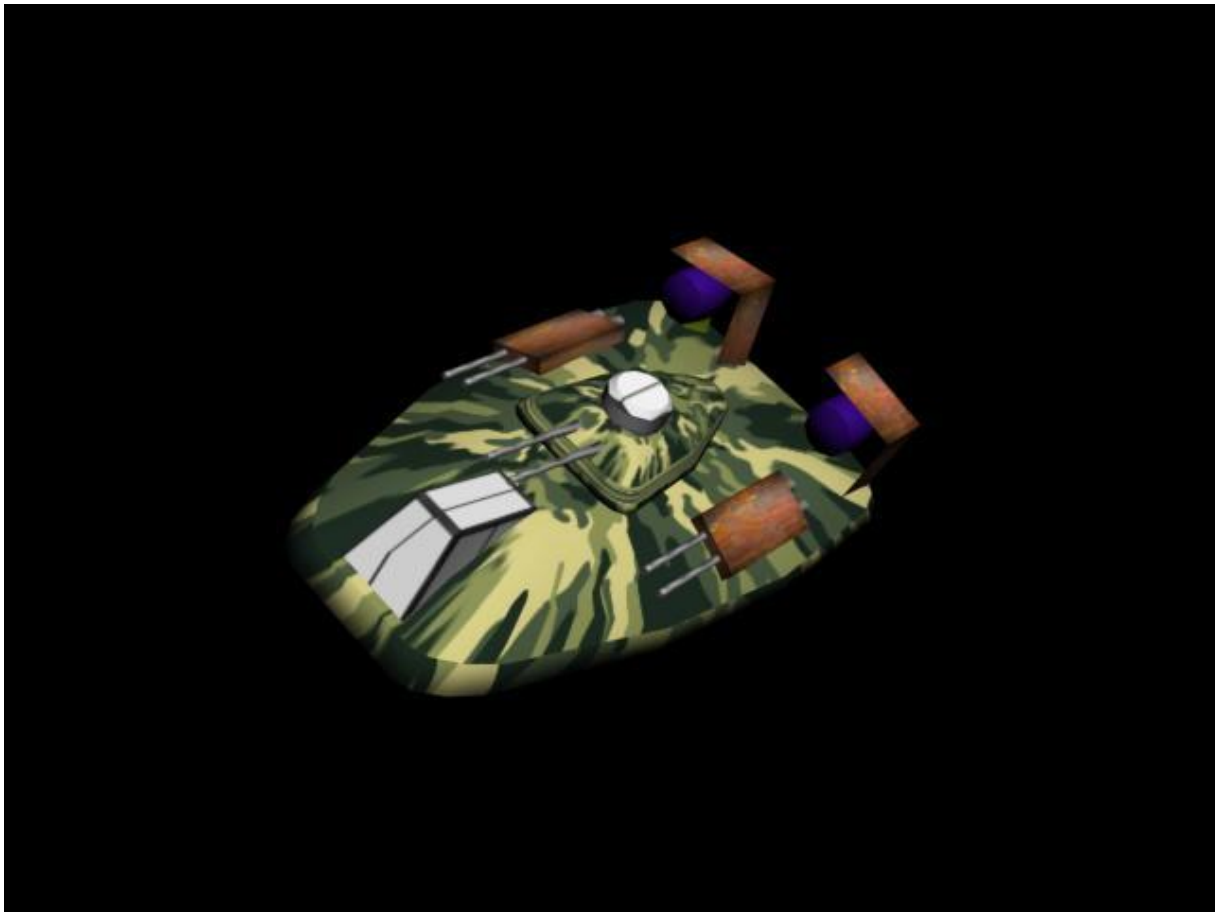
Option Jeux Vidéo

Documentation de la plateforme de développement.

Auteurs : F. Lamarche et R. Cozot

ESIR, Université de Rennes 1.



1 SOMMAIRE

2 INSTALLATION

Cette section contient les informations nécessaires à l'installation du projet de Jeux vidéo. Suivez bien les instructions car dans le cas contraire, le projet pourrait ne pas compiler. La section 2.3 sur la recompilation des dépendances ne vous est par directement utile mais est fournie au cas où vous souhaiteriez recompiler les projets pour des plateformes 64 bits par exemple.

2.1 STRUCTURE DES REPERTOIRES

La structure des répertoires associés à la plateforme est la suivante :

- *dependencies*. Ce répertoire contient des versions pré-compilées des dépendances de la plateforme.
- *dependencies_rebuilt*. Ce répertoire contient le code source des dépendances de la plateforme.
- *RTS*. Ce répertoire contient Les sources de la plateforme

2.2 INSTALLATION A PARTIR DES VERSIONS PRE-COMPILEES

2.2.1 CHOIX DES DEPENDANCES

Le répertoire *dependencies* contient des versions pré-compilées de Ogre 3D (le moteur de rendu utilisé pour le jeu) et de boost (une bibliothèque C++ offrant des compléments à la STL). Dans ce répertoire des versions précompilées pour Visual Studio 2008 (fichiers ayant 2008 en fin de nom) et pour Visual Studio 2010 (fichiers ayant 2010 en fin de nom) vous sont fournies.

En fonction de votre environnement de développement, décompressez les archives adéquates. Ces archives doivent être décompressées dans le répertoire courant, elles contiennent un répertoire englobant dont le nom est normalisé.

2.2.2 COMPILATION DE LA PLATEFORME RTS

Allez dans le répertoire RTS. Ouvrez la solution « RTS – vs 2010 » si vous travaillez sous visual studio 2010 ou « RTS.sln » si vous travaillez sous visual studio 2008. A partir de cette solution, vous pouvez compiler la plateforme en mode debug et release.

