# ODFAEG (Opensource Development Framework Adapted For Every Game) Tutorials. (French version)

## Sommaire :

# Chapitre I : Installation de ODFAEG.

Prérequis :

Avant d'installer ODFAEG il vous faudra installer 2 autres petites librairies, un compilateur c++ qui supporte le nouveau standard c++14 (gcc version 4.9.x) ainsi que cmake, rassurer vous, c'est très rapide à installer :

*Installation des outils de compilation.*

**Sous linux** il suffit d'ouvrir un terminal, de se connecter en mode super utilisateur et de taper cette commande apt-get install gcc-4.9 g++-4..9 cmake-gui **pour les plateformes 64 bits** ou bien apt-get install gcc-4.9-multilib g++-4.9-multilib cmake-gui et passer le flag -m32 lors de la compilation **pour les plateformes 32 bits.**

Il vous faudra peut être aussi installer la version 4.8 de gcc également si il vous avez des erreurs de linkage avec certains fonctions de la STL.

**Sous windows** vous devez aller sur ce site : http://tdm-gcc.tdragon.net/ et télécharger un compilateur tdm-gcc **32 bits pour les plateformes 32 bits ou 64 bits pour les plateformes 64 bits.** Ensuite, installer cmake-gui : http://www.cmake.org/cmake/resources/software.html

*Installation de la librairie SFML.*

Pour installer SFML tout est expliqué sur le site web du développeur principal : http://en.sfml-dev.org

Pas besoin donc d'en dire plus.

ODFAEG utilise SFML pour faire des applications portable et utiliser l'encapsulation qu'offre SFML sur toute les bibliothèques codée en C qui sont utilisée par les différents modules de ODFAEG.

*Installation de openSSL.*

*Fortement recommandée si vous voulez faire des applications sécurisée qui communique en réseau, cette librairie est utilisé par ODFAEG pour chiffrer les informations qui transite sur un réseau.*

*Sur linux il suffit de taper la commande apt-get install libssl-dev.*

*Sous windows la meilleur méthode est de télécharger MSYS :*
http://sourceforge.net/projects/mingw/files/

*Ensuite télécharger les sources de openssl :* https://www.openssl.org/

*Installer ensuite perl :* http://www.perl.org/get.html

*Lancer MSYS, aller dans le répertoire de openssl et ensuite, taper la commande suivante :*

*perl Configure mingw*

*Aller ensuite dans le fichier MakeFile, chercher la ligne PLATFORM et remplacer mingw par* ***TDM-GCC-32 pour les plateformes 32 bits et TDM-GCC-64 pour les plateformes 64 bits.(Sinon ça ne compile pas)***

Taper ensuite les commandes suivante : make et make install pour installer la librairie.

Installation de la librairie ODFAEG :

Une fois ces quelques outils d'installés, il ne vous reste plus qu'à installer ODFAEG, son installation est similaire à l'installation de SFML, ODFAEG ne fournis pas de librairie pré-compilée, vous devez donc télécharger le code source  ouvrir Cmake et la compiler.

Cliquer sur browse source et spécifier l'endroit ou vous avez télécharger ODFAEG.

Cliquer ensuite sur browse build et créer un nouveau dossier dans le répertoire ODFAEG, appeller le par exemple ODFAEG-build.

Cliquez ensuite sur le bouton Configure choissez l'option Specify native compiler et indiquer l'endroit ou se trouve vos compilateur c et c++. (*C:/TDM-GCC-32*/bin ou bien *C:/TDM-GCC-64/bin* sous windows ou bien /usr/bin sous linux.)

Là, il vous affiche une série d'option Cmake include préfix correspond à l'endroit ou vous voulez installer ODFAEG.

**Attention : pour les plateformes 32 bits spécifier la valeur i386 pour CMAKE_ARCHITECTURE,  sinon, spécifier la valeur i686 pour les plateformes 64 bits.**

**Si ça ne compile pas, décocher l'option Build shared lib. (Il se peut que le mode dynamique ne soit pas encore supporté)**

Lorsque tout est bon cliqué à nouveau sur configure et ensuite sur generate.

Ouvrez un terminal et aller dans le dossier ODFAEG-build (là ou se trouve le Makefile de ODFAEG)
Sous windows taper les commandes mingw32-make et mingw-make install pour installé la librairies.
Sous linux taper les commandes make et make install.

Et voilà ODFAEG est installé.

Il ne vous reste plus qu'à créer un projet dans votre EDI préféré et de créer un nouveau projet, n'oublier pas de lier les différentes bibliothèque de ODFAEG à votre projet (qui se trouve dans le dossier lib de ODFAEG) et de préciser l'endroit ou se trouvent les fichiers d'en tête de ODFAEG, de SFML et d'openssl. (Dans les dossier include de ODFAEG et de SFML et de openssl)
Il vous faudra aussi lier les librairies suivantes à votre projet :

-GLEW en static.
-SFML en dynamic.
-Opengl.
-Openssl et gdi32 si vous être sous windows. (ou bien libssl et libcrypto si vous être sous linux)

N'oublier pas de définir la macro ODFAEG_STATIC si vous avez compilé la librairie en mode

static.

Voilà normalement vous devez pouvoir compiler un projet utilisant ODFAEG, une fois que tout ceci est bon vous pouvez passé au chapitre suivant.

# Chapitre II : Création d'une application ODFAEG-

# LIKE.

Définition de l'application :

Pour créer une application « odfaeg-like » il faut hériter de la classe odfaeg::application. (Créer donc une nouvelle classe et faîtes la hériter de odfaeg::application.)

Si votre application est une application graphique, vous devez appeler le constructeur de la classe parent (celui de la classe odfaeg::Application) en lui passant les paramètres suivants :
sf::VideoMode, std::string (titre de la fenêtre)
MyAppli(sf::VideoMode wm, std::string title) : Application (wm, title)

Ensuite vous pouvez redéfinir les méthodes suivantes :

onLoad()

Méthode à redéfinir si vous avez des ressources externes à charger.

onInit()

C'est dans cette méthode que vous créerez et initialiserai tout ce qui doit être initialiser. (Les commandes, les entités graphiques, la définition des collisions, etc...)

onRender(odfaeg::FastRenderComponentManager *frcm)

Dans cette méthode vous pouvez définir ce qui devra être dessiné sur la frame courante en le dessinant sur un composant de rendu ODFAEG. (Le composant de rendu se chargera de rendre vos dessins de la manière la plus optimale suivant les fonctionnalités supportées par votre carte graphique)

Si cette méthode n'est pas redéfinie, ou bien que l'application n'est pas une application graphique, rien n'est dessiné.

onDisplay(odfaeg::RenderWindow *window)

Cette méthode est à redéfinir si vous devez dessiner des choses directement sur la fenêtre de rendu.

onUpdate(sf::Event& event)

Dans cette méthode vous pouvez mettre à jour la frame suivante. (Celle-ci sera mise à jour en même temps que le dessin de la frame courante à l'aide d'un thread.)

Ce qui rend le processus de rendu plus rapide. (Si l'application n'est pas une application graphique ou si cette méthode n'est pas redéfinie, rien n'est remis à jour)
Cette méthode prend un paramètre : le dernière événement SFML généré par l'application.

OnExec()

Cette méthode est à redéfinir si votre application ne possède pas de fenêtre, c'est le cas par exemple des applications serveur, mais elle peut aussi être redéfinie si votre application doit effectuer des traitement en fin de boucle.

Création de l'application :

Pour créer une application il suffit d'appeler le constructeur en lui passant les bons paramètres suivant le type d'application  dans le fonction main :

MyAppli app(sf::VideoMode(800, 600), "Test odfaeg");

Le constructeur de la classe odfaeg::application peut prendre des paramètres supplémentaires comme par exemple :

un booléen qui indique si l'on veut activer le depthtest d'opengl ou non. (par défaut ce boonéen vaut true)

sf::Style : le style de la fenêtre.

sf::ContextSettings : permet de spécifier des options de rendu opengl plus détaillées.

