ULTIMAADE

Documentation Technique

Ultimaade Team ultimaade@gmail.com

Abstract : `Ultimaade` *est un logiciel libre et multiplateforme, de création d'histoires et de support multimédia.*

La principale fonction du logiciel est l'édition de `romans visuels` *(*`visual novels` *en anglais), type particulier de jeux vidéo très répandu au Japon et peu connu en occident.*

Ultimaade permettra également de faire des présentations/montages type vidéo ou texte, offrant ainsi aux enseignants par exemple, de créer du contenu pédagogique selon leurs domaines de compétence.

Dans un sens plus large le logiciel est destinée à tout type d'utilisateur, désirant créer une application multimédia autonome et multiplateforme : technicien ou non, enfant ou adulte.

Ce document constitue la documentation technique du projet.

Il décrit les objets et méthodes du diagramme des classes et est appelé à évoluer pour prendre en compte les nouvelles contraintes de conception et de développement.

Il suit également la logique du document conception architecturale(AA2).

Les plugins et objets sont au centre de toute la logique du logiciel, et documentés dans l'API Ultimaade.

Le document couvre les rubriques suivantes : le moteur (Core), les plugins Ultimaade, les objets Ultimaade, la gestion du système de fichier, la gestion des projets Ultimaade, les préférences de configurations, les niveaux d'utilisation et la description de l'API.

`Le texte en violet indique un nom de classe ou une partie du diagramme des classes.`

Informations sur le document

- Date de mise à jour : 17/05/2012
- Objet : Compléments sur le gestionnaire du système de fichiers, et le format des projets
- Intitulé du Document : Documentation technique
- Version : 1.3
- Projet : Ultimaade
- Auteur : Groupe Ultimaade
- Site : http ://eip.epitech.eu/2013/ultimaade

`Historique des révisions :`

Réf	Objet	Date	Rédaction	Approbation
Version 1.1	Première version du document, reprenant une description technique des diagrammes des classes	15/12/2011	Groupe Ultimaade	Laboratoire EIP
Version 1.2	Deuxième version du document, décrivant la programmation du moteur du projet, et les modules implémentant l'API	23/03/2012	Groupe Ultimaade	Laboratoire EIP
Version 1.3	Troisième version du document, apportant des compléments sur la la gestion du système de fichier, et la structure des projets sous Ultimaade	17/05/2012	Groupe Ultimaade	Laboratoire EIP

Table des matières

Chapitre I

Contexte du projet

Suite à l'évolution rapide des technologies, EPITECH membre du groupe IONIS, a entrepris de former des futurs experts en technologies de l'information.

Dans cette optique l'école a mis sur pied une stratégie consistant pour les étudiants de chaque promotion, en la conception et la réalisation d'un projet innovant, et répondant à une problématique concrète de l'heure. Ce concept s'appelle EIP (Epitech Innovative Project).

Pour les étudiants, l'EIP réalisé représente le projet de fin d'étude, et conditionne le diplôme d'expert décerné par l'école.

C'est dans ce cadre que le groupe ULTIMAADE s'est réuni autour du projet qui fait l'objet du présent document d'architecture.

Chapitre II

Le Coeur et les modules de l'application

Le coeur du logiciel est le centre névralgique d'Ultimaade. C'est le seul élément à être indispensable au logiciel.

En effet c'est lui qui se permet de gérer les plugins et tous les autres comportements de base communs à l'ensemble des parties de l'application. Ces comportements sont détaillés dans les paragraphes ci-dessous.

II.1 La classe Core

La classe `Core` est simplement le contenant des différents managers du logiciel. Elle n'a pas de comportement associé.

Pour une plus grande flexibilité elle implémente un patron de singleton car on doit pouvoir y accéder d'une part depuis l'interface graphique `(IHM)` et d'autre part depuis les plugins de l'application.

II.2 Gestion des plugins

C'est le `Core` qui à la responsabilité des plugins, il gère les plugins en cours d'éxécution, en pause (inactifs), non ouverts, et désactivés.