2.2.3 CONFIGURATION A EFFECTUER SI VOUS TRAVAILLEZ DANS LES SALLES DE TP DE L'ESIR / ISTIC

La configuration par défaut du projet génère les fichiers intermédiaires de compilation dans le répertoire courant. Ceci fonctionne parfaitement sur vos ordinateurs personnels. Cependant, lorsque vous travaillez à l'ESIR / ISTIC, vos comptes sont sur des disques durs réseau. Cela implique certaines lenteurs. Afin d'accélérer la compilation, vous pouvez effectuer la manipulation suivante :

1. Ouvrez la fenêtre de propriétés du projet RTS.
2. A gauche, sélectionnez l'onglet « Propriétés de configuration / Général »
3. Dans la fenêtre de droite, remplacez le répertoire intermédiaire (actuellement « $(Configuration) ») par le répertoire « $(TEMP)\$(Configuration) ».

Les fichiers intermédiaires seront générés sur le disque dur de la machine plutôt que sur votre répertoire réseau. Les temps de compilation seront grandement réduits.

2.3 RECOMPILATION DES DEPENDANCES

2.3.1 COMPILATION DE OGRE

Voici les étapes à suivre pour créer le projet visual studio permettant de compiler Ogre 3D sur votre machine.

1. Ouvrez CMake GUI.
2. Dans la boite de saisie « Where is the source code », indiquez le chemin du répertoire *Ogre_1.7.3/ogre_src_v1-7-3*
3. Dans la boite de saisie « Where to build the binaries », indiquez le chemin du répertoire *Ogre_1.7.3/ogre_src_v1-7-3* et ajoutez */project* à la fin de ce répertoire. Il s'agit du répertoire dans lequel les fichiers relatifs à la compilation sous visual studio seront générés.
4. Cliquez sur configure deux fois puis sur générer.

A la fin de l'étape de génération, le répertoire *Ogre_1.7.3/ogre_src_v1-7-3* contiendra un sous répertoire *project* dans lequel vous trouverez le fichier solution de visual studio : *Ogre.sln.* Ouvrez ce fichier et compilez Ogre 3D en mode debug et en mode release.

2.3.2 COMPILATION DE LA PLATEFORME RTS

Allez dans le répertoire RTS. Ouvrez la solution RTS.sln. A partir de cette solution, vous pouvez compiler la plateforme en mode debug et release.

La solution et le projet ont été créés sous Visual Studio 2008, si vous utilisez Visual Studio 2010, suivez le processus de conversion des projets automatiquement proposé à l'ouverture de la solution.

3 PACKAGES UTILITAIRES

3.1 LA LIBRAIRIE DE MATHEMATIQUES

La librairie de mathématiques est définie dans l'espace de nommage *Math*.

3.1.1 VECTEURS

Vous avez à votre disposition trois classes de vecteurs

- *Math::Vector<Float, dimensions>.* Cette classe définit un vecteur générique prenant en paramètre générique le type de scalaire et la dimension du vecteur.
- *Math::Vector2<Float>.* Cette classe est une spécialisation du vecteur générique pour les vecteurs en deux dimensions.
- *Math::Vector3<Float>.* Cete classe est une spécialisation du vecteur générique pour la gestion des vecteurs en trois dimensions.

3.1.2 GEOMETRIE

La librairie de mathématiques met à disposition plusieurs classes permettant de gérer des éléments de géométrie 2D :

- *Circle2D<Float>*. Une classe générique permettant de gérer des cercles en 2D.
- *Line2D<Float>*. Une classe générique permettant de gérer des lignes en 2D.
- *Segment2D<Float>*. Une classe générique permettant de gérer des segments en 2D.
- *Triangle2D<Float>*. Une classe générique permettant de gérer des triangles en 2D.

Dans le fichier « Math/distance2D.h », vous trouverez différentes surcharges de la fonction *distance* prenant en paramètre deux formes géométriques (incluant les vecteurs 2D pour représenter des points) et permettant de calculer la distance entre ces deux géométries.

3.1.3 LES TRANSFORMATIONS 2D

Afin de faciliter la gestion des transformations, la classe *Math::Matrix3x3<Float>* est fournie. Cette classe représente une matrice 3x3 et met à disposition des fonctionnalités permettant de gérer des transformations 2D en coordonnées homogènes.

Les méthodes de classe suivantes permettent de créer des matrice homogènes correspondant aux transformations classiques :

- *Matrix3x3<Float>::getRotationHomogeneous.* Récupération d'une matrice de rotation.
- *Matrix3x3<Float>::getTranslationHomogeneous.* Récupération d'une matrice de translation.
- *Matrix3x3<Float>::getScaleHomogeneous.* Récupération d'une matrice de mise à l'échelle.

L'opérateur '*' est redéfini entre matrices et entre matrice et forme géométrique 2D pour faciliter la gestion des transformations géométriques.

3.1.4 AUTRES ELEMENTS

Le fichier *« Math/Constants.h »* contient la définition de différentes constantes mathématiques comme pi par exemple.

Le fichier « *Math/finite.h* » contient différentes fonctions permettant de traiter les valeurs particulières de flottant (infini, not a number, etc…).

3.2 LES STRUCTURES DE DONNEES

L'espace de nommage *DataStructure* est réservé à la définition de structures de données. Pour le moment, seule la classe *DataStructure::Grid<Type>* est définie. Il s'agit d'une grille régulière pour la laquelle la cellule contient un élément de type *Type*.