Enfin, pour lancer l'application il suffit d'appeler la méthode exec() de la classe application, celle-ci renvoie un booléen qui indique si l'application a réussi à s'exécuté correctement ou si une erreur s'est produite :

return app.exec() ;

Arrêt de l'application :

Pour arrêter l'application, il suffit d'appelé la méthode stop dans votre classe qui hérite de application, par exemple, ici, nous voulons arrêter l'application lorsque l'utilisateur ferme la fenêtre :

Nous devons donc rajouter ce code-ci dans la méthode onUpdate :

```
if (event.type == sf::Event::Closed) {

        stop();
}
```

Les classes de rendu graphique 2D d'odfaeg sont les même que celle de sfml à l'exception prêt qu'elles se trouvent dans le namepsace odfaeg et non dans le namespace sf. (Par exemple odfaeg::ConvexShape pour les formes convexe)

A la différence de SFML, les objets SFML-LIKE de odfaeg peuvent posséder des points en 3D pour rendre des entité en 2.5D. (C'est à dire des plans par exemple.)

Voilà maintenant vous savez créer des applications odfaeg toutes simple.

# Chapitre III : le système de gestion des ressources externes :

Charger et récupérer des ressource externes :

Pour charger des ressources externes, odfaeg utilise 2 classes :

ResourceManager : permet de charger toutes des ressources d'un seul et même type.

Resource cache : permet de charger des ressources de type différents :

Essayons par exemple de charger un tileset contenant toute des textures d'herbe, pour cela, il faut d'abord créer un gestionnaire de texture comme ceci :

odfaeg::ResourceManager<odfaeg::Texture> tm ;

Le 1er paramètre est le type des ressources externe et le second, le type de l'identifiant pour la ressource. (par défaut il vaut std::string)

odfaeg possède également quelques classes prédéfinie pour les classes de sfml chargeant des ressources : TextureManager, SoundManager, FontManager, ShaderManager.
Celle-ci ne prennent qu'un seul paramètre template qui vaut std::string par défault.
Voici comment instancier un gestionnaire de ressources de type odfaeg::Texture par exemple :
TextureManager<> tm ;

L'identifiant est comme un alias qui pointe vers une ressource, ici par exemple je vais créer une énumération qui référencera les différentes ressources dans l'application :
Il existe aussi des classes particulières pour les gestionnaires de ressources pour les types de ressources

```
enum TEXTURES {
	GRASS
};
```

Pour charger la ressource il suffit d'appeler la fonction fromFile du gestionnaitre de ressource si on veut la changer à partir d'un fichier :

tm->fromFile("tilesets/herbe.png", GRASS);

Le 1er paramètre est l'emplacement de la ressource par rapport à l'application, le second paramètre est l'alias qui pointe vers la ressource. (On n'est pas obligé de le précisé, dans ce cas la ressource ne sera accessible que par son 1er paramètre c'est à dire sont chemin, mais c'est déconseillé de procédé

comme ça.)

On peut ajouter les gestionnaires de ressources dans un cache, ceci est pratique si vous devez charger des ressources externes de différent types.
La classe resource cache prend un paramètre template : le type de l'identifiant qui référencera le gestionnaire de ressource. (Par défault le type est std::string)

ResourceCache<> cache ;
cache.addResourceManager(tm, "TextureManager");
Cette fonction prend en paramètre le gestionnaire de ressources à stocker dans le cache ainsi que le nom du ressource manager.

Ensuite vous pouvez récupérer le gestionnaire de ressource comme suit :

TextureManager<> tm = cache.resourceManager<odfaeg::Texture,
TEXTURES>("TextureManager");

En précisant le type de gestionnaire de ressources à récupérer dans le cache et le type de l'identifiant utilisé dans le gestionnaire de ressources..

Pour récupérer la ressource il suffit ensuite d'écrire ceci :

tm.getResourceByAlias(GRASS) ;

On peut récupérer la ressource de deux façons : soit par son alias ou bien par son chemin.

Charger des ressources avec son propre chargeur de ressource.

Supposons maintenant que nous voulons charger des ressources externes mais que le gestionnaire d'odfaeg ne possède pas de fonction pour charger ce type de ressource, pour cela il faut d'abord créer un chargeur personnalisé (par exemple un chargeur de fichier .3ds) et ensuite il faut passer la fonction qui charge la ressource au gestionnaire de ressources en créant un foncteur comme ceci :

std::function<bool(Object3DS,std::string> loadFunc(&MyLoader::fromFile) ;

Ensuite nous devons passez cette fonction à la fonction load du resource manager :

ResourceManager<Object3DS, OBJECTS3DS> om ;

om.load(loadFunc, « resource location ») ;

Voila, le chargeur de ressource externe de odfaeg n'est pas compliqué et possède pas mal de fonction utilitaire comme par exemple récupérer tout les chemins vers les ressources d'un même type qui ont été chargée.

La destruction des ressources :

La destruction des ressource ce fait automatiquement à la destruction du cache, lors de l'arrêt du programme, cependant, on peut le faire explicitement en appelant les fonctions deleteResouceByAlias ou bien deleteResourceByPath si on en a plus besoin.

# Chapitre IV : Le système de commandes de ODFAEG :

Créer des fonctions de callback :

odfaeg possède une classe spéciale pour pouvoir stocker des pointeurs de fonctions de type différents, des paramètres afin d'appeler ces fonctions plus tard, car, les commandes sont appelée dans un autre thread avec odfaeg que celui ou les arguments sont passé à la fonction de callback.
La seule contrainte est le type de retour de la fonctions de callback, qu'il faut mentionner à la classe.
Cette classe particulière porte le nom de odfaeg::FastDelegate. (Car elle va déléguer le travail à un autre foncteur)
Cette classe prend en paramètre le type de retour de la fonction de callback, et en arguments, un pointeur sur une fonction ainsi que les arguments de la fonctions de callback.
Pour ne pas devoir recréer le même delegate pour appeler plusieurs fois la même fonction de callback, la classe FastDelegate possède les fonctions setParam et bind pour les placeholders qui permettent de changer les paramètres de la fonction de callback avant de la rappeler.
La classe FastDelegate est très similaire à std::function et à std::bind, mais, à l'exception près que l'ont peut stocker tous les pointeurs de fonction retournant le même type dans un tableau si celle-ci on un nombre variable d'arguments ou bien des arguments de type différents, et c'est ce que fais la classe listener de odfaeg, cette classe permet de connecter des événements à des fonctions de callback et se charge d'appeler la bonne fonction de callback en fonction de l'événement généré.
Voici un exemple de code qui montre comment créer des delegates sur des fonctions de callback, les fonctions peuvent être soit des fonctions libres, des fonctions membres ou bien des fonctions anonymes, le passage par référence doit se faire via std::ref comme avec les std::function :

```
#include<iostream>
#include<string>
#include <fstream>
#include <functional>
#include "odfaeg/Core/fastDelegate.h"
#include "odfaeg/Core/serialization.impl"
using namespace std::literals;
using namespace std::placeholders;

void foo(int i, int j)
{ std::cout << i << j; }

struct A {
    A() {
        var = 10;
    }
        void foo(int i)
        { std::cout << i; }
        template <typename A>
        void serialize (A & ar) {
        ar(var);
        }
        int var;
};

struct B {
```

```
    B() {
        c = "serialize base";
    }
        virtual void foo()
        { std::cout << 1; }
        virtual void print() {
        std::cout<<c<<std::endl;
        }
        virtual ~B();
        std::string c;
};

B::~B(){}

struct C : B {
    C () {
        c = "serialize derived";
    }
    void foo();
    void print () {
        B::print();
        std::cout<<c<<std::endl;
    }
    std::string c;
};

void C::foo(){ std::cout << 2; }


int main (int argv, char* argc[]) {
    void(*f)(int, int) = &foo;
    odfaeg::FastDelegate<void> f1(f, 3, 4);
    f1.setParams(5, 6);
        f1();
        std::cout << std::endl;

        odfaeg::FastDelegate<void> f2(
                [](int i, int j){ std::cout << i << j; },
                7,8
        );
        f2();
        f2.setParams(9,10);
        f2();
        std::cout << std::endl;

        int i = 11;
        odfaeg::FastDelegate<void> f3(
                [i](int j){ std::cout << i << j; },
                12
        );
        f3();
        f3.setParams(13);
        f3();
        std::cout << std::endl;

        A a;
        odfaeg::FastDelegate<void> f4(&A::foo,&a,14);
        f4();
        f4.setParams(&a,15);
        f4();
        std::cout << std::endl;
        odfaeg::FastDelegate<void> f5 = f1;
        f5();
```

```
        f5=f3;
        f5();
        std::cout << std::endl;

        C c;
        B* b = &c;
        odfaeg::FastDelegate<void> f6(&C::foo,&c);
        f6();
        f6.setParams(b);
        f6();
        std::cout << std::endl;

        odfaeg::FastDelegate<void> f7(D(),16);
        f7();
        f7.setParams(17);
        f7();
        std::cout << std::endl;

        odfaeg::FastDelegate<void> f8(bar,"ab"s);
        f8();
        f8.setParams("abc"s);
        f8();
        std::cout << std::endl;
        int pi = 1;
    odfaeg::FastDelegate<void> f9(foo, &pi);
        f9();
        std::cout << std::endl;
        pi=2;
        f9();
        std::cout << std::endl;
        odfaeg::FastDelegate<int> f10(goo,18);
        std::cout << f10();
        f10.setParams(19);
        std::cout << f10();
        std::cout << std::endl;
        void(*fu)(int&) = &foo;
        int vi=1;
        odfaeg::FastDelegate<void> f11(fu, std::ref(vi));
        f11();
        std::cout << std::endl;
        vi=2;
        f11();
        std::cout<<std::endl;
        return 0;
}
```