`PLuginManager`, `PluginDescriptor`, `IPlugin`, `ePluginState` et `PreferenceManager` sont les classes conçues pour répondre à cette contrainte.

Un plugin existe uniquement sous forme de bibliothèque et se trouve dans un répertoire de l'application appelé "Plugins".

C'est à partir de ce répertoire que les plugins seront chargés par défaut, l'utilisateur pourra spécifier un autre répertoire de son choix lors de l'installation du logiciel.

La classe `PreferenceManager` ne sera pas détaillée dans cette partie.

II.2.1 ePluginState

`ePluginState` définit une énumération contenant les états possibles d'un plugin :

- `opened` est la valeur de l'état lorsque le plugin est au premier plan de l'application, l'utilisateur peut travailler dedans et la bibliothèque du plugin est ouverte.
- `paused` est la valeur de l'état lorsque le plugin est mis en pause après un changement de plugin par exemple, le travail en cours n'est pas perdu et la bibliothèque du plugin est ouverte aussi. Mais le plugin est simplement inactif.
- `closed` est l'état par défaut des plugins, lorsqu'ils ne sont pas ou plus ouverts, la bibliothèque du plugin est donc fermée.
- `desactivated` est un état dans lequel le plugin est complètement désactivé, les objets associés au plugin ne sont plus pris en charge. La bibliothèque du plugin est également fermée dans ce cas.

II.2.2 PluginManager

Le `PluginManager` contient toutes les informations des plugins.

C'est à travers cette classe que les plugins sont gérés. Elle possèdera donc un conteneur de plugins avec leurs différents états.

Un `PluginManager` est instancié lors du lancement de l'application. Il n'y aura qu'une seule instance de cette classe tout au long de l'exécution de l'application, qui sera détruite avec la fermeture du programme.

Cette classe utilise des fonctions prédéfinies de la librairie Qt pour ouvrir et fermer les bibliothèques.

Grâce au `PreferenceManager`, la liste des plugins est récupérée dans son `constructeur`.

Le répertoire des plugins n'est pas inspecté à chaque lancement du programme. Cette liste des plugins est stockée dans l'attribut `pluginList` de la classe.

Le `destructeur()` prend bien soin de refermer toutes les bibliothèques de plugins ouvertes.

La fonction membre `updatePLuginList()` va scanner le répertoire des plugins pour ouvrir et vérifier tous les plugins de l'application (même ceux désactivés) afin de mettre à jour les noms et les descriptions.

Par la même occasion elle vérifie la cohérence entre la liste des plugins de la classe et les binaires correspondant dans le répertoire de "Plugin".

En cas d'absence dans ce répertoire d'un plugin de la liste, ce dernier est supprimé par la fonction `updatePLuginList()` de la `pluginList` et du `PreferenceManager`.

Enfin cette fonction vérifie si de nouveaux plugins ne sont pas apparus dans le répertoire et les inclut dans le programme.

La fonction membre `addPLugin()` ajoute un plugin à la liste du `PluginManager`.

Elle vérifie la présence du plugin dans le répertoire avant de l'ajouter à la liste. Et en cas d'absence affiche une erreur.

La fonction membre `removePlugin()` supprime un plugin de la liste. Elle sera par exemple appelée pour la désactivation d'un module depuis la GUI.

La fonction `openPlugin()` sert à mettre au premier plan un plugin, c'est-à-dire lorsque l'utilisateur veut travailler dans le module correspondant.

Le plugin qui était auparavant ouvert est alors mis en pause.

Un plugin ouvert passe soit de l'état `paused` à l'état `opened` s'il était en pause, soit de l'état `closed` à `opened` si il n'était pas ouvert.

La fonction membre `closePlugin()` du `PluginManager` ferme simplement le plugin spécifié et la bibliothèque correspondante.

Le plugin passe à l'état `opened` `si et seulement si` il était à l'état `paused`, avant de passer à l'état `closed`.

La fonction `activateOPlugin()` fait passer un plugin de l'état `desactivated` à l'état `closed`.

