoir :

- Cliquer un point sur le plan ou sur un objet : le mobile va se rendre sur cet objectif.
- Orienter le véhicule vers cette direction puis déplacer le véhicule vers cette position cible.

Pour le changement d'orientation et le déplacement, utiliser `AnimationPath` et `AnimationPathCallback` déjà abordés plus haut.

⇨ Récupérer les entrées clavier et souris

Créer une classe derivée `osgGA::GUIEventHandler` qui va récupérer les événements et lancer les traitements associés :

```
class Picking_Cible : public osgGA::GUIEventHandler  {
…
}
```

Associer cette classe pour viewer pour l'activer :

```
viewer_1->addEventHandler(new Picking_Cible(…));
```

Dans la classe derivée `osgGA::GUIEventHandler` , c'est la fonction `handle` qui reçoit les événements ! Redéfinir cette méthode :

```
bool Picking_Cible_Demineur::handle(      const GUIEventAdapter& ea,
                                          GUIActionAdapter& aa) {
switch(ea.getEventType())
    {
    case(GUIEventAdapter::RELEASE):  {
            osgViewer::Viewer* viewer = dynamic_cast<osgViewer::Viewer*>(&aa);
            if (!viewer) return false;
            pickCibleObjetADeplacer (viewer, ea.getX(),  ea.getY());
            return true;
        }
    default:return false;
    }
}
```

⇨ La fonction de picking va rechercher le plus proche objet de la scène se trouvant « sous » le pointeur de la souris, puis calculer le point d'intersection 3D qui va devenir la position cible de l'objet à déplacer :

```
void Picking_Cible_Demineur::pickCible (Viewer* viewer, float mx,float my){
// définir le rayon pour l'intersection
ref_ptr< LineSegmentIntersector > intersector =
                  new LineSegmentIntersector(Intersector::WINDOW, mx, my);
// définir le visitor
osgUtil::IntersectionVisitor visitor_intersections(intersector.get());
// exécuter le visitor : parcours du graph de scene pour connaître les objets
intersectés par le rayon
viewer->getCamera()->accept(visitor_intersections);
// on traite les résultats obtenus
if (intersector->containsIntersections()) {
      //on prend le premier objet
      LineSegmentIntersector::Intersection intersection
                              = intersector->getFirstIntersection();
      // on récupère le point d'intersection : nécessaire si objet sélectionné
est trés large , donc sa position pas utile
      osg::Vec3d pos_cible = intersection.getWorldIntersectPoint();

      // à FAIRE  : déplacer le démineur vers sa position cible (pos_cible)
      // paramétrerDéplacement(…);
      // démarerDéplacement();
}
```

```
}
```

Pour déplacer l'objet, il faudrait ré-utiliser les méthodes déjà définies : `paramétrerDéplacement` et `démarerDéplacement`. Pour les rendre accessible à la classe `Picking_Cible_Demineur`, il faut passer l'objet par son constructeur.

Conseils :

Pour le picking, voir les exemples `osgkeyboardmouse` et `osgpick`

- ⇨ Pour un déplacement plus réaliste, faire pivoter le démineur pour l'orienter vers sa destination. A nouveau, il va falloir s'inspirer des méthodes définies ci-dessus (`rotationAutomatique` et `xxxDeplacement`) pour créer deux nouvelles méthodes : `paramétrerRotation()` et `démarerRotation()`.

- ⇨ Une autre difficulté est ici est de déterminer de quel angle il faut tourner l'objet.

Calculs:

***Le vecteur vitesse** (position Cible – position actuelle) donne le sens de déplacement.*

*Un vecteur d'orientation (initialisé à `Vec3(-1,0,0)` en début d'application) multiplié par la transformation du mobile (méthode `getAttitude()` du nœud `PositionAttitudeTransform`) donne **le vecteur d'orientation** actuelle.*

Une fois ces vecteurs normalisés, leur produit scalaire donnera le cos de l'angle entre ces vecteurs. Le déterminant (xy' –x'y) donnera le sinus de l'angle. Les inverses Cos^{-1} et Sin^{-1} (fonction `acos` et `asin`) vous indiqueront l'angle entre l'orientation actuelle et la direction à suivre pour rejoindre la position cible.

```
      // à FAIRE  : orienter  le démineur vers sa cible
      // paramétrerRotation(…) ;
      // démarerRotation(…) ;
      // à FAIRE  : déplacer le démineur vers sa position cible (pos_cible)
      // paramétrerDéplacement(…);
      // démarerDéplacement();
```

- ⇨ Que constatez vous à l'exécution? Pourquoi ? *(rappel : un seul callback par nœud)*

- ⇨ La solution la plus "simple" est de définir notre propre classe `DeuxAnimationPathCallback` dérivée de `AnimationPathCallback` qui permettrait d'exécuter 2 animations successives, en particulier en redéfinissant la méthode `operator()`. Dans cette fonction, des tests à chaque frame permettent de savoir si la première animation est terminée (comparer les réponses de `getAnimationTime` et `getPeriod`), et dans ce cas de charger et démarrer l'animation suivante. Le reste du traitement n'étantpas spécifique, il faut ré-utiliser le traitement de la classe de base : appel de la méthode `operator()` de la classe mère.

Ci-dessous la définition de la classe dérivée (fichier .h):

```
class DeuxAnimationPathCallback : public osg::AnimationPathCallback {
protected :
      double      _duree_animation;
      osg::ref_ptr<osg::AnimationPath> _anim_suivante;
public :
DeuxAnimationPathCallback(    osg::AnimationPath * anim_path_1,
```

```
                                osg::AnimationPath * anim_path_2)
                    :     osg::AnimationPathCallback(anim_path_1),
                          _duree_animation(anim_path_1->getPeriod()),
                          _anim_suivante(anim_path_2)
                    {}
// à faire : redéfinition de la méthode opérator
virtual void      operator() (osg::Node *node, osg::NodeVisitor *nv);
};
```

Ci-dessous le corps de la méthode `operator` (fichier .cpp):

```
void DeuxAnimationPathCallback ::operator() (   osg::Node *node,
                                                osg::NodeVisitor *nv){

      // comportement original du AnimationPathCallback conservé
      //en appelant la méthode originale (de la classe de base)
      osg::AnimationPathCallback::operator()(node,nv);

      // notre comportement particulier de de la classe dérivée commence ici

      // si pas un visitor UPDATE , alors on ne fait rien !
      if ( ! dynamic_cast<osgUtil::UpdateVisitor*> (nv) ) return;

      // si c'est un UPDATE visitor :
      // à faire : redéfinition de la méthode opérator
      // si animation 1 terminée => charger et démarrer animation 2
      // si animation 2 terminée => supprimer le callback du noeud
}
```