Et avec des placeholders :

```
void f (int i, int j, int l) {
    std::cout<<"i : "<<i<<" j : "<<j<<" l : "<<l<<std::endl;
}

int main (int argv, char* argc[]) {
     odfaeg::FastDelegate<void> fd(&f, 1, 2, odfaeg::ph<0,int>()) ;
     fd.bind(3);
     fd() ;
}
```

Créer des commandes :

Maintenant que l'on sait comment créer des delegate, on va pouvoir créer des commandes que l'on

connectera ensuite au listener à l'aide d'un identifiant.

ODFAEG permet de créer des commandes de 2 façons :

La première façon est de définir un ou plusieurs événements SFML qui déclencheront la commande.

Pour cela, odfaeg possède une classe Action qui permet de combiner plusieurs évènements SFML entre eux.

Ici nous allons définir une action qui se déclenche lorsque une des 4 touches suivante (Z, Q, S, D) est enfoncée :

pour cela, nous allons définir 4 actions que nous allons combiner ensemble, de la même manière que lorsqu'on combine des conditions avec les opérateurs ||, &&, | et !.

```
Action a1 (Action::EVENT_TYPE::KEY_HELD_DOWN, sf::Keyboard::Key::Z);
Action a2 (Action::EVENT_TYPE::KEY_HELD_DOWN, sf::Keyboard::Key::Q);
Action a3 (Action::EVENT_TYPE::KEY_HELD_DOWN, sf::Keyboard::Key::S);
Action a4 (Action::EVENT_TYPE::KEY_HELD_DOWN, sf::Keyboard::Key::D);
Action combined (a1 || a2 || a3 || a4);
```

Cette action doit ensuite être mappée à une commande, ceci se fait à l'aide de la classe Command, la commande n'est rien d'autre qu'une action liées à un pointeur de fonction (appelé un slot) ainsi que des paramètres qui seront envoyé à ce slot, pour créer un slot avec odfaeg ils suffit de créer un object de type FastDelegate<void> en lui passant en paramètre  un pointeur sur la fonction qu'appellera la commande lorsqu'elle sera déclenchée ainsi que la valeur des paramètres de la commande (ici je met Unknow car on ne connaît pas encore la touche qui a été pressée et le temps écoulé depuis le début de l'application)

L'avantage majeur de la classe FastDelegate est qu'elle peut stocker n'importe quelle type de fonction.

```
Command moveCommand(combined, FastDelegate<void>(&MyAppli::keyHeldDown, this,
sf::Keyboard::Key::Unknown, realTime.restart()));
```

Nous lions donc notre action et notre slot à notre commande, voilà, nous avons créer la commande.

Connecter des commandes  :

Une fois que la commande est créée nous devons la connecter au gestionnaire d'événement (au listener), le gestionnaire d'événement n'est rien d'autre qu'un thread qui exécute les commandes qu'on lui fournis de manière synchrone ou assynchrone en fonction des événements SFML généré par l'application.

Pour connecter une commande il faut récupérer l'interpréteur de commande du gestionnaire d'événements, ceci se fait à l'aide de la méthode getListener()  de la classe InputSystem :

```
InputSystem::getListener().connect("MoveConnexion", moveCommand);
```

On fournis donc le nom de la connection ainsi que la commande.

Définir un trigger personnalisé pour une commande :

Ce système est bien mais ne convient pas pour tout les cas, imaginez que vous devez déclenché une commande lorsque la souris est dans un rectangle, le principal soucis est que vous devez définir une action pour chaque pixel contenu dans le rectangle, ce qui n'est pas du tout pratique.
ODFAEG permet de créer une commande avec une fonction spéciale. (appelée un trigger)
C'est alors la fonction et non plus l'action qui va décider de l'exécution de la commande, voici un exemple, créons une fonction qui affiche quelque chose à l'écran lorsque la souris est dans le rectangle 0, 0, 100, 100 de la fenêtre de l'application :

```
bool mouseInside (sf::Vector2f mousePos) {

    BoundingBox bx (0, 0, 0, 100, 100, 0);
    if (bx.isPointInside(Vec3f(mousePos.x, mousePos.y, 0))) {
        return true;
    }
    return false;
}
void onMouseInside (sf::Vector2f mousePos) {
    std::cout<<"Mouse inside : "<<mousePos.x<<" "<<mousePos.y<<std::endl;
}
```

Nous avons 2 méthodes, une qui vérifie si la souris est dans le rectangle et une autre qui affiche un message en indiquant la position de la souris.

Créer un trigger s'effectue de la même manière que la création d'un slot, à l'exception près que le type du trigger est FastDelegate<bool>.

Le type de fonction et les paramètres du trigger et du slot doivent être les mêmes !

La création d'une commande avec un trigger se fait de la même manière que la création de la commande avec une action à l'exception prêt que c'est le trigger qu'on envoie et non plus l'action car ici la commande n'est pas liée à un ou plusieurs événements SFML mais directement à une fonction) :

```
Command mouseInsideCommand(FastDelegate<bool>(&MyAppli::mouseInside,this,Vector2f(-1, -1)), FastDelegate<void>(&MyAppli::onMouseInside, this, Vector2f(-1,-1)));

InputSystem::getListener().connect(« mouseInsideCommand »,mouseInsideCommand) ;
```

On met -1, -1 pour la valeur de la position de la souris car on ne la connais pas encore, ceci évite d'afficher le message au lancement de l'application lorsque la fenêtre n'est pas encore affichée.

Il existe une dernière et troisième méthode qui permet de passer un trigger, une action et un slot à une Commande, la commande sera alors exécutée si le trigger ainsi que l'action sont déclenchés.

# Chapitre V : les vues.

Orientation des axes et origine des vues.

Les vues de odfaeg sont différentes des vues SFML, car elle n'imposent pas d'orientation particulière pour les axes, contrairement à SFML qui impose que l'axe y pointe vers le bas, c'est à dire que le coin en haut à gauche de la fenêtre a toujours pour coordonnée 0,0 et le coin en bas à droite à pour coordonnée la largeur et la hauteur de la fenêtre.
Odfaeg n'impose pas d'orientation pour les axes et on peut changer l'orientations des axes comme on le désire, contrairement à SFML, les vues on pour origine le center de la fenêtre et non pas le coin supérieur gauche de la fenêtre et le sens des axes est vers la droite pour l'axe x, vers la gauche pour l'axe y et vers vous pour l'axe z.
On peut changer facilement l'orientation des axes avec la méthode setScale comme ceci :

```
getView().setScale(1, -1, 1);
```
getView() permet de récupérer la vue de la fenêtre de rendu dans le l'application.

Dans ce cas là l'axe des y pointera vers le bas comme pour SFML !
Pour changer l'origine il suffit de faire une translation de la vue avant de tout dessiner.