Les objets associés au plugin sont donc ouverts à ce moment et deviennent utilisables dans le programme, mais le pluging ne peut encore être utilisé (bibliothèques non ouvertes), jusqu'à ce que ce dernier soit fermé.

La fonction membre `desactivate()` fait passer le plugin à l'état `desactivated`.

Si le plugin était en pause (`paused`) alors il est d'abord ouvert (`opened`), et s'il était ouvert il est fermé (`closed`).

Ensuite toutes les bibliothèques associées au plugin sont fermées, elles deviennent alors inutilisables.

La fonction `getPluginList()` est un accésseur sur l'attribut `pluginList` permettant d'obtenir la liste des plugins durant l'exécution du programme.

II.2.3 PluginDescriptor

La classe `PluginDescriptor` décrit un plugin. La classe `PluginListManager` contient un attribut de ce type.

Il s'agit uniquement des informations d'un plugin particulier.

Ces informations sont ensuite reportées dans l'interface graphique de l'application pour l'utilisateur.

- L'attribut `description` de cette classe est une description succincte de ce que fait le plugin. C'est un texte montré à l'utilisateur lorsqu'il cherche les détails d'un plugin.
- L'attribut `name` est simplement le nom complet du plugin montré à l'utilisateur et présent dans les différentes interfaces graphique du logiciel.
- L'attribut `path` est le chemin et le nom de la bibliothèque du plugin dans le système de fichier sur lequel est installé le logiciel.
- L'attribut `plugin` est un pointeur sur la classe `IPlugin`. Lorsque le plugin est dans l'état `opened` ou `paused` alors la bibliothèque de ce plugin est ouverte, c'est donc dans cet attribut qu'est stocké le pointeur sur cette bibliothèque ouverte. Dans les autres cas le pointeur doit simplement être mis à la valeur NULL.
- L'attribut `state` de la classe tient à jour l'état du plugin, il peut prendre une des valeurs de l'énumération `ePluguinState`.

II.2.4 La classe IPlugin

`IPlugin` est l'interface de l'API des plugins ultimaade.

Elle représente ce qui est possible de faire avec un plugin. Tous les plugins devront implémenter cette interface.

La fonction membre `getDescription()` de la classe devra renvoyer une brève description sous forme de texte du plugin.

La fonction `getIrisMenu()` renvoie ce que le designer du plugin désire mettre dans le menu Iris réservé au plugin.

La fonction `getName()` retourne le nom du plugin.

La fonction `IrisMenuButtonClicked()` sera appelé par la `GUI` lorsque la souris clique sur une entrée du menu Iris réservé au plugin.

La fonction `onExit()` est appelée du `Core` lorsque le plugin doit se fermer et ainsi passer à l'état `closed`.

La fonction membre `onLoad()` est appelée du `Core` juste au moment où le plugin passe à l'état `opened`opened (la blibiothèque du plugin est ouverte).

La fonction membre `pause()` de cette classe est appelée lorsque le plugin doit être mis en pause.

La fonction `resume()` est appelée du Core lorsque le plugin sort de l'état `paused`, pour passer à l'état `opened`.

Exemple de code d'une bibliothèque implémentant l'API des plugins :

```
1  extern "C"
2  {
3    IPlugin * getPluginInst();
4  }
5
6  class DummyPlugin : public IPlugin
7  {
8    public:
9
10   DummyPlugin()
11   {
12     //Constructeur du DummyPlugin
13   }
14
15   virtual ~DummyPlugin()
16   {
17     //Destruction du DummyPlugin
18   }
19
20   virtual QStringgetDescription()
21   {
22     return ("Ce plugin permet de faire des test");
23   }
24
25   virtual QStringgetName()
26   {
27     return ("Dummy");
28   }
29
30   virtual QList<QString> *getIrisMenu()
31   {
32     return new QList<QString>();
33   }
```

```
34
35    virtual intIrisMenuButtonClicked()
36    {
37      return (0);
38    }
39
40    virtual intonExit()
41    {
42      return (0);
43    }
44
45    virtual intonLoad()
46    {
47      return (0);
48    }
49
50    virtual intpause()
51    {
52      return (0);
53    }
54
55    virtual intresume()
56    {
57      return (0);
58    }
59  }
60
61  IPlugin * getPluginInst()
62  {
63    return new DummyPlugin();
64  }
```

Ces exemples seront repris plus en détail dans la documentation de l'API.

