3DWIGS

3D Web Interactive Game Studio

Manuel de reprise

Equipe Mini3D : Berlon Antoine, Bouzillard Jérôme, Chegham Wassim,
Clergeau Thomas, Faghihi Afshin, Guichaoua Mathieu, Israël Quentin,
Kien Emeric, Le Corronc Thibault, Le Galludec Benjamin,
Le Normand Erik, Lubecki Aurélien, Marginier David, Sanvoisin Aurélien,
Tolba Mohamed Amine, Weinzaepfel Philippe et Zadith Ludovic

Responsable Projet : Docteur Rémi Cozot

Année de réalisation : Année Universitaire 2010-2011

Ce document a été créé dans le but d'assurer la reprise du projet « Plateforme de création de mini-jeux 3D sur le WEB ». Ce projet a été initié par le Dr R.Cozot dans le cadre des Unités d'Enseignement GPL et PROJ du premier et second semestre de Master 1 à l'ISTIC. Avant de lire ce document, il est important de s'être d'abord intéressé au manuel d'utilisation du logiciel. Ce rapport est d'un grand intérêt, en effet il permet d'avoir une connaissance directe des points qui n'ont pas pu être complètement développés et des améliorations prévues par les membres du groupe du projet Mini3D. Dans ce document nous nous intéresserons tout d'abord à la reprise du langage de description, puis à celle de l'Interface Homme Machine. Une troisième partie présentera les insuffisances du compilateur et ses améliorations prévues, suivie d'une partie consacrée aux modifications du moteur 3D. Enfin, vous trouverez une annexe indiquant où trouver les sources de l'ensemble du projet.

Table des matières

I. Le langage de description....4
1. Gestion de l'Intelligence Artificielle....4
2. Phases de jeu....4
3. Affichage de chaîne de caractères personnalisables....4
4. Interactions (base de données de dialogues)....4
5. Affichage préprogrammé / personnalisable....4
II. L'interface homme machine....5
1. Technologies utilisées....5
2. Organisation du code....5
3. Contenu du répertoire « ./assets/js/ »....6
4. Configuration de l'auto-complétion dans l'éditeur de code....6
5. Contenu du répertoire utilisateur « /u/ »....7
a. Description des répertoires....8
b. Fichiers critiques....8
6. Convention de nommage (namespace)....8
7. Améliorations possibles....9
8. Extension de l'IHM....9
III. Le compilateur....10
1. Pourquoi ANTLR ?....10
2. Répartition du code....10
3. Améliorations possibles....10
4. Comment étendre le compilateur ?....11
IV. Le moteur 3D....12
1. Organisation du moteur 3D....12
2. Listing des variables globales du moteur....12
3. Commentaire sur la gestion du graphe de scène....12
4. Commentaire sur la gestion des collisions....12
5. Commentaire concernant les transformations géométriques....13
6. La librairie GLGE....13
7. Commentaire sur les améliorations prévues....13
8. Comment étendre le moteur 3D ?....14
V. Annexe....15
1. Où trouver les sources....15
2. Structure du compilateur....15
3. IndexedDB....17

I. LE LANGAGE DE DESCRIPTION

1. Gestion de l'Intelligence Artificielle

La gestion de l'intelligence artificielle se fait seulement par des règles du jeu. Elle est basique mais sa création par l'utilisateur pourrait être améliorée en offrant des définitions préprogrammées (langage de 3DWIGS ou Javascript) pouvant être appelées.

2. Phases de jeu

L'ajout de phases de jeu serait possible pour la création de RPG par exemple (phase d'exploration, phase de combat...). Cependant, le nombre d'objets 3D supplémentaires chargés en mémoire ou le changement d'environnement d'une phase à l'autre pourrait entraîner des baisses de performances sur le navigateur utilisé pour le jeu créé.