*Les composants possèdent également une vue, par défaut, c'est la même vue que celle de la fenêtre de rendu de l'application.*

Pour déplacer la vue il suffit d'appeler la méthode move :

```
getView().move(0, 300, 0);
```

Ceci déplace la vue de 300 unités vers le bas ou vers le haut. (Cela dépend de l'orientation de l'axe y.)
Pour faire une rotation, il suffit d'appeler la méthode rotate et de passer l'angle et pour zoomer, il suffit d'appeler la fonction zoom et de passer l'échelle.

Par défault la vue est othographique. (en 2D donc)
Avec comme znear, une valeur un peut plus petite que 0 et comme z far une valeur un peu plus grande que la hauteur de la fenêtre, par conséquent tout ce qui est visible doit avoir un z compris entre 0 et la hauteur de la fenêtre sinon l'entité ne sera pas affichée. (Il suffit juste de mettre comme composante z le centre de l'entité 2D et de déplacer aussi la vue en z avec la même valeur que pour y lorsqu'on se déplace dans le monde. (avec le clavier ou bien avec la souris par exemple)

On peut changer la vue et la passer en perspective (en vue 3D) mais ce ne sera pas l'objet de se chapitre, nous verrons cela dans le tutoriel consacré à la 3D.
On peut également changer le viewport de la vue avec la méthode setViewport.

# Chapitre VI : les entités 2.5D de odfaeg.

Les entités de haut niveau de ODFAEG :

Les entités SFML-LIKE sont utile dans les petites jeux, mais restent fort limitées dans les gros jeux ou il y a beaucoup d'entités graphique de type différent.

ODFAEG permet donc de créer ses propres entités qui sont des entités plus évoluée que les entités SFML-LIKE qui permettent juste de tracer des formes, les entités ODFAEG se servent aussi des tableaux de sommets de SFML mais possèdent des attributs supplémentaires : des états, des volumes de collision, des faces, des matériaux, des sommets pré-transformé, des entités enfants et une entités parent, et quelque fonction que l'on peut redéfinir si nos entités doivent interagir avec d'autres entités afin d'éviter de devoir faire trop d'héritage)

Les faces peuvent contenir un ou plusieurs tableau de sommets utilisant le même matériel et le même type de primitive.
Les matériaux peuvent avoir une ou plusieurs textures (multi-texturing), ainsi qu'une couleur.

Les entités ODFAEG-LIKE permettent donc de faire des rendus de manière plus optimisé. (tout comme les commandes qui permettent de gérer des événements SFML de façon plus optimisée.)

ODFAEG possède déjà des entités 2D et 2.5D que j'utilise uniquement dans le but de faire des tests :
Ces entités se situent dans le module g2d du module graphique de ODFAEG.

Chaque entité possède un type qui est en fait une sorte de groupe, qui regroupe un ensemble d'entité de même type, voici les différents groupe d'entité 2D et 2.5D de ODFAEG :

odfaeg::Tile : similaire à un sprite à la SFML-Like mais hérite de la classe Entity du framework.

Odfaeg::BigTile : un ensemble de tiles qui forment une grand tile.

odfaeg::Wall : un mur. (Contient une tile et quelques informations supplémentaires pour la projection des ombres et les collisions avec la lumière, et un type (un int) qui sert à générer des cartes de façon aléatoire.)

odfaeg::Decor : tout modèle statique. (même chose que ci-dessus excepté le type)

odfaeg ::Animation : toutes entité qui évolue en fonction du temps, cette classe peut contenir plusieurs entités mais une seule ne peut être affichée en même temps.
Une animation peut contenir d'autres animations et ainsi interagir avec des animations enfants afin de créer des animations squelettique

Et chaque entité de l'animation correspond à une frame de l'animation.

odfaeg::Shadow, elles sont générées automatiquement par le framework donc pas besoin de s'en

soucier.;)

odfaeg::PonctualLight : une lumière poncutelle.

Créer des entités.

Pour créer des entités, rien de bien compliqué, il suffit d'appeler les constructeurs suivants :

Pour les tiles :

Tile(tm.getResourceByAlias(GRASS), Vec2f(0, 0), Vec2f(120, 60),IntRect(0, 0, 100, 50))

On lui passe un pointeur vers une texture, une position, une taille et les coordonnées de textures.

Pour les mur :

```
Tile *t = new Tile(walls[3]->getFaces()[0]->getMaterial().getTexture(), Vec2f(0, 130),
Vec2f(walls[3]->getSize().x, walls[3]->getSize().y), walls[3]->getFaces()[0]-
>getMaterial().getTexRect());
      w = new Wall(3, t,AmbientLight::getAmbientLight());
```

On lui passe un type qui correspond à ceci :
0 = mur de gauche.
1 = mur de droite.
2 = mur du coin en bas à droite.
3 = mur du coin en haut à gauche.
4 = mur du coin en haut à droite.
5 = mur du bas en bas à gauche.

Plus tard je ferai une enum plutôt que un int. (Ca sera plus facile.:P)

le deuxième paramètre est la tile du mur et le dernier, la lumière utilisée pour généré l'ombre du mur.

Pour les décors :

```
Decor* decor = new Decor(new Tile(tm.getResourceByAlias(HOUSE), Vec2f(0, 200), Vec2f(250,
300), IntRect(0, 0, 250, 300)), AmbientLight::getAmbientLight());
```

On lui passe une tile, et la lumière pour générer l'ombre.

Les animations :

Il suffit de créer plusieurs entités (de n'importe quel type) quis ervirons alors de frames pour l'animation et de les ajouter à l'animation avec la méthode addEntity, une animation prend en paramètre le framerate de l'animation.

```
Anim* fire = new Anim(0.1f);
```

# Chapitre VII : Le système de gestion d'entités 2D et 2.5D et les composants :

Stocker des entités dans le gestionnaire :

Pour stocker des entités dans le gestionnaire il suffit de créer un objet de type map, les gestionnaires d'entités contiennent une grille et les entités sont stockées dans les cellules de cette grille, la grille s'agrandis et se rétréci en fonction de la taille de la map, les cellules de la grille peuvent être définie à l'aide d'une matrice de changement de base ainsi que d'une largeur, d'une hauteur et d'une profondeur pour la 3D.

La matrice de changement de base est utile si les entités ont été rendues dans un autre repère (avec un autre logiciel) que celui utilisé par l'application. (C'est le cas par exemple pour les jeux en 3D isométrique, les objects sont rendus dans un repère en 3D sur une image en 2D et l'application rend ses images dans un repère en 2D)
On peut aussi utiliser cette matrice si l'on veut stocker les entités dans un autre repère que celui utilisé par opengl.

Le gestionnaire d'entité va alors changé de repère lors de la création des cellules de la grille suivant la matrice de changement de base.

Les entités seront donc placée dans une grille qui est contenue dans un repère en 3D isométrique par exemple. (tandit que les entités seront dessinées dans un repère en 2D)

Par défaut la grille contient la matrice de changement de base en 3D iso mais on peut la changer en créant une instance de BaseChangementMatrix et en la passant au gestionnaire d'entité.
La scène peut être rendu sur un ou plusieurs composants de rendu, ces composants choisissent la meilleur option de rendu suivant les fonctionnalités supportée par votre carte graphique. (Shaders, etc...)

Pour créer un gestionnaire d'entité il suffit d'appeler cette fonction :

```
theMap = new Map(&getRenderComponentManager(), "Map test", 100, 50, 0);
```

En lui passant le gestionnaire de composant de rendu utilisé par l'application.

Le gestionnaire de composant contiendra et dessinera tout composants utilisé par l'application (composants de rendu de la scene, guis (menus, ...), shadowMap, LightMap, etc....)
On passe ensuite le nom de la map, ainsi que la largeur, la hauteur et la profondeur des cellules de la grille.

Il n'est pas possible d'utiliser un quadtree ou un bsp-tree simplement avec odfaeg (à moins de créer son propre gestionnaire d'entité en héritant de la classe g2d ::EntityManager pour la simple et bonne raison que odfaeg veut rester dans la simplicité et une grille est plus facile à gérer dans le cadre du

pathfinding et afin de permettre la création d'une map de n'importe quelle taille sans devoir tout régénérer.

Un quadtree nécessite de connaitre la taille de la map à l'avance et un bsp-tree nécessite d'être construit à l'avance avec un éditeur de map.