4 GESTION DES MESSAGES (COMMUNICATION ENTRE OBJETS)

Le système proposé dispose d'un gestionnaire de messages. Ce dernier gère la communication par messages de manière synchrone. Autrement dit, lorsqu'un message est émis, il est immédiatement transmis aux écouteurs du message qui effectuent immédiatement les traitements adéquats.

4.1 LES FONDATIONS DU GESTIONNAIRE DE MESSAGE

4.1.1 LES CLASSES PRIMITIVES

Le gestionnaire de messages repose sur deux classes génériques : *System::MessageEmitter<MessageType>* et *System::MessageListener<MessageType>*. Le paramètre générique MessageType correspond au type de message envoyé / reçu. Il n'y a aucune contrainte sur ce type de données, vous pouvez définir vous-mêmes vos types de messages via une structure / classe C++ classique.

Voici une description plus précise des classes de gestion de messages :

- *MessageEmitter<MessageType>*. Cette classe correspond à un émetteur de messages de type *MessageType*. Elle possède une méthode *send(MessageType const & msg)* qui doit être appelée pour émettre un message
- *MessageListener<MessageType>*. Cette classe correspond à un écouteur de messages de type *MessageType*. Elle possède une méthode abstraite *onMessage(MessageType const & msg)* qui est appelée à la réception d'un message.

4.1.2 FONCTIONNEMENT DE L'EMISSION / RECEPTION DE MESSAGES

L'émetteur de messages. Une classe souhaitant émettre des messages d'un type donné doit soit hériter de la classe *MessageEmitter<MessageType>* ou posséder un attribut de ce type. La création d'un émetteur ne nécessite aucun paramètre.

L'écouteur de messages. Un écouteur de message doit forcément hériter de la classe *MessageListener<MessageType>*. Le constructeur d'un écouteur de message prend en paramètre un pointeur sur l'émetteur de messages qu'il écoute ; cela permet de gérer l'abonnement. Cette classe possède une méthode abstraite *onMessage(MessageType const & msg)* qui est appelée lors de l'émission d'un message par le émetteur de message auquel cet écouteur est abonné. Pour créer un écouteur de message, il faut onc hériter de la classe *MessageListener<MessageType>* et implémenter la méthode *onMessage* de manière à ce qu'elle réalise les traitements adéquats lors de la réception d'un message.

Envoi de message et réception. L'envoi d'un message est réalisé par un appel à la méthode *send(MessageType cont & msg)* sur une instance de *MessageEmitter<MessageType>.* Lorsque cet appel est réalisé, l'ensemble des écouteurs abonnés à cet émetteur est prévenu par un appel à la méthode *onMessage* des instances de *MessageListener<MessageType>* abonnées. L'émission / réception est synchrone, autrement dit, l'appel aux méthodes *onMessage* des écouteurs se fait dès l'émission du message. Il faut donc faire attention à ne pas faire de boucles de messages.

4.2 CLASSES UTILITAIRES FOURNIES

4.2.1 UN ECOUTEUR DE MESSAGES VIA CALLBACK

La classe *System::CallbackMessageListener<MessageType>* est une implémentation de la classe *MessageListener<MessageType>* permettant d'appeler une méthode prenant une référence constante sur une instance de type *MessageType* en paramètre, sur une instance de classe quelconque à la réception d'un message.

Une instance de cette classe d'écouteur se construit en fournissant un pointeur sur une instance de *MessageEmitter<MessageType>* à laquelle cet écouteur s'abonne. Ensuite, la méthode générique setCallback(ClassType * instance, void (ClassType ::*methodPtr)(const MessageType &)) doit être appelée pour signaler l'instance et la méthode qui doivent être appelés lors de la réception du message.

Voici un exemple d'utilisation de cette classe :

```
class MyClass
{
	void print(const std::string & message)
	{
		::std::cout<<this<<" received: "<<message ;
	}
};

void test()
{
	// Emmiter of messages of type string
	System::MessageEmitter<::std::string> stringEmitter ;
	// Listener of messages of type string, listening messages sent by stringEmitter
	System::CallbackMessageListener<::std::string> listener(&stringEmitter) ;
	// An test of type MyClass
	MyClass test ;
	// Sets the callback. When string emitter sends a message, it is forwarded by listener
	// by calling the method print on instance test
	listener.setCallback(&test, &MyClass::print) ;
	// Message is sent, method print of test is called.
	// printed message is the adress of test followed by "received: Hello world"
	stringEmitter.send("Hello world\n") ;

}
```

4.2.2 MESSAGES DE CONSTRUCTION / DESTRUCTION D'OBJETS

Pour faciliter le référencement des objets dans les différents contrôleurs d'une application, une méthode utilisée peut-être d'envoyer un message de construction d'objet lors de la création d'une instance et de destruction d'objet lors de la destruction d'une instance. De cette manière, les contrôleurs peuvent s'abonner à ces messages et être constamment au courant des créations / suppressions d'instances d'un certain type.

Le système qui vous est fourni propose deux classes de messages envoyés à la construction / destruction d'un objet:

- *System::ConstructionMessage<Type>* est un message qui doit être envoyé lors de la construction d'une instance de type *Type*.
- *System::DestructionMessage<Type>* est un message qui doit être envoyé lors de la destruction d'une instance de type *Type*.