3. Affichage de chaîne de caractères personnalisables

Il y a deux facteurs importants dans les jeux que nous n'avons pas développés :

En premier, le scénario. Pour le moment, seuls les médias permettent de raconter une histoire par le biais de l'image et du son. Ce qui permettrait à l'utilisateur "non-cinéaste" de conter son histoire serait l'ajout d'une commande d'affichage d'une chaîne de caractères personnalisable. Comme dans les langages de programmation, elle pourrait être de la forme :

```
        display "L'histoire qui vous est contée est celle de " + name of
personnage1 + ".";
```

4. Interactions (base de données de dialogues)

Le deuxième facteur est l'immersion et le fait que lorsqu'on visite une ville dans un jeu, on peut s'attendre à voir parler les personnages. À la manière de la commande d'affichage précédente, il pourrait y avoir une base de données de dialogues liés au nom d'entités déjà déclarées dans les initialisations. Une commande « parler » permettrait d'afficher la chaîne de caractères correspondante.

```
        definition parler means
            if personnage1 touches Character then
                personnage1 talkto Character
            endif;
```

5. Affichage préprogrammé / personnalisable

Enfin, dans l'optique d'afficher des informations essentielles comme la vie d'un personnage, le nombre de munitions restantes ou le temps qu'il reste pour finir un niveau, des interfaces préprogrammées pourraient être choisies par l'intermédiaire d'un attribut à ajouter dans « game ». Sinon, pour un affichage complètement personnalisable, il faudrait ajouter un nouveau type nommé InterfaceElement contenant une position en deux dimensions par rapport à l'origine de l'écran, l'attribut d'un objet créé à afficher ('life', 'speed'...), une valeur numérique pour la taille des caractères.

II. L'INTERFACE HOMME MACHINE

1. Technologies utilisées

- ✔ Html5 : pour l'utilisation du contexte canvas 3D,
- ✔ CSS3 : pour un affichage plus moderne,
- ✔ PHP 5.3 : pour l'exécution des scripts côté serveur,
- ✔ Java 1.6 : pour l'exécution du compilateur,
- ✔ JavaScript :
  - jQuery 1.5.1 (http://jquery.com)
  - jQuery UI 1.8.11 (http://jqueryui.com)
  - Modernizr 1.6 (http://www.modernizr.com)
  - File Uploader (http://github.com/valums/file-uploader)
  - CodePress Editor
  - JSON
  - GLGE (http://www.glge.org/)
  - JSLint (http://www.jslint.com/)

2. Organisation du code

L'arborescence suivante (figure 1) représente le contenu du répertoire d'installation de 3DWIGS.

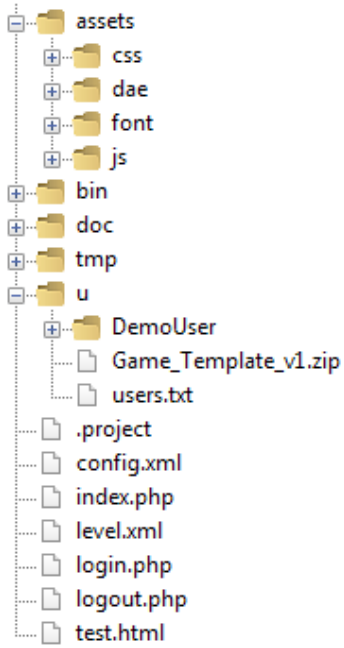


Illustration 1: Répertoire d'installation de 3DWIGS

- ✔ **/assets/** : contient les fichiers css, js et medias. Il contient également les modèles Collada.
- ✔ **/bin/** : contient les scripts php, et le compilateur.
- ✔ **/u/** : contient les créations des utilisateurs.
- ✔ **/config.xml** : contient les règles de configuration de l'éditeur.
- ✔ **/level.xml** : est utilisé par la librairie GLGE pour l'initialisation du caneva.