Il a fallut donc trouvé une alternative et la meilleur que j'ai pu trouvée c'est celle-ci :

Ne travailler que avec des coordonnées entières (et donc pas de coordonnées flottantes qui causes en général des problèmes lors des calculs. (qui doivent être réglé avec un epsilon)

Utiliser plusieurs gestionnaires d'entités avec divers tailles pour les cellules de la grille si les objects sont vraiment très proche les uns des autres, ou bien si le monde est très grand, et stocker ses gestionnaire d'entités dans une classe.

La classe qui permet de stocker ses gestionnaires d'entités s'appelle world et on peut ajouté et récupérer des gestionnaires d'entité comme ceci (il ne faut pas non plus oublier de définir avec quel gestionnaire d'entité on veut travaillé grâce à la méthode setCurrentEntityManager)

```
World::addEntityManager(theMap);
World::setCurrentEntityManager("Map test");
```

Le monde à l'avantage de pouvoir stocker des gestionnaires d'entités pour les entités en 2D et 2.5D mais aussi plus tard pour les entités en 3D.

Ensuite pour ajouté une entité dans le monde il suffit d'appeler la méthode addEntity de la classe World :

```
World::addEntity(myEntity) ;
```

Dessiner des entités :

Pour dessiner des entités ODFAEG il suffit d'appelé la méthode drawOnComponents de la classe world : en lui passant le type d'entité que l'on veut dessiner, comme ceci :

```
World::drawOnComponents("E_TILE", 0);// draw everything here...
World::drawOnComponents("E_WALL+E_DECOR+E_ANIMATION+E_CARACTER", 1);
```

Le 1er paramètre est le type d'entité à dessiner sur le composant, le second paramètre est la position en z du composant sur lequel on veut dessiner la scène.

0 = le composant du fond.
1 = le composant du dessus.

Remettre à jour les entités :

Pour remettre à jour les entités odfaeg possède 2 classes, une classe EntityUpdater qui remet à jour les entités présente de la vue passée à la fenêtre de rendu, et AnimUpdater pour remettre à jour la frame courante des animations, il suffit de créer ces deux objets et de les ajouter à notre monde pour remettre à jour la frame couante : (setInterval permet de définir la fréquence avec laquelle le thread va remettre à jour la frame courante des différentes animations.)

```
eu = new EntitiesUpdater();
```

```
World::addEntitiesUpdater(eu);
au = new AnimUpdater();
au->setInterval(seconds(0.01f));
```

Si votre monde possède des lumières, n'oublier pas d'appeler la méthode computeIntersectionsWithWalls de la classe World , pour rendre les ombres et les lumières, il suffit d'appeler la méthode getShadowMap de la classe World, voici donc par exemple comment tracer une scene dans la méthode onRender :

```
odfaeg::World::drawOnComponents("E_BIGTILE", 0);

odfaeg::World::drawOnComponents("E_WALL+E_DECOR+E_ANIMATION+E_CARACTER", 1);
odfaeg::RenderStates states(sf::BlendMode(sf::BlendMultiply));
odfaeg::g2d::Entity& shadowMap = odfaeg::World::getShadowMap<odfaeg::g2d::Entity>();
odfaeg::World::drawOnComponents(shadowMap, 0, states);
```

Et voici un exemple de code qui met à jour les entités dans la méthode onUpdate() :

# Chapitre VIII : Le système des states.

Créer un state :

Avant de créer un state il faut redéfinir 2 fonctions qui applique et annule le state, pour cela il faut hériter de la classe StateExecutor et redéfinir les 2 méthodes suivantes :

doState(State) et undoState(State), celles-ci prenne en paramètre le state courant.

Pour créer un state il suffit de lui passer un nom et un pointeur sur un objet d'un type dérivé de StateExecutor.

Nous pouvons par la suite, ajouter des paramètres à notre state, et les récupérer dans notre méthode de notre classe dérivée de StateExecuror :

MyState state(« The state », &myStateExecutor) ;

state.addParameter(« ParamName », paramValue) ;

La valeur peut être de n'importe quel type il faudra donc préciser son type lors de la récupération :

ParamType value = state.getParameter(« ParamName ») .getValue<ParamType>;

Nous pouvons associer plusieurs states à un groupe grâce à la classe StateGroup.

Nous pouvons aussi ajouté des groupes de states dans une pile avec la classe StateStack afin d'appliquer les derniers states qui ont été annulés ou bien d'annuler les derniers states qui ont été appliqué.

Ceci est surtout utile dans les applications si par exemple vous souhaiter faire un éditeur de map pour votre jeux.

Mais nous n'en somme pas encore là donc je rentrerai dans les détails en temps voulu.

Chaque entité odfaeg possède un state qui peut prendre un ou plusieurs paramètres personnalisé, ce qui évite de devoir faire trop d'héritage lors de la création d'entités personnalisées.

Les fonctions pour ajouter des nouveau attributs à nos entités s'appellent addAttribute et getAttribute, removeAttribute et changeAttribute et fonctionnent de la même façon que addParameter, getParameter, removeParameter et changeParameter.

Pour modifier appliquer ou annuler un state d'une entité il suffit d'appelé les méthodes interract et uninterract de la classe entity en lui passant un pointeur vers le state executor qui va modifier le state de l'entité.

# Chaptire IX : Les classes utilitaires de odfaeg.

Les variants :

Tout comme boost, odfaeg possède quelque classe utilitaires qui sont utilisées lors du codage du gameplay et l'application de comportements différents suivant les différents type d'entités. (A la diffence prêt qu'elles utilisent le nouveau standart c++14.)

Voici un exemple de code utilisant un variant qui permet d'appliquer deux comportement différent pour une classe A et B.

```
class Chien {
};

class Chat {
};

struct Attaquer : public odfaeg::Visitor<> {
       void operator()(Chat& chat) {
            std::cout<<"griffe!"<<std::endl;
       }
       void operator()(Chien& chien) {
            std::cout<<"mort!"<<std::endl;
       }
};
```

Et dans le main il suffit juste de faire ceci :

```
Chat chat;
Chien chien;
std::vector<odfaeg::Variant<Chien, Chat>> vector;
vector.push_back(odfaeg::Variant<Chien, Chat>(chat));
vector.push_back(odfaeg::Variant<Chien, Chat>(chien));

Attaquer attaquer;

for (unsigned int i = 0; i < vector.size(); i++) {
     odfaeg::apply_visitor(attaquer, vector[i]);
}
```

Une autre méthode :

odfaeg possède une autre méthode mettant permettant de faire cela à l'aide d'une interface pour les diffent type d'entités, et d'appliquer un multiple dispatch dessus comme ceci :

```
struct chat;

struct chien;
```

```
struct animal
        : odfaeg::accept_visitor<animal,chat,chien>
{ };

struct chat
        : odfaeg::acceptable<animal,chat>
{ };

struct chien
        : odfaeg::acceptable<animal,chien>
{ };

struct attaquer

        : odfaeg::dispatchable<attaquer,animal>
{
        void operator()(const chat&, const chat&) const
        { std::cout <<  "griffe - griffe" << std::endl;}
        void operator()(const chat&, const chien&) const
        { std::cout <<  "griffe - mord  " << std::endl;}
        void operator()(const chien&, const chat&) const
        { std::cout <<  "mord - griffe" << std::endl;}
        void operator()(const chien&, const chien&) const
        { std::cout << "mord - mord  " << std::endl;}
};

chat c1;

chien c2;

animal& a1 =c1;

animal& a2 =c2;
attaquer().apply(a1,a1);
attaquer().apply(a1,a2);
attaquer().apply(a2,a1);
attaquer().apply(a2,a2);
```

# Chapitre X : création d'entités personnalisées.

Création d'un personnage animé :

Ici nous allons créer un personnage animé à l'aide de tileset contenant les diverse animation de notre personnages dans les directions suivantes : (nord, nord ouest, nord est, sud, sud ouest, sud est)