II.3 Gestion des bibliothèques objets Ultimaade

Les objets Ultimaade sont tout ce qui vont être manipulés par les plugins.

Ils pourront être utilisés par n'importe quel autre module de l'application même si le plugin duquel il est issue n'est pas ouvert.

En effet chaque objet se trouve dans sa propre bibliothèque, donc il est utilisable indépendamment d'un quelconque plugin.

Un plugin Ultimaade peut manipuler ou non des objets. Les plugins manipulant les objets sont créés avec les types concrets d'objets qu'ils manipulent.

Ainsi les objets tout comme les plugins, implémentent une interface de programmation, et sont créés avec les plugins qui les utilisent.

De cette façon les objets Ultimaade rentrent dans la définition de l'API de plugins et en font partie.

La classe `ObjectManager` gère les bibliothèques d'objet au même titre que la classe `PluginManager` gère les plugins.

Cependant il peut y avoir plusieurs instances d'un Object dans le programme principal. Tandis qu'un plugin n'est instancié qu'une fois, et reste en mémoire jusqu'à ce que la bibliothèque correspondant soit fermée.

Dans le programme principal (le `Core`), on n'a aucune connaissance de ce qu'est un objet.

Cela relève de la vocation des seuls plugins qui manipulent les objets.

Une relation de 0 à plusieurs est définie de `Plugin` vers `Objet`. Et une relation de 1 à 1 de `Objet` vers `Plugin`.

Ainsi un Objet est associé à un et un seul plugin, le plugin pour lequel il a été créé.

Cependant un objet d'un plugin peut agir sur des objets d'un autre plugin. Le mécanisme d'action est décrit plus bas dans la classe `Action`.

II.3.1 objAssociationTable

L'`ObjectManager`, possède un attribut `objAssociationTable` (table d'association).

Cet attribut contient les correspondances entre les bibliothèques d'objets et les types de fichiers correspondants.

Pour chaque objet est associé un ou plusieurs types de fichiers.

Ainsi lorsqu'un plugin est chargé dans le programme principale, il déclare tous les objets qui lui sont associés dans l'`ObjetManager`.

Par la même occasion il ajoute à la table la table d'association un couple d'entrée (objet, liste des types de fichiers associés) pour chaque objet déclaré.

La table d'association est utilisée chaque fois qu'un objet doit être mappé en mémoire avec l'instance de la classe correspondante.

En effet, lorsque l'utilisateur veut ouvrir un fichier correspondant à un objet sur disque, la bibliothèque objet correspondante doit être chargée. Celle-ci permettra de manipuler l'objet en question.

Le plugin dans lequel on travaille se servira de la table d'association pour déterminer si le type du fichier ciblé correspond bien à un des types des objets qu'il reconnait.

Si c'est le cas, la bibliothèque correspondante est ouverte, si non aucune bibliothèque objet n'est ouverte et une erreur est renvoyée à l'utilisateur.

La fonction membre `addObject()` ajoute une nouvelle bibliothèque d'objet au gestionnaire d'objets.

La référence à la bibliothèque est ajoutée, mais la bibliothèque en question ne sera ouverte que lorsque le plugin associé en aura besoin.

La fonction `removeObject()` va supprimer complètement une bibliothèque objet du gestionnaire d'objet, et ferme la bibliothèque associé si elle était ouverte.

Elle nettoie également la table d'association pour supprimer le couple d'entrée correspondant à l'objet.

`Il serait dangereux de supprimer un objet dont les instances sont en cours d'utilisation.`

La fonction `getObjInstance()` prend en paramètre un fichier et renvoie un pointeur sur objet. L'objet correspondant au type de fichier.

Cette fonction est celle utilisé par le plugin de l'objet correspondant au fichier lors de l'accès au fichier sur disque.