3. Contenu du répertoire « ./assets/js/ »

L'arborescence suivante représente le contenu du répertoire /assets/js/ (Figure 2)

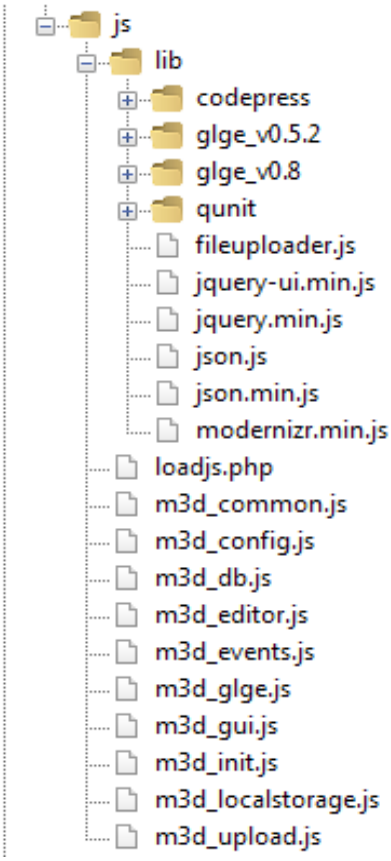


Illustration 2: Répertoire /assets/js/

Ce répertoire contient tous les fichiers JS (modules) utilisés par l'éditeur. Ainsi que toutes les dépendances nécessaires au bon fonctionnement de l'application.

Le fichier « loadjs.php » est un script permettant de combiner tout le code Javascript en un seul fichier afin d'optimiser le chargement de l'application et de réduire le nombre de requêtes entre le serveur et le client au démarrage de l'application.

4. Configuration de l'auto-complétion dans l'éditeur de code

Afin d'activer l'auto-complétion dans l'éditeur de code, les deux fichiers suivants ont été configurés pour reconnaître les règles de la grammaire 3DWIGS :

- ✔ **/assets/js/lib/codepress/languages/edwigs.js** : contenant les règles Javascript permettant de reconnaître la grammaire 3DWIGS et d'activer l'auto-complétion,
- ✔ **/assets/js/lib/codepress/languages/edwigs.css** : permettant d'appliquer des propriétés CSS afin d'activer la coloration syntaxique associée aux règles définies dans le fichier edwigs.js.

L'emplacement de ces fichiers est montré sur la figure 3.

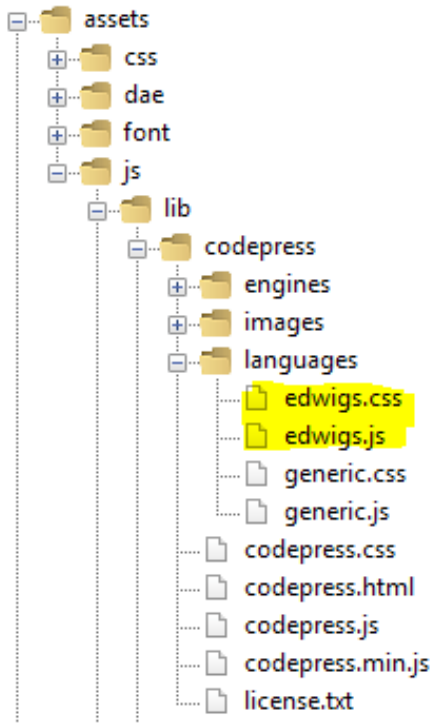


Illustration 3: Fichiers de configuration de l'auto-complétion

5. Contenu du répertoire utilisateur « /u/ »

L'arborescence ci-dessous représente le contenu du répertoire qui sera généré pour chaque utilisateur.

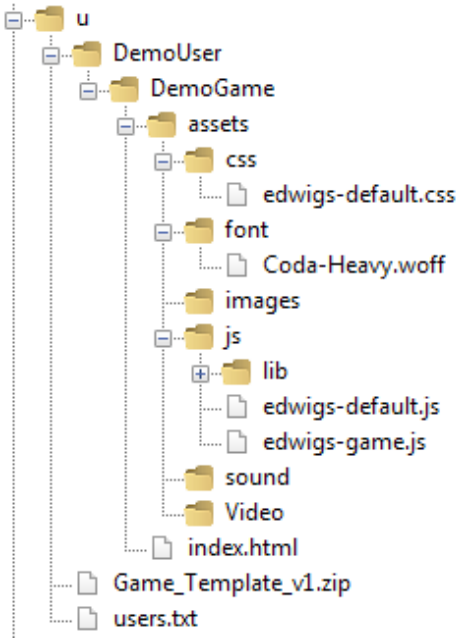


Illustration 4: Contenu du répertoire utilisateur

a. Description des répertoires

Voici une brève description du rôle de chaque répertoire :