Nous devons d'abord hériter de la classe AnimatedEntity car notre entité personnage est animé, nous allons aussi en profiter pour spécifier des attributs à notre personnage  (pour plus tard lorsque on créera le jeux) :

```
#include <vector>
#ifndef CARACTER
#define CARACTER
#include "odfaeg/Math/vec2f.h"
#include "odfaeg/Graphics/2D/anim.h"
#include <string>
#include <SFML/Graphics.hpp>

class Caracter : public odfaeg::g2d::AnimatedEntity {
   public :
      Caracter (std::string factionName, std::string pseudo, std::string sex, std::string
currentMapName, std::string hairColor,
            std::string eyesColor, std::string skinColor, std::string faceType, std::string classs, int
level);
      bool isMovable() const {
         return true;
      }
      bool selectable() const {
         return false;
      }
      bool operator== (Entity& other);
      void addAnimation (odfaeg::g2d::Anim *anim);
      odfaeg::Vec2f getPosition ();
      odfaeg::g2d::Tile& getCurrentTile();
      void setMoving(bool b);
      bool isMoving ();
      void setDir(odfaeg::Vec2f dir);
      odfaeg::Vec2f getDir();
      void setPath(std::vector<odfaeg::Vec2f> path);
      std::vector<odfaeg::Vec2f> getPath();
      int getSpeed();
      void setSpeed(int speed);
      odfaeg::g2d::Anim* getAnimation(unsigned int index);
```

```
        unsigned int getCurrentPathIndex ();
        void setCurrentPathIndex (unsigned int index);
        bool isMonster() {
           return false;
        }
        void setRange(int range);
        int getRange();
        void setLife(int life);
        void setMaxLife(int life);
        int getLife ();
        int getMaxLife();
        int getLevel();
        std::string getClass();
        int getAttack();
        void setAttack(int attack);
        void setAttackSpeed(float attackSpeed);
        float getAttackSpeed();
        void setFightingMode(bool b);
        bool isInFightingMode();
        bool isAttacking ();
        void setAlive(bool b);
        bool isAlive();
        void setAttacking(bool b);
        void setCurrentXp(int xp);
        void setXpReqForNextLevel(int xpReqForNextLevel);
        void setLevel (int level);
        sf::Time getTimeOfLastAttack();
        sf::Time getTimeOfLastHpRegen();
        void attackFocusedMonster();
        void up (int xp);
        int getCurrentXp ();
        int getXpReqForNextLevel ();
        float getRegenHpSpeed();
        void setRegenHpSpeed(float regenHpSpeed);
        int getRegenHpAmount();
        void setRegenHpAmount(int regenHpAmount);
        Entity* getCurrentEntity() const;
        void onDraw(odfaeg::RenderTarget&, odfaeg::RenderStates) const;
        virtual ~Caracter();

    private :
        std::string factionName, pseudo, sex, currentMapName, hairColor, eyesColor, skinColor,
faceType, classs;
        int level, currentPointIndex, attack, speed, range;
        float attackSpeed, regenHpSpeed;
        bool moving, alive;
        odfaeg::Vec2f dir;
        std::vector<odfaeg::Vec2f> path;
        std::vector<odfaeg::g2d::Anim*> anims;
        int currentAnimIndex;
        int life, maxLife, xp, xpReqForNextLevel, regenHpAmount;
        bool attacking, fightingMode;
```

```
        sf::Clock clockAtkSpeed, clockRegenHp;
};
#endif
```

Si les attributs sont communs pour tout les types de personnages de notre jeux, alors, nous ne sommes pas obligé d'utiliser les states, le cas contraire il est fort recommandé d'utilisé des states afin d'évité de devoir faire trop d'héritage et d'interfaces si on a pleins de type de personnages (ou d'armes) différentes et que en plus on définis des règles pour chaque type de personnage.

Les states sont donc un bon moyen d'éviter le bordel lorsque le gameplay de votre jeux devient plus compliqué.

Bref examinons le fichier .cpp

```
#include "caracter.h"
#include <iostream>
using namespace std;

using namespace odfaeg;
using namespace odfaeg::g2d;
Caracter::Caracter (string factionName, string pseudo, string sex, string currentMapName, string
hairColor,
                string eyesColor, string skinColor, string faceType, string classs, int level) :
AnimatedEntity (Vec2f(-50, -25), Vec2f (100, 50), Vec2f(50, 25), "E_CARACTER") {
    currentAnimIndex = 0;
    this->factionName = factionName;
    this->pseudo = pseudo;
    this->sex = sex;
    this->currentMapName = currentMapName;
    this->hairColor = hairColor;
    this->eyesColor = eyesColor;
    this->faceType = faceType;
    this->skinColor = skinColor;
    this->classs = classs;
    this->level = level;
    currentPointIndex = 0;
    speed = 100;
    moving = false;
    dir = Vec2f(0, 1);
    this->life = 100;
    this->maxLife = 100;
    range = 50;
    attackSpeed = 1.f;
    attack = 10;
    fightingMode = attacking = false;
    alive = true;
    xp = 0;
    xpReqForNextLevel = 1500;
    regenHpSpeed = 1.f;
    regenHpAmount = 1;
}
float Caracter::getRegenHpSpeed () {
```

```
    return regenHpSpeed;
}
void Caracter::setRegenHpSpeed(float regenHpSpeed) {
    this->regenHpSpeed = regenHpSpeed;
}
sf::Time Caracter::getTimeOfLastHpRegen() {
    return clockRegenHp.getElapsedTime();
}
void Caracter::setLevel(int level) {
    this->level = level;
}
void Caracter::setCurrentXp(int xp) {
    this->xp = xp;
}
void Caracter::setXpReqForNextLevel(int xpReqForNextLevel) {
    this->xpReqForNextLevel = xpReqForNextLevel;
}
void Caracter::up (int xp) {
    this->xp += xp;
    if (this->xp >= xpReqForNextLevel) {
        level++;
        this->xp = this->xp - xpReqForNextLevel;
        xpReqForNextLevel *= 2;
    }
}
int Caracter::getCurrentXp () {
    return xp;
}
int Caracter::getXpReqForNextLevel () {
    return xpReqForNextLevel;
}
void Caracter::setSpeed(int speed) {
    this->speed = speed;
}
int Caracter::getSpeed() {
    return speed;
}
int Caracter::getRegenHpAmount() {
    return regenHpAmount;
}
void Caracter::setRegenHpAmount(int regenHpAmount) {
    this->regenHpAmount = regenHpAmount;
}
void Caracter::setLife(int life) {
    this->life = life;
    clockRegenHp.restart();
}
int Caracter::getLife() {
    return life;
}
void Caracter::setRange(int range) {
    this->range = range;
```

```
}
int Caracter::getRange() {
    return range;
}
void Caracter::setAttackSpeed (float attackSpeed) {
    this->attackSpeed = attackSpeed;
}
float Caracter::getAttackSpeed () {
    return attackSpeed;
}
void Caracter::setAttacking(bool b) {

    this->attacking = b;
}
void Caracter::setAlive(bool b) {
    alive = b;
}
bool Caracter::isAlive () {
    return alive;
}
bool Caracter::isAttacking() {
    return attacking;
}
void Caracter::setFightingMode(bool b) {
    this->fightingMode = b;
}
bool Caracter::operator== (Entity &other) {
    if (getType() != other.getType())
        return false;
    Caracter& caracter = static_cast<Caracter&>(other);
    if (anims.size() != caracter.anims.size())
        return false;
    for (unsigned int i = 0; i < anims.size(); i++) {
        if (anims[i] != caracter.anims[i])
            return false;
    }
    return true;
}
bool Caracter::isInFightingMode() {
    return fightingMode;
}
void Caracter::setAttack(int attack) {
    this->attack = attack;
}
int Caracter::getAttack() {
    return attack;
}

sf::Time Caracter::getTimeOfLastAttack() {
    return clockAtkSpeed.getElapsedTime();
}
```

```
void Caracter::setDir (Vec2f dir) {

    anims[currentAnimIndex]->setCurrentTile(0);
    float angleRadians = const_cast<Vec2f&>(Vec2f::yAxis).getAngleBetween(dir);
    int angle = Math::toDegrees(angleRadians);
    //Sud
    if (angle >= -10 && angle <= 10)
        currentAnimIndex = 0;
    //Sud ouest
    else if (angle > -80 && angle < -10)
        currentAnimIndex = 3;
    //Ouest
    else if (angle >= -100 && angle <= -80)
        currentAnimIndex = 6;
    //Nord ouest
    else if (angle > -170 && angle < -100)
        currentAnimIndex = 1;
    //Nors est
    else if (angle > 100 && angle < 170)
        currentAnimIndex = 7;
    //Est
    else if (angle >= 80 && angle <= 100)
        currentAnimIndex = 2;
    //Sud est
    else if (angle > 10 && angle < 80)
        currentAnimIndex = 5;
    else
        currentAnimIndex = 4;

    if (attacking)
        currentAnimIndex += 8;
    this->dir = dir;
}

Vec2f Caracter::getDir () {
    return dir;
}
void Caracter::setMoving (bool b) {
    this->moving = b;
    if (moving) {
        anims[currentAnimIndex]->play(true);
    } else {
        anims[currentAnimIndex]->stop();
        anims[currentAnimIndex]->setCurrentTile(0);
    }

}
bool Caracter::isMoving () {
    return moving;
}

Vec2f Caracter::getPosition () {
```

```
    return Vec2f(anims[currentAnimIndex]->getPosition().x, anims[currentAnimIndex]-
>getPosition().y);
}

void Caracter::setPath(vector<Vec2f> path) {
    this->path = path;
}
vector<Vec2f> Caracter::getPath() {
    return path;
}
void Caracter::addAnimation (Anim *anim) {
    addChild(anim);
    anim->setParent(this);
    anims.push_back(anim);
}
Anim* Caracter::getAnimation(unsigned int index) {
    if (index >= 0 && index < anims.size())
        return anims[index];
    return NULL;
}
unsigned int Caracter::getCurrentPathIndex() {
    return currentPointIndex;
}
void Caracter::setCurrentPathIndex (unsigned int currentPointIndex) {
    this->currentPointIndex = currentPointIndex;
}

void Caracter::setMaxLife(int life) {
    this->maxLife = maxLife;
}

int Caracter::getMaxLife() {
    return maxLife;
}
int Caracter::getLevel() {
    return level;
}
string Caracter::getClass () {
    return classs;
}
void Caracter::onDraw(RenderTarget &target, RenderStates states) const {
    target.draw(*getCurrentEntity(), states);
}
Entity* Caracter::getCurrentEntity() const {
    return anims[currentAnimIndex]->getCurrentEntity();
}
Caracter::~Caracter() {

}
```