Pour faciliter la gestion de l'envoi / réception de ces messages, deux autres classes vous sont fournies :

- *System::ConstructionDestructionEmitter<Type>*. Cette classe propose deux méthodes de classe
    - *static void sendConstruction(Type & object)*. Cette méthode envoie un message de type *System::ConstructionMessage<Type>* signalant la construction de l'objet *object*.
    - *static void sendDestruction(Type & object)*. Cette méthode envoie un message de type *System::DestructionMessage<Type>* signalant la destruction de l'objet *object*.
- *System::ConstructionDestructionListener<Type>*. Cette classe est un écouteur de messages de construction / destruction d'objets de type *Type*. Elle propose deux méthodes qui sont appelées à la réception des messages :
    - *void onCreateObject(ConstructionMessage<Type> const & msg)*. Cette méthode est appelée lors de la réception d'un message de type *ConstructionMessage<Type>.*
    - *void onDestroyObject(DestructionMessage<Type> const & msg).* Cette méthode est appelée lors de la réception d'un message de type *DestructionMessage<Type>*.

  Pour écouter les messages de construction / destruction, il faut donc hériter de la classe *System::ConstructionDestructionListener<Type>* et implémenter les méthodes *onCreateObject* et *onDestroyObject* pour effectuer les traitements adéquats.

5 LE SIMULATEUR

5.1 LES OBJETS ACTIFS

La plateforme définit le concept d'objet actif qui est un objet qui est mis à jour régulièrement. Les objet actifs sont de type *System::ActiveObject* et possèdent une méthode abstraite *update(Config::Real const & dt)* qui est une méthode appelée régulièrement et qui prend en paramètre le temps écoulé (en secondes) depuis le dernier appel.

Le rôle de ces objets actifs est de vous permettre de gérer les différents acteurs intervenant dans votre jeu. Chaque acteur doit donc hériter de la classe *System::ActiveObject* et implémenter la méthode *update*. Dans l'implémentation de cette méthode, il faut définir le comportement de l'objet.

5.2 LE CONTROLEUR

Les objets actifs sont gérés par un contrôleur dont le rôle est d'activer les objets les uns après les autres pour leur permettre d'effectuer leurs calculs. Pour que la simulation s'exécute, il faut tout d'abord (*avant la création des objets actifs*) créer une instance de contrôleur (*classe System::Controller*). Les objets actifs et le contrôleur utilisent le système de messages de construction / destruction (voir 4.2.2) pour le référencement. Autrement dit, une fois un contrôleur créé, ce dernier sait automatiquement quels sont les objets actifs créés et détruits ; il prend donc automatiquement en charge leur exécution.

Pour mettre à jour les objets actifs gérés par le contrôleur, il faut appeler la méthode *update(Config::Real const & dt)*. Cet appel doit être effectué dans la boucle de rendu de manière à ce que les objets soient remis à jours à chaque affichage.

6 TRIGGERS ET DETECTION DE COLLISION

La plateforme met à disposition un détecteur de collisions entre objets 2D afin de facilement détecter les collisions entre éléments ou encore gérer la perception des entités. Dans ce détecteur de collision il existe deux types d'objets :

- Les entités : il s'agit d'objets devant réagir aux collisions avec d'autres entités. Elles représentent des entités physiquement présentes dans le monde.
- Les triggers : il s'agit d'objets qui ne sont pas physiquement présents dans le monde et qui ont pour rôle de détecter les entités entrant en collision avec eux. Le rôle de ces triggers est principalement de gérer la perception des entités.

Le moteur de détection de collision cherche les pairs d'objets en collision. Lorsque deux entités sont en collision, cette collision est signalée aux deux entités concernées. Lorsqu'une collision est détectée entre une entité et un trigger, seul le trigger est prévenu de la collision. Si deux triggers sont en collision, rien ne se passe.

6.1 LES OBJETS MANIPULES PAR LE DETECTEUR DE COLLISION

Les objets manipulés par le détecteur de collision sont de type *Triggers::CollisionObject*. Ces objets peuvent être déplacés et orientés de manière à suivre le déplacement des entités. Ils peuvent être de deux types : entité ou trigger en fonction de leur rôle. Ils possèdent une forme géométrique associée. Cette forme géométrique est définie par des iso-surfaces autour d'un squelette qui peut être un point, une ligne ou encore un triangle.

6.1.1 LES OBJETS DE COLLISION

Les objets de collision sont de type *Triggers::CollisionObject.* Lors de leur création, il faut fournir leur type qui est décrit par le type énuméré *Triggers::CollisionObject::CollisionObjetType.* Ce type énuméré possède deux valeurs :

- *Triggers::CollisionObject::entity* pour créer un objet se comportant comme une entité. Cet objet, lorsqu'il entre en collision avec un autre objet de type *entity* reçoit un message signalant la collision et contenant l'entité qui est entrée en collision.
- *Triggers::CollisionObject::trigger* pour créer un objet se comportant comme un trigger. Cet objet agit comme un observateur des objets de type *entity* passant dans son voisinage. Lorsqu'un objet de type *entity* entre en collision avec le *trigger*, le *trigger* est prévenu par un message, par contre, l'entité n'est pas prévenue.

Lors de leur création / destruction, ces objets émettent des messages de construction / destruction (voir 4.2.2) qui leur permettent d'être automatiquement référencés par le gestionnaire de collision.

Le placement et l'orientation des objets de collision sont gérés par l'appel à la méthode *Triggers::CollisionObject::setTransform(Math::Matrix3x3<Config::Real> const & transform)*. La matrice en paramètre de cette méthode correspond à la matrice de transformation 2D exprimée en coordonnées homogènes.