- ✔ **DemoUser/** : correspond au nom du répertoire de l'utilisateur. Ce répertoire contient tous les jeux créés par cet utilisateur.
- ✔ **DemoUser/DemoGame/** : correspond au répertoire du jeu. Ce répertoire contient tous les fichiers nécessaires à l'exécution du jeu.
- ✔ **DemoUser/DemoGame/assets/** : correspond au répertoire qui contient tous les fichiers JavaScript et feuilles de style ainsi que tous les fichiers utiles au fonctionnement du jeu.
- ✔ **DemoUser/DemoGame/assets/css/** : contient les feuilles de style.
- ✔ **DemoUser/DemoGame/assets/js/** : contient les scripts JavaScript.
- ✔ **DemoUser/DemoGame/assets/images/** : contient les images utilisées dans l'application.
- ✔ **DemoUser/DemoGame/assets/font/** : contient les polices utilisées par l'application.
- ✔ **DemoUser/DemoGame/assets/sound/** : contient les fichiers médias utilisés par le jeu.
- ✔ **Game_Template_v1.zip** : est une archive qui contient l'arborescence « type » d'un répertoire d'utilisateur.
- ✔ **Users.txt** : fait office d'une mini base de données et dont voici l'organisation :
  - <Login> : <Profile> : <Mot De Passe Crypté En MD5> : <Champ Vide>
  - Exemple : **DemoUser:GUEST:fe01ce2a7fbac8fafaed7c982a04e229:**

b. Fichiers critiques

Lors de la génération du répertoire utilisateur, les fichiers critiques sont les suivants :

- ✔ **index.html** : correspond au fichier qui contient l'affichage de l'application ainsi que le conteneur du jeu.
- ✔ **assets/js/edwigs-game.js** : contient tout le code du jeu. Le code étant généré par 3DWIGS directement dans ce fichier. Ce code pouvant être minimifié.

6. Convention de nommage (namespace)

Tous les modules doivent appartenir au même espace de nom afin d'éviter des collisions dans le nommage. Cet espace de nom est implémenté dans le module « m3d_init.js », et est organisé comme suit :

- ✔ **Window.M3D** : l'espace de nom global,
- ✔ **Window.M3D.Config** : l'espace de nom réservé à la configuration,
- ✔ **Window.M3D.GUI** : l'espace de nom réservé au module « m3d_utils.js »,
- ✔ **Window.M3D.DB** : l'espace de nom réservé au module « m3d_localstorage.js » ET « m3d_db.js »,
- ✔ **Window.M3D.Common** : l'espace de nom réservé aux opérations globales,
- ✔ **Window.M3D.Editor** : l'espace de nom réservé à l'éditeur de code du module « m3d_editor.js »,
- ✔ **Window.M3D.Uploader** : l'espace de nom réservé au module de chargement des modèles dans l'éditeur.

Note : Pour la gestion de la base de données, l'application utilise par défaut le module «m3d_localstorage.js» à cause de quelques complications rencontrées avec le module « m3d_db.js » qui est basé sur l'API d'indexedDB (cf. documentation de l'IndexedDB en annexe).

Toutefois, ce module est utilisable en modifiant par src=" assets/js/loadjs.php?db " dans le fichier index.html et la variable indexed à « true » dans m3d_editor.js.

7. Améliorations possibles

Voici une liste des fonctionnalités à améliorer :

1. Permettre une meilleure gestion de la caméra : permettre l'ajout et l'édition des propriétés des caméras actives.
2. Améliorer le processus de chargement des modèles Collada dans l'éditeur, notamment en permettant à l'utilisateur d'importer des modèles avec textures.
3. Améliorer l'affichage des modèles présents dans le canvas, ainsi que ceux proposés, sous forme de vignette pour une meilleure ergonomie.