Les choses importante à remarqué ici est la redéfinition de la méthode onDraw ou l'on dessine l'entité courante de l'animation courante du personnages, on change aussi l'animation courante en

fonction de la direction du personnage.
Nous avons aussi les méthodes addChild et setParent lorsque nous ajoutons une animation pour notre personnage.
Ceci à pour conséquence que lorsqu'on dessinera notre personnage il appelera la méthode onDraw des entités enfants et les méthodes onMove, onRotate et onScale ce qui aura pour conséquence de dessiner aussi les entités enfants de l'entité (c'est à dire l'entité courante de l'animation courante ici) lorsqu'on dessinera l'entité toutes les entités enfants seront transformées par rapport à la transformation de l'entité parent. (c'est à dire toutes les frames des animations du personnages)

Si vous devez remettre des données à jour lorsque le personnage bouge ou bien si vous voulez modifier la transformation vous pouvez redéfinir la méthode onMove en n'oubliant pas d'appelé la méthode move de la classe de base Entity en lui passant la transformation si vous voulez que les transformations se combinent avec celles des entités parent :

```
void Caracter::onMove(Vec3f t) {
        Entity::onMove(t) ;
        //Mise à jour des autres informations. (La physique par exemple)
}
```

Ici nous avons un vector en 3D, il faut savoir que pour les entités ODFAEG en 2.5D la position en z de l'entité est égale à sa position en y, sa transformation en y sera donc la même que celle en z.

Si vous ne voulez pas que la translation se combine, vous pouvez laisser cette méthode vide.

Vous pouvez effectuer la même chose pour la rotation et le changement d'échelle en redéfinissant les méthodes onRotate et onScale.

ODFAEG possède 5 classes de bases dont peuvent hériter toutes vos entités personnalisées :

Entity pour les entités de base qui n'ont pas d'ombre ni d'intersections avec la lumière. (Par exemple les tiles)

Model : pour les entités qui possèdent une ombre et qui peuvent avoir des intersections avec la lumière et une ombre : (Par exemple les murs et les décors)

AnimatedEntity : pour toutes les entités animées.

Shadow : pour toutes les entités qui sont des ombres.

Light : pour toutes les entités qui sont des lumières.

Et enfin pour transformer une entité, nous devons appeler la méthode move de la classe World, ceci aura pour effet de mettre à jour également l'entité dans la grille en fonction de son rectangle englobant :

```
World::moveEntity(caracter, caracter->getDir().x * t, caracter->getDir().y * t);
```

# Chapitre XI : La gestion des collisions.

La méthode simple :

La méthode simple consiste à récupérer la cellule de la grille du monde à un endroit, et de faire en sorte que le joueur ne puisse pas passer sur cette case comme ceci :

World::getGridCellAt(Vec3f(x, y, 0))

Le z vaut toujours zéro ici car pour nos entités 2D nos cellules n'ont pas de profondeur.

Une méthode plus précise :

Faire les collisions sur les cases de la grille peut ne pas être très précis, odfaeg permet également d'associer à chaque entité une hiérarchie de volumes englobants, c'est à dire que tout comme les entités, les volumes englobant peuvent contenir des volumes englobants enfants, le test de collision se fera alors sur les volumes englobant enfant si le volume englobant parent est en collision avec un autre entité du monde.
Voici un exemple de code qui ajoute un volume de collision à une entité, par défaut les entités n'ont pas de volume de collision.

```
 odfaeg::BoundingVolume *bb = new odfaeg::BoundingBox(decor->getGlobalBounds().getPosition().x, decor->getGlobalBounds().getPosition().y + decor->getGlobalBounds().getSize().y * 0.4f, 0,
        decor->getGlobalBounds().getSize().x, decor->getGlobalBounds().getSize().y * 0.25f, 0);
        decor->setCollisionVolume(bb);
```

odfaeg possède actuellement 5 types de volumes englobant pouvant s'imbriquer les uns dans les autres :
BoundingBox : une boîte alignée avec les axes x, y et z.
OrientedBoundingBox : une boîte orientée.
BoundingSphere : une sphere.
BoundingEllipsoid : une éllipsoïde.
BoundingPolygon : un polygône convexe.

Tester les collisions :

afin de tester la collision, il suffit d'appeler la fonction collide de la classe World en lui passant l' entité sur laquelle on veut tester la collision :

```
if (World::collide(caracter)) {
    World::moveEntity(caracter, -caracter->getDir().x * t, -caracter->getDir().y * t);
    getView().move(-caracter->getDir().x * t, -caracter->getDir().y * t, 0);
}
```

Ici, si l'entité est en collision avec une case ou une autre entité dans le monde, on annule le déplacement de l'entité.

Le système de particules :

Celui-ci reprendra celui de la thor library donc il suffit de lire les tutoriels de la librairie thor :

http://www.bromeon.ch/libraries/thor/v2.0/tutorial-particles.html

Vous trouverez en annexe un exemple de code source qui résume tout ce qu'on a vu jusqu'ici et qui montre déjà ce que le framework propose de faire, mais ce n'est que le commancement car je suis encore assez loin comparé à d'autres projets déjà existant.

# Chapitre XII : Le système de sauvegarde de odfaeg.

La sauvegarde d'entités simple :

Pour sérialiser des données il faut deux choses. (une archive qui contiendra le flux dans lequel on mettra les données, ainsi que le flux bien sur)
Pour le moment odfaeg gère 2 types d'archives : une pour l'écriture (OTextArchive) et une pour la lecture. (ITextArchive)
Mais plus tard il est fort possible que j'incorpore d'autres types d'archive comme par exemple les archives binaires ou bien les archives au format xml.