La classe *Triggers::CollisionObject* possède une méthode abstraite onCollision(CollisionMessage const & msg) qui est appelée lorsqu'une collision entre cet objet et un autre objet est détectée par le détecteur de collision. Pour réagir aux collisions, il faut donc implémenter cette méthode.

6.1.2 ASSOCIATION D'UNE FORME GEOMETRIQUE A UN OBJET DE COLLISION

Pour matérialiser la forme d'un objet de collision, la classe *Triggers::Shape* est utilisée. Cette classe possède trois constructeurs permettant de créer une forme de collision sur la base d'un squelette et d'une distance à ce squelette. Le squelette utilisé peut être de type :

- Triggers::Shape::Vector2 pour un squelette de type point.
- Triggers::Shape::Segment2 pour un squelette de type segment.
- Triggers::Shape::Triangle2 pour un squelette de type triangle

Une fois créée, une forme géométrique de type *Triggers::Shape* peut être positionnée dans son repère local par appel à la méthode *setTransform(Math::Matrix3x3<Config::Real> const & transform)*. Cette méthode prend en paramètre une matrice de transformation 2D en coordonnées homogènes.

Pour associer cette forme géométrique à un objet de collision, il faut appeler la méthode *CollisionObjet::addShape(Triggers::Shape const & shape)*.

Plusieurs formes géométriques peuvent être associées à un objet de collision, lui permettant de posséder une forme complexe. Le placement relatif des formes géométriques est géré par l'appel à la méthode *setTransform* de la classe *Triggers::Shape*.

6.2 LES DETECTEURS DE COLLISION FOURNIS

La plateforme met à disposition deux classes de détecteur de collision :

- *Triggers::BasicCollisionDetector*. Cette classe contient un détecteur de collisions implémenté de manière naïve et ayant donc une complexité en $O(n^2)$, n étant le nombre d'objets de collision traités.
- *Triggers::SweepAndPrune.* Cette classe contient un détecteur de collisions implémenté avec l'algorithme « sweep and prune » qui est beaucoup plus efficace que l'implémentation naïve.

Le détecteur de collision est un objet actif (voir 5.1), autrement dit, lors de sa création, il est automatiquement référencé dans le contrôleur de l'application et effectue donc ses calculs à chaque activation du contrôleur.

Les deux détecteurs de collision fonctionnent exactement sur le même principe :

- Lorsqu'une collision est détectée entre deux entités (*CollisionObject* de type *Triggers::CollisionObject::entity*), la méthode *onCollision* est appelée sur les deux objets en collision.
- Lorsqu'une collision est détectée entre une entité et un trigger(*CollisionObject* de type *Triggers::CollisionObject::trigger*), la méthode *onCollision* est appelée sur le trigger uniquement.
- A chaque détection de collision, le détecteur de collision émet un message de type *Triggers::CollisionObject::ColllisionMessage*. L'émetteur de se message se trouve dans la classe *CollisionDetectorBase* et peut être récupéré par appel à la méthode *getCollisionEmitter* de cette classe.

7 LE FRAMEWORK OGRE 3D

7.1 L'APPLICATION PRINCIPALE

L'application principale est décrite par la classe *OgreFramework::MainApplication* qui hérite de la classe *OgreFramework::BaseApplication* (une classe du tutoriel de Ogre 3D très légèrement modifiée).

La classe *OgreFramework::MainApplication* possède plusieurs méthodes virtuelles (héritées *OgreFramework::BaseApplication*) de que vous pourrez utiliser / compléter pour réaliser votre application.

Gestion des événements claviers :

- *virtual bool keyPressed(const OIS::KeyEvent &arg)*. Cette méthode est appelée lorsqu'une touche est enfoncée.
- *virtual bool keyReleased(const OIS::KeyEvent &arg)*. Cette méthode est appelée lorsqu'une touche est relâchée.

Gestion des événements souris :

- *virtual bool mouseMoved(const OIS::MouseEvent &arg)*. Méthode appelée lorsque la souris bouge dans la fenêtre Ogre.
- *virtual bool mousePressed(const OIS::MouseEvent &arg, OIS::MouseButtonID id)*. Méthode appelée lorsqu'un boutn de la souris est enfoncé.
- *virtual bool mouseReleased(const OIS::MouseEvent &arg, OIS::MouseButtonID id)*. Méthode appelée lorsqu'un bouton de la souris est relâché.

Création de la scène et configuration de l'interface :

- *virtual void createScene(void)*. Cette méthode est appelée au lancement de l'application pour créer la scène 3D et configurer l'application.

Boucle de simulation :

- *virtual void update(Ogre::Real dt)*. Cette méthode est appelée à chaque rendu d'image par le moteur de Ogre 3D. C'est dans cette méthode qu'il faut ajouter les traitements propre à la mise à jour de votre moteur.

7.2 LE CHARGEUR DE GEOMETRIE

La classe *OgreFramework::GeometryLoader* est une classe facilitant le chargement de géométries. Elle permet de charger des fichiers au format *mesh* (format standard de Ogre 3D) et au format *scene* (format xml décrivant un graphe de scènes). Une fois une instance de cette classe créée, il suffit d'appeler l'une des méthodes suivantes pour charger vos objets :

- Ogre::SceneNode * loadMesh(std::string const & fileName, bool yUp=true). Cette méthode charge un fichier au format *mesh*. Le paramètre yUp, s'il est à vrai signale que l'objet 3D est modélisé avec Y up et corrige donc l'orientation de manière à obtenir un objet avec Z up.
- Ogre::SceneNode * loadScene(std::string const & fileName, bool yUp=true). Cette méthode charge un fichier au format *scene*. Le paramètre yUp, s'il est à vrai signale que l'objet 3D est modélisé avec Y up et corrige donc l'orientation de manière à obtenir un objet avec Z up.