Voici une liste des fonctionnalités à implémenter :

4. Positionnement des objets sur un plan…
5. … puis sur un terrain (collision).
6. Implémentation d'un graphe de scène.
7. Positionnement de la caméra par rapport à un objet.

8. Extension de l'IHM

Parmi les évolutions possibles qui peuvent être apportées à l'IHM, on peut citer :

1. Parallélisation de certains traitements en utilisant l'API WebWorkers offerte par HTML5,
2. Ajout d'un module d'import/export permettant ainsi à l'utilisateur d'exporter ou d'importer le contenu de sa base de données en JSON (recommandé), XML ou CSV.
3. Offrir une version d'une partie de l'application disponible en mode Offline, et ceci en exploitant l'API Offline Application Caching.

III. LE COMPILATEUR

1. Pourquoi ANTLR ?

Lors de la réalisation du compilateur, nous avions une liberté totale sur le choix des outils, entre autre celui pour la gestion du langage (parser et lexer). Faisant directement suite au cours de compilation, notre choix s'est naturellement tourné vers ANTLR pour la simple et bonne raison que nous l'avions déjà utilisé. De plus, étant écrit en Java, il permettait à notre compilateur d'être multiplateforme et cela sans avoir à adapter le code. Néanmoins, dans un souci d'objectivité, nous avons également examiné les autres alternatives (ie : xtext). Mais aucune n'est ressortie, soit parce qu'elle ne correspondait pas à nos attentes, soit parce qu'elle n'apportait rien de plus.

2. Répartition du code

Le langage est découpé en 6 parties différentes :

1. Les Modèles. Pour faire une analogie avec la Programmation Orientée Objet (POO), il s'agit d'une classe à l'exception que le modèle n'est qu'un conteneur de données. C'est un pattern que l'on peut reprendre pour décrire un élément de l'environnement.
2. Les Entités. Il s'agirait des objets en POO. C'est une représentation concrète en mémoire de l'état d'un élément présent dans l'environnement. À noter que tous les éléments n'ont pas forcément de représentation graphique (ex : un compteur).
3. Les Définitions. Ce sont des fonctions qui peuvent prendre en paramètres une ou plusieurs entités afin d'effectuer des actions dessus. Les actions sont exécutées de façon itérative et peuvent être effectuées sous certaines conditions en fonction de ce que l'utilisateur précise.
4. Les Commandes. Cela correspond à l'interface utilisateur (clavier et souris). Le but est ici d'attribuer à une touche ou un déplacement de souris une action particulière. Pour ce faire, une définition est rattachée à l'action faite sur l'interface utilisateur.
5. Les Règles. Les règles du jeu sont le cœur même du compilateur. Il s'agit de la notice du jeu. L'action d'une règle est décrite dans une définition. Lorsqu'une condition (déclencheur) est remplie, on appelle la définition associée à ce déclencheur. Différentes règles peuvent s'exécuter en parallèle.
6. Les Intelligences Artificielles. Parler d'intelligences artificielles est un peu présomptueux. Il s'agit en fait d'une suite de règles fonctionnant sur le principe de l'action / réaction. Cette section reprend en grande partie la section sur les règles du jeu.

Pour éviter à l'utilisateur de partir de rien, nous avons mis à sa disposition un certain nombre de Modèles qui sont chargés en mémoire au début de la compilation. Il peut ainsi se baser sur ces modèles pour créer son jeu. Ces modèles sont stockés dans un fichier XML. Dans un souci d'optimisation, seuls les modèles utilisés par le jeu sont générés en JavaScript.

3. Améliorations possibles

Des améliorations et des fonctionnalités peuvent être ajoutées. Il faudrait compléter le compilateur au niveau des définitions et des commandes. La gestion des fichiers médias est la première de toutes. Cela permettrait à l'utilisateur d'intégrer du son et des vidéos dans le jeu. D'autre part, il faudrait coder les actions non réalisées dans les définitions. De plus, il serait nécessaire de gérer les commandes à la souris, notamment la récupération des coordonnées et la détection des sources de clics de façon générique (pour tous les navigateurs). Ou encore la révision en profondeur des intelligences artificielles pour aller au-delà du simple action / réaction présent actuellement. Il serait aussi possible de mettre en place le système de contrainte qui est uniquement présent dans le XML et pas du tout utilisé. Enfin pour finir, une optimisation du code JavaScript généré est à prévoir afin que l'exécution soit plus rapide chez les utilisateurs.