La libératiion de l'espace mémoire allouée au pointeur sur l'objet est de la responsabilité du plugin.

II.3.2 Object

La classe `Object` représente un objet Ultimaade en mémoire.

Les instances de cette classe sont créées grâce à la fonction `getObjInstance()` de la classe ObjectManager.

Les bibliothèques d'objet implémentent des classes dérivées de cette classe.

Le `constructeur` de cette classe prend en paramètre un pointeur fichier soit rien du tout.

Le `destructeur` prend soin de fermer le ou les fichiers associés à cet objet.

L'attribut `actionNb` est le nombre d'actions que contient l'attribut `actions`.

C'est le nombre d'actions qu'il est possible de faire avec cet objet. Il est peut être utilisé afin d'éviter un dépassement du conteneur lors de l'accès aux actions.

L'attribut `actions` est un conteneur d'instances de la classe `Action`.

Ce sont les actions qu'il est possible de faire avec cet objet.

Ces actions sont accessibles via l'opérateur « [] » de la classe `Action` (voir détail plus bas).

L'attribut `associatedType` contient les types de fichiers compatible avec l'objet.

L'attribut `description` est une brève description de l'objet.

L'attribut `name` est le nom de l'objet dans un format clair et lisible par un humain.

L'attribut `path` est le chemin du fichier encapsulé dans l'instance de l'objet.

L'attribut `representation` est un pointeur sur `QWidget` représentant graphiquement l'objet dans l'IHM.

Le pointeur peut être nul dans le cas où l'objet n'est pas destiné à être affiché.

La fonction membre `operator[]()` renvoie une référence sur Action.

Elle renvoie l'action correspondante à l'index dans le conteneur `actions`. Elle vérifie également que l'index se trouve bien dans les bornes du conteneur.

Exemple de code d'une bibliothèque implémentant l'API des objet Ultimaade :

```
1  extern "C"
2  {
3    Object * getObjectInst();
4  }
5  Action NothingAction(&DummyObject::nothingAction,
6  "Action qui ne fait rien",
7  "Nothing");
8
9  Action DoubleNothingAction(&DummyObject::doubleNothingAction,
```

```
10  "Action qui ne fait rien mais agit sur plusieurs objets",
11  "Double Nothing");
12
13  class DummyObject : public Object
14  {
15    public:
16    DummyObject()
17    : Object(2,
18    { NothingAction(), DoubleNothingAction() },
19    { ".dummy" },
20    "Objet de test",
21    "Dummy",
22    "",
23    NULL)
24    {
25      //Constructeur par defaut du DummyObject
26    }
27
28    DummyObject(File file)
29    : Object(2,
30    { NothingAction(), DoubleNothingAction() },
31    { ".dummy" },
32    "Autre objet de test",
33    "Dummy",
34    file,
35    NULL)
36    {
37      //Constructeur du DummyObject avec un fichier
38    }
39
40    virtual ~DummyObject()
41    {
42      //Destruction du DummyObject
43    }
44
45    // Pas besoin de redefinir l'operateur []
46
47    //Definition des actions
48    private:
49    intnothingAction(Object *)
50    {
51      return (0);
52    }
53
54    intdoubleNothingAction(Object * dude)
55    {
56      if (dude)
57      return (1);
58      return (0);
59    }
60  }
```

```
61
62  Object * getObjectInst()
63  {
64    return new DummyObject();
65  }
```

Ces exemples seront repris plus en détail dans la documentation de l'API.

II.3.3 Action

La classe `Action` représente une action qu'il est possible de faire avec un objet Ultimaade.

Comme spécifié dans le diagramme des classes, un Objet est une composition d'action.

Ce qui signifie que la libération ou la mort de l'objet entraîne celle de toutes les actions qu'il permet d'effectuer.

Le `constructeur` prend en paramètre un pointeur sur `Object`.

En effet une action est associée à une méthode d'un objet. Plus exactement, une action permet de stocker une méthode d'un objet.

Ceci permet au développeur d'exposer facilement les méthodes de son objet qu'il veut publier, et de les rendre accessibles.

L'attribut `action` est un pointeur sur membre de la classe `Object`.