Ces deux méthodes renvoient un nœud du graphe de scènes Ogre. Ce nœud possède une transformation égale à l'identité. Autrement dit, si vous souhaitez placer l'objet dans la scène, lui appliquer des transformations, ce nœud est à votre disposition. Les objets chargés sont directement intégrés au graphe de scène et le nœud renvoyé est fils du nœud racine.

7.3 LE PICKING

Afin de faciliter la sélection des objets par l'interface graphique, les classes dérivées de *OgreFramework::Picking* et *OgreFramework::PickableObject* vous sont fournies.

Une classe dérivée de *OgreFramework::Picking* permet de gérer la sélections d'objets par picking. Ces classes sont actuellement au nombre de deux :

- *OgreFramework::PickingBoundingBox*. Le picking est effectué sur la boite englobante des objets géométriques.
- *OgreFramework::PickingSelectionBuffer*. Le picking est effectué sur la géométrie de l'objet et est précis au pixel près.

Une instance de l'une de ces deux classes doit être créée dans l'application principale. Afin de gérer ce picking, un appel à la méthode *OgreFramework::Picking::update(const OIS::MouseEvent &arg, OIS::MouseButtonID id)* doit être effectué depuis la méthode *mousePressed* de l'application principale.

Les objets sélectionnables via la méthode de picking doivent dériver de la classe *OgreFramework::PickableObject*. Cette classe défini deux méthodes abstraites en relation avec le picking :

- *virtual void onSelect()*. Cette méthode est appelée lorsque l'objet est sélectionné via le picking.
- *virtual void onUnselect().* Cette méthode est appelée lorsque l'objet est désélectionné via le picking.

Pour créer une instance d'une classe héritant de *OgreFramework::PickableObject*, il faut posséder un pointeur sur une instance d'un nœud du graphe de scène de Ogre 3D (de type *Ogre::SceneNode*). Lorsqu'un descendant de ce nœud sera sélectionné / désélectionné, les méthodes *onSelect* et *onUnelect* seront automatiquement appelées par le gestionnaire de picking.

7.4 ENCAPSULATION DES OBJETS OGRE 3D

Afin de faciliter la mise en correspondance entre les objets Ogre 3D et vos objets de simulation, la classe *OgreFramework::EntityAdapter* vous est fournie. Son rôle est double :

1. Cette classe encapsule une instance de *Ogre::SceneNode* et met à disposition des méthodes permettant de gérer les translation et rotation de cet objet en utilisant les classes de la plateforme et non les classes fournies par Ogre 3D. Cela vous permet de vous abstraire d'Ogre 3D.
2. Cette classe implémente la classe abstraite *OgreFramework::PickableObject* de manière à ce que tous les objets de type *OgreFramework::EntityAdapter* puissent être sélectionnés par picking.

8 LES ELEMENTS DE JEU

8.1 LA CLASSE GAMEOBJECT

La classe *GameElements::GameObject* est la classe mère de tous les éléments de jeux ayant une représentation graphique et étant dotés d'un comportement. Cette classe hérite des classes suivantes :

- *System::ActiveObject*. Cela permet à un élément de jeu d'être considéré comme un élément actif (voir 5.1) et donc d'être doté d'un comportement et d'être remis à jour régulièrement.
- *Triggers::CollisionObject*. Cela permet à un élément de jeu d'avoir une représentation physique dans le moteur de détection de collision (voir 6.1.1).
- *OgreFramework::EntityAdapter*. Cela permet à un élément de jeu d'avoir une représentation 3D et de pouvoir être sélectionné via le picking (voir 7.2).

Les éléments de jeu que vous implémenterez lors de la réalisation de ce projet devront donc tous hériter de la classe GameElements::GameObject pour s'intégrer facilement au sein de la plateforme.

8.2 LES AGENTS

La classe *GameElements ::Agent* est la classe mère de toutes les classes implémentant un agent doté de points de vie. Elle hérite de la classe *GameElements::GameObject*. Cette classe n'implémente aucune méthode particulière, sont rôle est d'ajouter des attributs permettant de gérer les points de vie d'un agent ainsi que ses points d'armure. La gestion de ces points de vie de d'armure est effectuée par la classe *GameElements ::Bullet* décrite dans la section suivante.

8.3 LES BALLES

Les balles sont décrites par des classes dérivant de la classe *GameElements::BulletBase*. Cette classe, mère de toutes les balles pouvant être tirées par un aéroglisseur gère la collision entre une balle et un élément du jeu. Lorsque la collision est effective, les points de vie et d'armure associés aux éléments de jeux sont retirés en fonction des caractéristiques de la balle. La classe *GameElements::BulletBase* hérite de la classe *GameElements::GameObject* et possède donc une représentation graphique, une forme de collision et s'avère être un objet actif doté d'un comportement.

Dans son état actuel, la plateforme de simulation met deux types prédéfinis de balles à votre disposition :

- *GameElements::StraightBullet*. Il s'agit d'une balle possédant une trajectoire parallèle au sol. Cette dernière explose lors de l'impact avec un objet du jeu (un agent ou une autre balle de ce type).
- *GameElements::BallisticBullet*. Il s'agit d'un obus ayant une trajectoire en cloche. Cet obus explose lorsqu'il tombe au sol. Il possède une aire d'impact et inflige des dégâts à toutes les entités située dans cette aire d'impact.