4. Comment étendre le compilateur ?

Le langage de description de 3DWIGS est lié au compilateur via un fichier d'extension '.g', nommé « high.g » qui a été réalisé sous Antlr. Il se situe dans la partie Compilation du logiciel. Voici son arborescence d'accès : « compilateur/high/com/istic/mini3d/grammars ».

Ce fichier contient la grammaire « high.g » et son arbre de syntaxe abstrait (AST) « highTree.g ».

L'ajout de règles supplémentaires se fait dans un premier temps par des ajouts dans le fichier grammaire « high.g ».

Puis l'arbre de syntaxe vide (le fichier « highTree.g ») pourra être complété afin de réaliser la génération de code désirée et la gestion d'erreur.

IV. LE MOTEUR 3D

1. Organisation du moteur 3D

Le moteur 3D s'organise de la manière suivante :

- ✔ dae : fichier qui sert de dépôt aux éléments 3D
- ✔ documentation : fichier où on peut trouver la documentation du moteur
- ✔ images : fichier qui sert au dépôt des fichiers au format .png servant pour les textures
- ✔ js : ensemble des fichiers code du moteur
    - moteur3d.js : contient les fonctions d'initialisation
    - moteur_tools.js : contient différentes fonctions outils
    - moteur_collision.js : contient les fonctions concernant les collisions
    - moteur_management.js : contient les fonctions sur les manipulations des objets dans le graphe de scène
    - moteur_movement.js : contient les fonctions de transformation des éléments
    - moteur_camera.js : contient les fonctions sur les caméras
    - moteur_physique.js : contient tout ce qui concerne l'application de la physique
- ✔ lib : fichier contenant un exemplaire de la librairie glge v0.8

Les fichiers sources du moteur sont disponibles dans le dossier js de celui-ci. En parcourant ces fichiers, chaque fonction y est clairement identifiée à l'aide d'une JavaDoc précise.

2. Listing des variables globales du moteur

Ci-dessous, la liste des variables globales du moteur accessibles dans l'ensemble des fichiers du moteur :

- ✔ gameScene : instance de GLGE.Scene représentant la scène
- ✔ tabObject : tableau indexé par des chaînes de caractères où tabObject[id] contient l'instance de GLGE.Group portant le nom id
- ✔ tabIdObject : tableau indexé par les naturels, listant les différents identifiants correspondant à des objets réels dans la scène
- ✔ tabCamera : comme tabObject mais uniquement pour les caméras

3. Commentaire sur la gestion du graphe de scène

Les objets sont ajoutés directement au graphe de scène : un groupe est créé avec l'identifiant donné, dont le fils est l'objet Collada en lui-même portant le même identifiant précédé d'un #. Cela permet de récupérer le volume englobant précis de l'objet et ce, même si plus tard, un fils est ajouté à l'objet (puisqu'il sera ajouté au groupe).

Les fonctions getChildren et getAllChildren permettent de récupérer les groupes, vrais objets, caméras et lumières, fils directs ou descendants d'un objet.

Afin de gagner en complexité, ils sont stockés dans des attributs qui sont mis à null en cas de changement et recalculés uniquement en cas de besoin.

Par exemple, un appel à getChildren renvoie l'attribut moteurChildren s'il est non null, ou refait les calculs sinon, sachant que l'attribut est mis à null en cas de suppression d'un descendant ou d'ajout d'un descendant.

4. Commentaire sur la gestion des collisions

Afin de gagner en temps de calcul, les collisions sont testées lors du déplacement d'un objet via l'idée suivante : un premier test grossier est réalisé, fait uniquement avec les fils directs de la scène. En cas de collision détectée, un raffinement est effectué afin de connaître précisément les sources de cette collision.