La fonction pointée contient le code de l'action elle-même, et se trouve dans la classe dérivée d'`Object` associée à cette action.

Il prend en paramètre un pointeur sur objet, permettant d'inclure un autre objet dans l'action. Le pointeur est nul lorsque l'action s'applique uniquement à l'objet.

L'attribut `description` est une brève description de ce que fait l'action.

L'attribut `name` est le nom donné à cette action.

La fonction `operator()` permet d'utiliser une action comme un fonteur `(functor`.

Cette fonction exécute le pointeur sur fonction `action`.

Plus précisément les actions permettent d'agir sur les objets d'un plugin et d'interagir entre les objets manipulables par le plugin.

La définition des actions dans un objet dépend strictement des méthodes que le développeur souhaite exposer.

II.4 Gestion du système de fichiers, des collections initiales et du presse-papier

II.4.1 Le système de fichiers

L'installation d'Ultimaade suppose l'existence d'un système de fichiers natif atteignable sur le système d'exploitation hôte. Le logiciel étant portable, il s'intègrera donc aux différents systèmes de fichiers implémentés par les systèmes d'exploitation supportés.

L'implémentation du gestionnaire de système de fichier sous Ultimaade se basera sur la bibliothèque `QFile` de `QT`.

Le gestionaire de fichier Ultimaade prendra en compte les fichiers locaux et distant (via le réseaux).

II.4.2 Les collections initiales

Les collections initiales sont des objets par défaut du programme. Cela veut dire qu'on fournira à l'utilisateur une banque d'images, sons, videos, exemples (samples), etc, lui permettant de commencer un projet à partir d'objets existants.

Ces collections seront inaltérables, et si l'utilisateur modifie un de ces objets, une copie est faite automatiquement.

Les collections se trouveront dans le répertoire d'installation d'Ultimaade, dans un sous-répertoire appelé : `Collections`. Dans ce répertoire il y aura donc un sous-répertoire pour chaque type d'objet :

- Images : pour les images
- Sons : pour les musiques et sons
- Vidéos : pour les vidéos
- Samples : pour exemples de projets complets

Ces fichiers objets devront être accessibles via le menu Iris de l'interface graphique du logiciel tel que décrit dans le document utilisateur.

II.4.3 Presse-Papier

Une implémentation du presse-papier apportera une meilleure ergonomie au logiciel. Le presse-papier aura permettra les actions :

- copier
- couper
- coller

Ces opérations pourront être réalisées sur n'importe quel objet durant l'exécution du programme.

- La combinaison des touches `Ctrl + C` effectuera une opération de copie de l'objet.
- La combinaison des touches `Ctrl + X` effectuera une opération de déplacement (couper), l'objet sera déplacé.
- La combinaison des touches `Ctrl + V` effectuera une opération d'écriture/insertion de l'objet dans le plugin cible. Le dernier objet copier dans le presse papier sera donc collé.

Il est possible de copier/couper plusieurs objets les uns après les autres, ils seront sauvegardés grâce à la classe `ClipBoard`. Seule la classe `ClipBoard` intervient dans la gestion du presse-papier.

L'attribut `objList` contient la liste des objets copiés ou coupés (déplacés). Cette liste pourra être consultée via l'interface graphique.

Elle se situera dans le menu Iris Bibliothèque. En actionnant un bouton de cette Iris on pourra la faire passer à chacun des modes : bibliothèque standard -> bibliothèque de collection -> bibliothèque du presse-papier.

Dans le dernier mode sont affichés les objets du presse-papier. A partir de ce menu, il est donc possible de copier un objet : en le déplaçant dans l'Iris correspondant, ou décoller un objet : en le déplaçant de l'Iris vers un plugin.

La fonction `ctrlC()` sera raccordée à l'évènement associé aux touches `Ctrl + C`. Elle correspondra à une opération de copie. L'objet passé en paramètre est copié et mis dans la liste du presse-papier : `objList`.

