Class Diagram & Modules Subdivision

# Modules Subdivision

Game

This is the main module of the application. It contains the whole game and is used to initialize, loop and finalize the application.

Graphics

This module will deal with the Graphics Engine (Ogre), and will be in charge of initialize, configure and finalize it. Other graphic engine related specialized managers will be found here too (i.e. LightsManager)

Input

The Input module deals with User Input. It is in charge of receiving Input events from input devices (mouse, keyboard, joystick(s)) and send them to all registered listeners. The InputManager is the sub-module that manages all this. A class that wants to receive user input events will have to implement the InputListener interface and register itself to the InputManager.

Audio

The Audio manager is the module that abstracts the audio layer.

Data

The Data manager will be in charge of loading/saving resources from disk. It will offer a common interface to data and resources.

Physics

The physics engine will be implemented and controlled by the Physics module. A PhysicsManager will offer a standard interface to the underlying physics engine. A ParticleSystemsManager will manage all particle systems we will add.

Camera

This module will deal with cameras. Since a lot of different cameras will be handled in the game, a Camera Manager will be used to manage them all.

GUI

The GUI module will help dealing with the GUI. There will be a GuiManager that will manager all GuiScene (s). A GuiScene is the collection of GuiWidget for a game state.

Logic

The Logic module wil deal with game’s logic, enemies AI, and in a general way will manage scripts.

Entity

The Entity module will manage all entities (players, enemies, items) in the game.

States

States are managed by a StatesManager, that will listen to user input, and accordingly to game’s logic will change the current state.

Events

The EventsManager is the class that manages the events engine, which is the inter-communication system between different entities/manages in the game. Classes that want to be able to receive/raise events will have to implement the EventsInterface, and subscribe to the EventManager in order to be able to communicate, using Event(s) and EventsHandler(s) classes.

## Game Loop

The Big Picture

Initialization

The Loop

Finalize

# Class Diagram

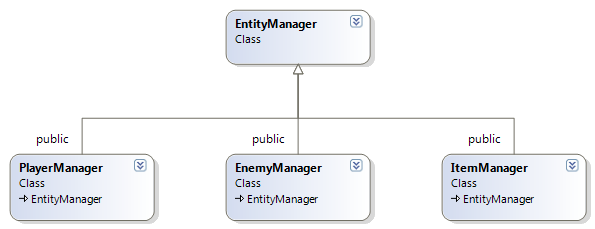
## The StatesManager and the game states



## The EntityManager and its subclasses

The Entity Manager is the base class used to manage all the ‘entities’ in the game. With the term entity we refer to all the things a player can interact with, and that can change their internal status, and that can receive user input, or follow some kind of logic, physics, AI. For instance, a Wyvern is an entity (controlled by the player, subject to physics, can change its status…), just like an enemy (soldier, wizard) or an item (power up).

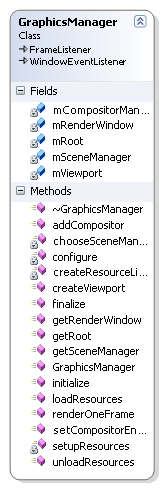
An object in the scene is an item if and only if the player can interact with it (a rock that can be destroyed, a house that can be destroyed, a tree that can be destroyed… well the game is about destroying stuff). If an object is just part of the scenery, it does not need to be managed by this kind of manager (the ground, the sky, background hills and mountains).



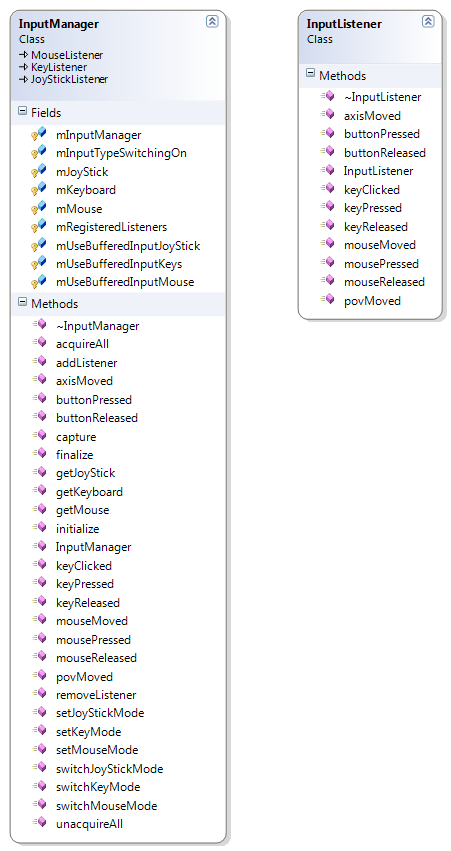
## GraphicsManager

The GraphicsManager is layer between the game and Ogre. It manages Ogre’s initialization, resources, and finalization. It creates Ogre’s Root, SceneManager, RenderWindow and returns them to those who need them. This class will grow bigger along with the application, and will for sure export more features.