8.4 LES ARMES

Les armes sont gérées par la classe *GameElements::Weapon*. Cette arme est construite à partir d'un archétype d'arme (classe GameElements::WeaponsArchetypes::Archetype) qui décrit les propriétés des balles associées à cette arme ainsi que la fréquence de tir de l'arme en tant que tel. Pour tirer, la méthode *Weapon::fire* est à votre disposition. Cette méthode prend comme paramètre le point source de la balle ainsi que le point

destination. L'appel à cette méthode génère une balle (voir section précédente) qui est lancée dans l'environnement.

La classe *GameElements::Weapon* met aussi à votre disposition des méthodes permettant de mettre des filtres décrivant les objets qui ne doivent pas subir de dommages s'ils entrent en collision avec les balles.

9 LA GESTION MEMOIRE

9.1 UTILISATION DE « SMART POINTERS »

Les différentes classes dérivant des classes *System::ActiveObject* et de *Triggers::CollisionObject* sont gérées par un système de « smart pointer », autrement, des pointeurs avec compteurs de référence.

Le système de « smart pointers » utilisé est celui fourni par la bibliothèque *boost*[1] et plus précisément les « intrusive smart pointers ». Les deux classe cités ci-dessus héritent donc de la classe *Ext::boost::reference_count* qui a pour rôle d'associer un compteur de référence à chaque objet héritant de cette dernière.

Cela impose quelques contraintes lors de la programmation :

- Les instances de ces classes ainsi que de leurs classes dérivées doivent être systématiquement crées dynamiquement.
- Les pointeurs utilisés pour désigner une instance d'une classe X héritant d'au moins une des classes citées ci-dessus **doivent** être du type *boost::intrusive_ptr<A>* où A est une classe dont X hérite ou X lui-même. Vous êtes invités à vous référer à la documentation de la bibliothèque *boost* sur l'utilisation de la classe générique *boost::intrusive_ptr<T>*.
- Il ne faut en aucun cas manipuler ces instances via des pointeurs C++ classiques car ces derniers ne comptent pas les références. Il se pourrait alors que votre programme ait des comportements étranges (plantages plus ou moins « aléatoires » etc…).
- Il ne faut en aucun cas détruire les objets à la main. Lors de la destruction de la dernière instance de *boost::intrusive_ptr<T>*, le destructeur de l'instance désignée sera appelé et la mémoire allouée sera libérée.

Pour vous faciliter la vie, toutes les classes fournies dans le projet de jeu vidéo et héritant des classes *System::ActiveObject* et *Triggers::CollisionObject* possèdent un type interne nommé *Pointer* qui est une notation raccourcie pour le type *boost::intrusive_ptr<T>*, T étant la classe définissant le type interne *Pointer*.

9.2 DEMANDE DE DESTRUCTION DES OBJETS DE SIMULATION : METHODE DESTROY()

Les classes héritant des classes *Triggers::CollisionObject* et *System::ActiveObject* possèdent une méthode nommée destroy. Cette méthode a pour rôle de demander la destruction des objets au système gérant la simulation.

Si à prime abord cette méthode peut sembler redondante avec l'utilisation de « smart pointers » pour la gestion mémoire, il n'en est rien. Le système de simulation conserve des « smart pointers » vers les objets simulés. Lorsque que l'on souhaite qu'un objet ne soit plus géré par le système, il faut en informer ce dernier : il s'agit du rôle de la méthode destroy().

Lorsque vous souhaitez qu'un objet simulé (héritant de *System::ActiveObject*) soit dérérencé du système de simulation, il faut faire un appel à sa méthode destroy(). Cette méthode émettra les messages adéquats afin de provoquer le déréférencement. Cela devrait aboutir à la destruction de l'objet lorsque le dernier « smart pointer » le référençant est détruit.

[1] http://www.boost.org

Attention. Si vous implémentez une classe héritant de *System::ActiveObject* et qu'il y a un risque d'auto référencement (instance se désignant elle-même via des smart pointers), vous pouvez redéfinir la méthode destroy de manière à ce que les données pouvant provoquer cet auto référencement soient nettoyées. Cependant, si vous redéfinissez cette méthode, n'oubliez jamais de faire appel à la définition de cette méthode dans la classe mère, la stabilité du système en dépend !

10 LA CONFIGURATION DE L'APPLICATION ET ACCES AUX INFORMATIONS GLOBALES

10.1 FICHIER XML DE CONFIGURATION

Le fichier de configuration de l'application est décrit dans le format XML. Ce dernier est organisé en trois parties : la description des armes et de leurs caractéristiques, la description des unités et de leurs caractéristiques ainsi que la description des cartes.

10.1.1 DESCRIPTION DE LA LISTE DES ARMES

La description des armes s'effectue par l'intermédiaire de l'entité *<weapons>*. Cette entité contient une liste de description des armes. Chaque description d'arme est réalisée par l'intermédiaire de l'entité *<weapon>*. Cette entité décrit une arme via plusieurs attributs :

- *name* : le nom, sous forme textuelle, de l'arme.
- *strenght* : le nombre de points de dommage réalisés par cette arme.
- *armorDamage* : le nombre de points de dommages réalisé à l'armure de l'adversaire.
- *frequency* : la cadence de tir de l'arme en balles par secondes.
- *extent* : le rayon de l'aire de dégâts, en mètres, de l'arme lors d'un impact.
- *type* : défini le type de tir. Cet attribut peut prendre deux valeurs
    - straight : il s'agit d'un tir droit. Lorsque la balle touche un élément sur sa trajectoire, elle explose et afflige des dégaits.
    - ballistic : il s'agit d'un tir en cloche. La balle explose lors de l'impact au sol en passant par-dessus tous les obstacles.
- *bulletMesh* : la représentation graphique de la balle. Par défaut, la plateforme fournit le fichier « JetEngine.scene » qui contient un générateur de particules montrant la trajectoire de la balle.
- *scale* : le facteur d'échelle associé à la géométrie de la balle.
- *speed* : la vitesse de la balle en mètres par seconde.