La fonction `ctrlV()` sera connectée à un l'évènement associé aux touches `Ctrl + V` et correspondra à une opération d'écriture (coller). L'objet passé en paramètre est simplement retiré de la liste `objList` et transmis au plugin qui le demande.

`Le pointeur en mémoire sur l'objet n'est pas libéré.`

La fonction `ctrlX()` sera raccordée à un appui sur les touches `Ctrl + X`, et correspondra à une opération de déplacement (couper). L'objet passé en paramètre est copié puis ajouté dans la liste du presse-papier `objList`. Cette fois le pointeur en mémoire est libéré.

La fonction `copyObject()` effectuera l'opération de copie et d'ajout dans la liste du presse-papier. Elle prend un objet en paramètre.

La fonction `pasteObject()` colle et transmet un objet au plugin cible. Cette fonction prend en paramètre une position dans la liste `objList` et renvoie le pointeur correspondant, après l'avoir retiré de la liste.

II.5 Structure des projets sous Ultimaade

Un projet Ultimaade est composé d'objet et existera sur le système de fichiers dans un format particulier. Pour en faire une illustration concrète, nous prendrons dans cette partie le cas particulier des projets du type `roman visuel`.

II.5.1 Anatomie d'un roman visuel

Un roman visuel sera constitué d'une part, de composants visibles, il s'agit de ce que le lecteur du roman voit. Ce sont les dessins/sprites et animations.

De l'autre côté il y a les composants qui sont sous le capot, c'est à dire toute la partie qui anime et fait défiler les images. Il s'agit là des composants invisibles, et c'est à ceux là que nous nous intéressons dans cette partie.

Car c'est de ce côté-là que les jeux vidéo et les visual novels (qui sont la vocation même d'ultimaade) ont le plus de points communs, puisqu'en réalité ils partagent la même technologie.

Le schéma ci-dessus illustre et décrit la structure minimale d'un visual novel type créé avec `Ultimaade`

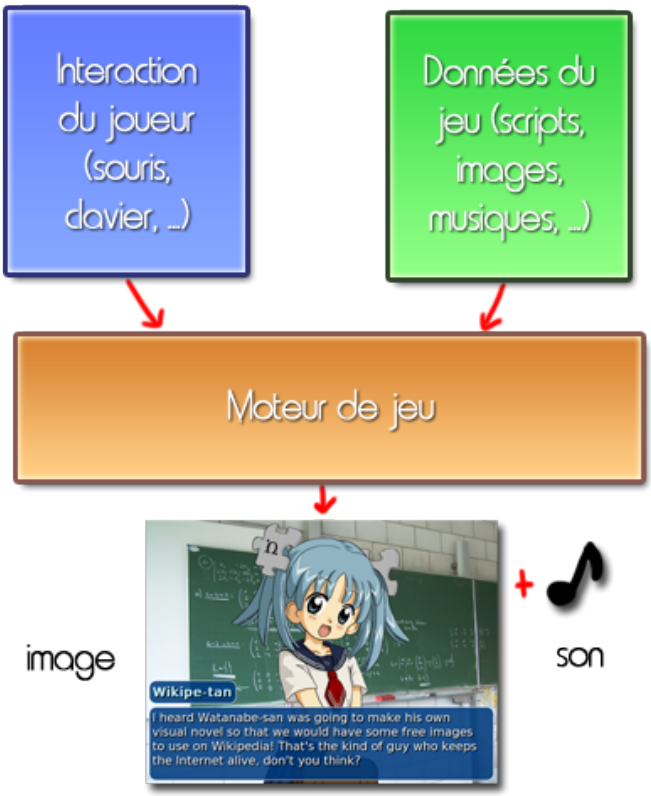

FIGURE II.1 – `Anatomie d'un roman visuel`

Parmi les différents types de ressources voici celles que l'on retrouvera principalement :

- les images, il s'agit des background, des sprites, et de toutes les images composant l'interface utilisateur (comme une boîte de dialogue par exemple).
  Il s'agira de fichiers BMP, JPEG, PNG.
- les sons, comme le bruit d'une porte qui s'ouvre ou une réplique doublée par un acteur. Il pourrait s'agir de fichiers WAV.
- les musiques.