Right now, this is its structure:



## InputManager



## The SceneLoader and the XML Parser



## Lua Interfaces

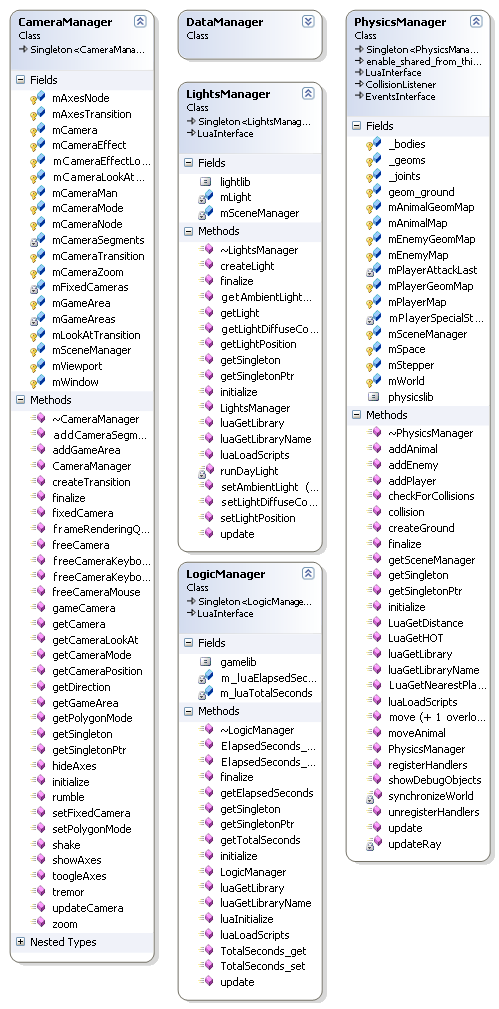
## 

Almost all the Manager classes in the game export methods that can be used from Lua scripts. These classes are the most suitable for this purpose since they have access to all the elements of the game from different points of view. Since Lua exported methods are static, these classes implement the singleton pattern in order to be able to access to their running instances when needed.

## The GUI and its widgets



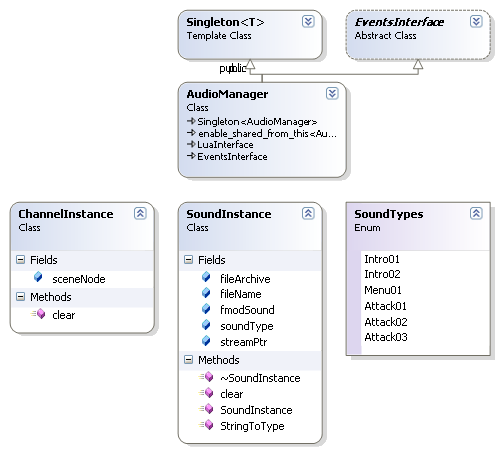
## Other Managers



## Object Interfaces

Enemies and players implement common interfaces for Logic, Graphics and Physics. In order to have some Logic (i.e. AI, scripting) an object needs to implement the LogicInterface. Same is valid for PhysicsInterface and EntityInterface. The previous scheme shows that e.g. the Enemy implements the PhysicsInterface, and is through this interface that the PhysicsManager controls it. The Enemy also implements a LogicInterface, that allows him to call and deal with Lua scripts. From lua scripts it is possible to call sunctions exported by the PhysicsManager or any other manager that implements the LuaInterface.

## Audio System

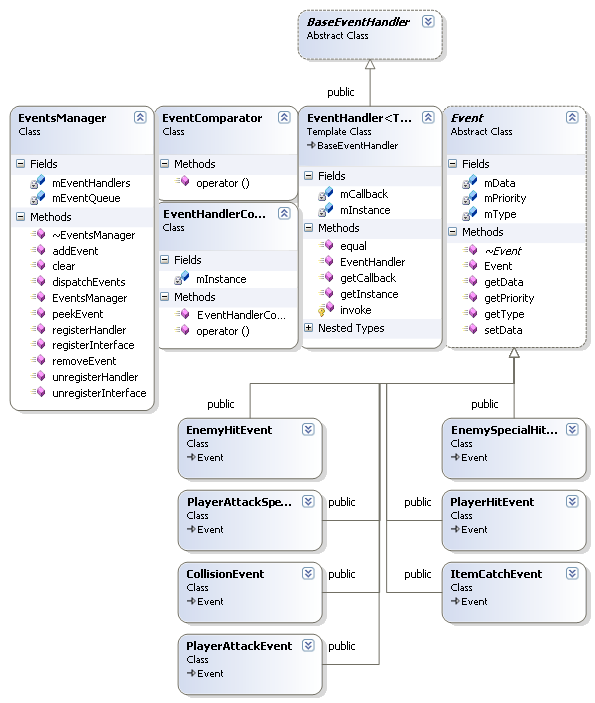


The Audio system is based on FMOD, and relies on a couple of classes. The main class is the AudioManager. Its role is to manager all the sounds, react to the events, and export basic functionalities like playSound(), stopSound(), createSound() …

Every sound will have its own SoundInstance, and every time a sound has to be played, a specific ChannelInstance will be assigned to the SceneNode where the sound has to be played in the 3D world.

Like any other manager, it is event enabled, and can be controlled through LUA scripts.

## Event System



The Events system provides the event feature in the game. Events are the way different modules (e.g. Managers) communicate each other. Core of the system is the EventsManager class. This class manages the events queue and all the registered listeners/handlers.

A class that implements the EventsInterface can be registered to the manager through the registerInterface() method. This method will register the class as listener and sender.

An EventsInterface will register one or more methods that the EventsManager will call when a give envent happens.

Whenever an EventsInterface will raise an event (using the raiseEvent() method), the event itself will be added to the events priority queue, and the manager will dispatch it to all the handlers that have been registered for this event type.

This approach helps to have independent modules, and allows a more generic way of dealing with events that can happen during the game.