Il faut donc créer un objet de ce type en lui passant un flux, par exemple, un fichier :

```
ofstream ofs("thefile");
OTextArchive oa (ofs);
```

Ensuite pour sérialiser il suffit d'appeler l'opérateur () sur l'archive en lui passant l'objet, ici par exemple nous allons sérialiser un objet de type std::string :

```
std::string text="blablabla";
    std::string text2;
    std::ofstream ofs("FichierDeSerialisation");
    {
        odfaeg::OTextArchive oa(text);
        oa(text);
    }
    ofs.close();
```

Pour la lecture le code est similaire à part que l'on utilise les classe ifstream et ITextArchive :

```
std::ifstream ifs("FichierDeSerialisation");
{
    odfaeg::ITextArchive ia(ifs);
    ia(text2);
}
std::cout<<text2<<std::endl;
ifs.close();
```

Pour sérialiser vos objets personnels, il suffit de redéfinir la méthode serialize dans la classe et de passer à l'archive toutes les variables de la classe.

```
struct A {
    A() {
        var = 10;
```

```
    }
        void foo(int i)
        { std::cout << i; }
        template <typename A>
        void serialize (A & ar) {
        ar(var);
        }
        int var;
};
```

Car c'est cette méthode que odfaeg va rechercher pour sérialiser l'objet dans l'archive, le code pour sérialiser l'objet dans le main est le même que pour les objets de types std::string.

Cette manière de faire est simple mais ne fonctionnera pas dans le cadre des objets polymorphique, en effet, quel donnée de l'objet faut t'il sérialiser, les données de la classe de base seulement, ou celles de la classe dérivée seulement, ou bien les deux ?

Par défaut odfaeg ne sérialize que les données de la classe de base.

Pour lui dire de sérialiser les données de la classe dérivée, il faut redéfinir une autre méthode : la méthode vtserialize.

Pour serializer un objet odfaeg utilise une classe dont il faut hériter dans la classe de base comme ceci :

**La classe de base doit au moins avoir une méthode virtuelle pour pouvoir être considérée comme une classe polymorphique, car, odfaeg fait le test à l'exécution avec typeid, et l'héritage doit être pulibque! (Sinon ça ne fonctionnera pas)**

```
struct B : public Registered<B> {
    B() {
        c = "serialize base";
    }
        virtual void foo()
        { std::cout << 1; }
        virtual void print() {
        std::cout<<c<<std::endl;
        }
        template <typename A>
        void vtserialize (A & ar) {
        ar(c);
        }
        virtual ~B();
        std::string c;
};

B::~B(){}
```

Registered n'est rien d'autre qu'un objet qui va savoir comment sérialiser l'objet polymorphique, on doit lui passer en paramètre template la classe de Base.

Ensuite il ne faut pas oublié de redéfinir la méthode virtuelle temlplate vtserialize dans la classe dérivée.

```
struct C : B {
    C () {
        c = "serialize derived";
    }
    void foo();
    void print () {
        B::print();
        std::cout<<c<<std::endl;
    }
    template <typename A>
        void vtserialize (A & ar) {
        B::vtserialize(ar);
        ar(c);
        }
        std::string c;
};

void C::foo(){ std::cout << 2; }
```

Mais ce n'est pas tout, odfaeg sait comment sérialiser les objets polymorphiques, c'est grâce à la classe Registred héritée par la classe de base qu'il sait qu'il doit aussi sérialiser les attributs de la classe dérivée.

Mais il ne sait pas quelle fonction de la classe dérivé il doit appeler pour sérialiser les données ni quel allocator utiliser pour allouer les pointeurs sur les types de base pour les objets des classes dérivées, pour lui dire, il faut appeler  la macro EXPORT_CLASS_GUID dans la fonction main et lui passer 3 paramètres!

Le 1er paramètre est un id qui doit être unique pour chaque exportation, car on peut très bien avoir plusieurs classes dérivées pour une classe de base, ou bien même avoir des classes dérivées qui héritent de plusieurs classe de base! (héritage multiple)

Le second paramètre est le type de la classe de base dont hérite la classe Dérivée.

Et le dernier paramètre est le type de la classe dérivée à exporter, ce qui donne le code final suivant :

```
struct B : public odfaeg::Registered<Base> {
    B() {
        c = "serialize base";
    }
    virtual void foo()
    { std::cout << 1; }
    virtual void print() {
    std::cout<<c<<std::endl;
    }
    template <typename A>
    void vtserialize (A & ar) {
    ar(c);
    }
    virtual ~B();
    std::string c;
};

B::~B(){}

struct C : B {
    C () {
```

```
        c = "serialize derived";
    }
    void foo();
    void print () {
        B::print();
        std::cout<<c<<std::endl;
    }
    template <typename A>
        void vtserialize (A * ar) {
        B::vtserialize(ar);
        ar(c);
        }
        std::string c;
};
void C::foo(){ std::cout << 2; }
int main() {
    EXPORT_CLASS_GUID(BC, B, C)
    C c;
    B* b = &c;
    B* b2;
    std::ofstream ofs("FichierDeSerialisation");
    odfaeg::OTextArchive oa(ofs);
    oa(b);
    std::ifstream ifs("FichierDeSerialisation");
    odfaeg::ITextArchive ia(ifs);
    ia(b2);
    b2->print();
}
```

# Chapitre XIII : le réseau

Le serveur :

Pour créer une application serveur, il suffit juste de faire une classe qui hérite de la classe application comme vu dans le chapitre II, et on peut redéfinir les méthodes onExec, onInit et onLoad.

OnLoad permet de charger les ressources utilisées par le serveur (en principe un serveur ne charge pas de resource vu que c'est le client qui affiche tout mais il peut par exemple charger les données d'une map dans un fichier ou bien des données sur les joueurs dans une bases de données)

Pour le moment ODFAEG ne permet pas la communication avec un SGDB sans l'utilisation d'une librairie externe mais ça sera à venir dans une version future. (La dernière version très probablement)

Pour pouvoir établir la comunication entre le serveur et le client il faut créer un serveur et le faire écouter sur un port, ceci est possible grace à la classe Network, le moteur réseau de ODFAEG gère 2 protocoles pour la communication : le protocole TCP et le protocole UDP.

Pour lancer le serveur et le faire écouter sur un port tcp et udp, il suffit d'appeler la méthode suivante de la classe Network :

```
Network::startSrv(portTCP, portUDP) ;
```

Cette fonction démarre le serveur et puis rend la main ce qui nous permet de passer à la méthode exec dans le main :

```
Application app ;
Network::startSrv(portTCP, portUDP) ;
return app.exec() ;
```

Récupérer une requête envoyée par un client :

Pour récupérer une requête envoyée par un client, il suffit d'appeler la méthode getLastRequest de la classe network, cette méthode peut prendre en paramètre un object de type user, la classe user contient des informations qui identifie le client sur le réseau, on peut donc récupérer les informations sur le client qui a envoyé la requête et renvoyer une réponse au client en appelant les méthode sendTCPPacket et sendUDPPacket de la classe client.

La classe user peut être redéfinie si nous avons besoin de stocker d'autre informations comme par exemple un mot de passe et un nom d'utilisateur si il doit y avoir un système d'authentification.

La classe Network possède aussi une méthode sendTcpPacket et sendUdpPacket, dans ce cas, le packet sera envoyé a tout les utilisateurs connecté sur le réseau.
On peut aussi récupérer le ping d'une utilisateur.
En mode TCP, on peut envoyer deux types de packets, des bêtes paquets de type sf::packet ou bien, des packets chiffrés de type odfaeg::SymEncPacket si le client utilise une connection SSL. (pour le savoir il suffit d'appeler la méthode isUsingSecuredConnection de la classe user.
**Attention : en mode udp les packets ne sont pas chiffrés.**

Le client :

Pour connecter un client à un serveur, il suffit d'appeler en plus cette méthode dans le main en lui passant le port TCP du serveur, le port UDP du serveur, et l'adresse ip du serveur.

Network::startCli(portTCP, portUDP, ipAddress)

Pour récupérer les réponses reçus par le serveur il suffit d'appeler la méthode getLastResponse de la classe Network. (ou bien getResponse(« Tag »)) pour récupérer une réponse précise précédée par un tag au choix.

Nous pouvons aussi appeler la méthode waitForLastResponse avec un timeout.

Par défault le client utilise une connection chiffrée en mode TCP mais si l'on ne veut pas utiliser de connection chiffrée on peut passer false en 4ème paramètre à la méthode startCli.