10.1.2 DESCRIPTION DES UNITES

La liste des archétypes d'unités est contenus dans les éléments fils de l'élément *<units>*. Chaque archétype d'unité est décrit via un élément *<unit>*. Cet élément accepte les attributs suivants :

- *name* : nom de l'archétype.
- *mesh* : le nom du fichier graphique représentant l'unité.
- *scale* : le facteur d'échelle appliqué à la géométrie de l'unité.
- *cost* : le prix de l'unité.
- *perceptionRange* : La distance de perception de l'unité. Tout éléments à une distance supérieure n'est pas perçu par cette unité.
- *speed* : la vitesse de déplacement de l'unité en mètres par seconde.
- *life* : le nombre de points de vie de l'unité.
- *armor* : le nombre de points d'armure de l'unité.
- *weapon* : le nom de l'arme associée à cette unité. Il est à noter que dans la description actuelle, les unités ne possèdent qu'une seule arme.

10.1.3 DESCRIPTION DES CARTES

Les cartes de jeu sont décrites via un élément *<maps>* contenant la liste des cartes disponibles. Chaque carte est décrite dans un élément *<map>* qui accepte les attributs suivants :

- *name* : le nom de la carte.
- *mesh* : la représentation graphique de la carte.
- *scale* : le facteur d'échelle appliqué à la géométrie de la carte.
- *mapData* : une image encodant la structure de la carte. Cette image peut être dans plusieurs formats tels que jpg, png…

L'image décrivant la structure de la carte correspond à une grille régulière où chaque pixel représente une cellule et où la couleur du pixel encode une information sur la nature de la zone. Autrement dit, il existe une correspondance entre la couleur du pixel et des informations liées à la simulation. La description du décodage de cette carte s'effectue via un ensemble d'éléments fils de l'élément *<map>*. Ces éléments fils sont de type *<ground>* . L'élément *<ground>* possède les attributs suivants :

- *name* : la catégorie de sol décrite
- *speedReduction* : le facteur de réduction de la vitesse de déplacement impliquée par ce type de sol.
- *red*, *green*, *blue* : les composantes rouge, vert, bleu de la couleur associé à ce type de sol dans l'image décrivant la carte.

Pour le moment, la description des éléments de sol ne permet que de décrire des réductions de vitesse impliquées par la nature du sol. Cet aspect pourra éventuellement être étendu si vous en ressentez le besoin. Il s'agira alors d'ajouter des attributs à l'élément *<ground>*, voire des éléments fils si besoin. Dans ce cas, il faudra aussi modifier le chargeur XML afin de prendre des nouvelles informations en compte.

10.1.4 EXEMPLE DE FICHIER DE CONFIGURATION XML

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
        <!-- weapons description -->
        <weapons>
                <weapon         name="MousticWeapon"
                                strength="2"
                                armorDamage="0"
                                frequency="15"
                                extent="0"
                                type="straight"
                                range="30"
                                bulletMesh="JetEngine.scene"
                                scale="0.01"
                                speed="8"
                />

                <weapon         name="CrocoWeapon"
                                strength="5"
                                armorDamage="0"
                                frequency="7"
```

```
                                extent="0"
                                type="straight"
                                range="20"
                                bulletMesh="JetEngine.scene"
                                scale="0.01"
                                speed="5"
                        />
        </weapons>

        <!-- Units descriptions -->
        <units>
                <!-- Blue units -->
                <unit           name="MousticB"
                                mesh="MousticBleu01.scene"
                                scale="0.01"
                                cost="250"
                                perceptionRange="10"
                                speed="6"
                                life="400"
                                armor="0"
                                weapon="MousticWeapon"
                />

                <unit           name="CrocoB"
                                mesh="CrocoBleu01.scene"
                                scale="0.01"
                                cost="500"
                                perceptionRange="15"
                                speed="4"
                                life="800"
                                armor="0"
                                weapon="CrocoWeapon"
                />
        </units>

        <!-- Maps description -->
        <maps>
                <map    name="map01"   mesh="mapFinal.scene" scale="0.1" mapData="mapJPG.jpg">
                        <ground name="Grass" speedReduction="0" red="0" green="255" blue="0"/>
                        <ground name="Water" speedReduction="0.5" red="255" green="255" blue="0"/>
                        <ground name="Rock" speedReduction="1" red="255" green="0" blue="0"/>
                </map>
        </maps>
</configuration>
```

10.2 ACCES A LA CONFIGURATION DANS LA PLATEFORME

La configuration globale de l'application est accessible via la classe *OgreFramework::GlobalConfiguration*. Cette classe propose une certain nombre de méthodes de classe vous permettant d'accéder à diverse informations partagées :

- Le gestionnaire de scène ogre 3D utilisé dans l'application.
- Le chargeur de géométrie.
- Le contrôleur d'application.
- Le chargeur de configuration.
- Le système de gestion du son.

Cette classe vous offre aussi quelques fonctions utilitaires globales :

- Consultation / modification de la carte courante.
- Une méthode permettant de jouer un son.