`Attention c'est différent des sons !`

La différence étant plutôt technique, puisqu'une musique est juste un son qui est plutôt long (plusieurs minutes) et qui doit donc être géré différemment par le moteur de visual/jeu. Les formats de fichier sont là aussi bien connus : MP3, OGG, etc.

- les scripts. Il s'agit du script du roman visuel. Comme le script d'un film : c'est

à dire le déroulement des scènes, la narration et les dialogues. Il s'agit de formats textuels. Ulltimaade stockera les scripts sous un format implémentant le modèle de données défini pour le logiciel.

Les choix réalisés par le joueur ou lecteur du visual, et leurs conséquences sont exprimés via des arbres de dialogues (`dialog tree`).

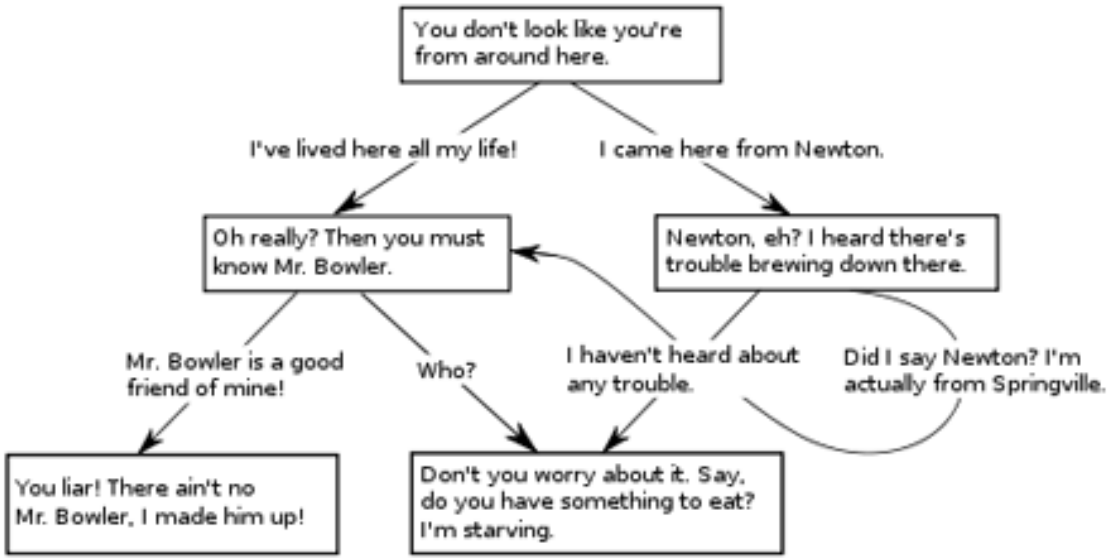

FIGURE II.2 – `Arbre de dialogue d'un roman visuel`

II.5.2 Implémentation du modèle de données

Le document architectural (AA2) un projet décrit succinctement le modèle de données d'un projet Ultimaade, avec les diagrammes associés.

Comme mentionné dans document d'architecture ces données seront stockées sur le disque local de l'utilisateur. Le choix technique entre XML et SQLITE3 sera fait en fonction des tests sur l'optimisation apportée par ces deux bibliothèques.

La prochaine version de ce document tranchera définitivement sur la problématique. Elle sera assortie des classes et méthodes permettant la gestion de ces données (lecture, écriture, modification/mise à jour).

II.6 Gestion des préférences, options et réglages du logiciel (Aide, Docs, Tutos)

II.7 Les niveaux d'utilisation : Basique, Normal, Professionnel

Chapitre III

L'API Ultimaade

L'API Ultimaade définit la manière dont les plugins du logiciel seront développés.

Elle inclut les objets Ultimaade dont l'utilisation est réservée exclusivement aux plugins.

Ces deux parties de l'API ont déjà été mentionnées dans les grandes lignes ci-dessus, et feront l'objet d'une documentation à part.

Cette documentation sera publiée pour la communauté des développeurs du logiciel, avec des explications succinctes et exemples de code.