5. Commentaire concernant les transformations géométriques

Toutes les transformations géométriques sont réalisées en changeant la matrice locale plutôt que les positions / rotations / échelles locales proposées par GLGE. En effet cela simplifie les calculs pour, par exemple, calculer l'effet d'une rotation dans le repère d'un autre objet.

En GLGE, la matrice locale est calculée comme le produit de la matrice de translation, la matrice de rotation et la matrice d'échelle.

Ainsi, pour effectuer une translation, il suffit de multiplier la matrice locale par la matrice de la translation souhaitée.

Les valeurs de position locale peuvent être récupérées avec la multiplication de la matrice locale par le vecteur [0,0,0,1].

Les valeurs de rotation locale sont implémentées dans une fonction de moteur_tools.js (formule valable uniquement si on effectue la matrice de rotation qui est calculée comme le produit de la rotation selon x par le produit de celle selon y et selon z) avec un angle selon y dont la valeur absolue est inférieure ou égale à pi/2.

L'échelle locale peut être obtenue comme coefficients diagonaux de la matrice résultant du produit de la matrice locale multipliée par sa transposée.

6. La librairie GLGE

Cette librairie a été réalisée par le Britannique Paul Brunt. Il s'agit d'une librairie JavaScript facilitant l'utilisation de WebGL. Elle permet d'accéder directement à OpenGL ES2 et permet ainsi l'accès à l'utilisation de l'accélération matérielle des applications 2D/3D sans avoir besoin de télécharger aucun plugins.

Le but de GLGE est de masquer l'utilisation directe du WebGL au développeur web, qui peut alors se concentrer sur la création d'un contenu plus riche pour le web.

La documentation concernant cette librairie est disponible sur son site web : http://www.glge.org/

7. Commentaire sur les améliorations prévues

Dans cette partie, sont listés les différents éléments que le groupe Moteur3D n'a pas eu le temps de réaliser durant la phase de développement du logiciel :

- ✔ la gestion de la souris,
- ✔ la gestion des animations,
- ✔ la gestion de l'affichage.

Par « gestion des animations » on évoque le déclenchement d'animations liées à un modèle 3D. Par exemple, un objet 3D lapin pourrait sauter lors de ses déplacements, déployant ainsi ses pattes arrières. Il faudrait pour cela créer une animation 3D de saut du lapin et gérer son affichage.

Par « gestion de l'affichage » on évoque deux aspects. La création de fenêtre pour la gestion d'un inventaire ou d'un menu, d'une part et l'affichage des Heightmap avec texture, de l'autre. Une Heightmap (carte d'altitude en français) est une map représentant des différences de hauteurs. Dans le principe, on peut la rapprocher d'un relevé topographique.

8. Comment étendre le moteur 3D ?

Dans le chapitre "Organisation du moteur", il a été montré que les sources du moteur 3D se situaient dans le répertoire js de celui-ci. Pour des raisons d'organisation, le moteur a été séparé en différents modules regroupant chacun une spécification (gestion de déplacement, gestion des collisions, etc). Cependant, il est possible d'étendre le moteur en lui ajoutant des nouveaux modules ou des nouvelles fonctions aux modules existants.

D'une part, pour rajouter une fonction dans un module existant, il faut s'assurer que cette nouvelle fonction commence par l'intitulé "M3D.MOTEUR" afin de pouvoir être correctement identifiée. D'autre part, pour procéder à un ajout d'un nouveau module au moteur, il faut suivre la démarche suivante :

- ✔ Rajouter le nouveau module dans le répertoire js
- ✔ Dans le fichier "moteur3d.js" qui se trouve dans le répertoire js, rajouter la ligne suivante à la fin du code de la fonction "M3D.MOTEUR.includeAllJS()" :

```
M3D.MOTEUR.includeJS('js/nom_du_nouveau_module');
```

- ✔ Dans le module créé, ajouter en en-tête :

```
(function(M3D){
    listes de nouvelles fonctions.
})(window.M3D);
```

Dans le but de pouvoir accéder aux variables globales et aux autres fonctions déjà implémentées par le moteur de 3DWIGS.

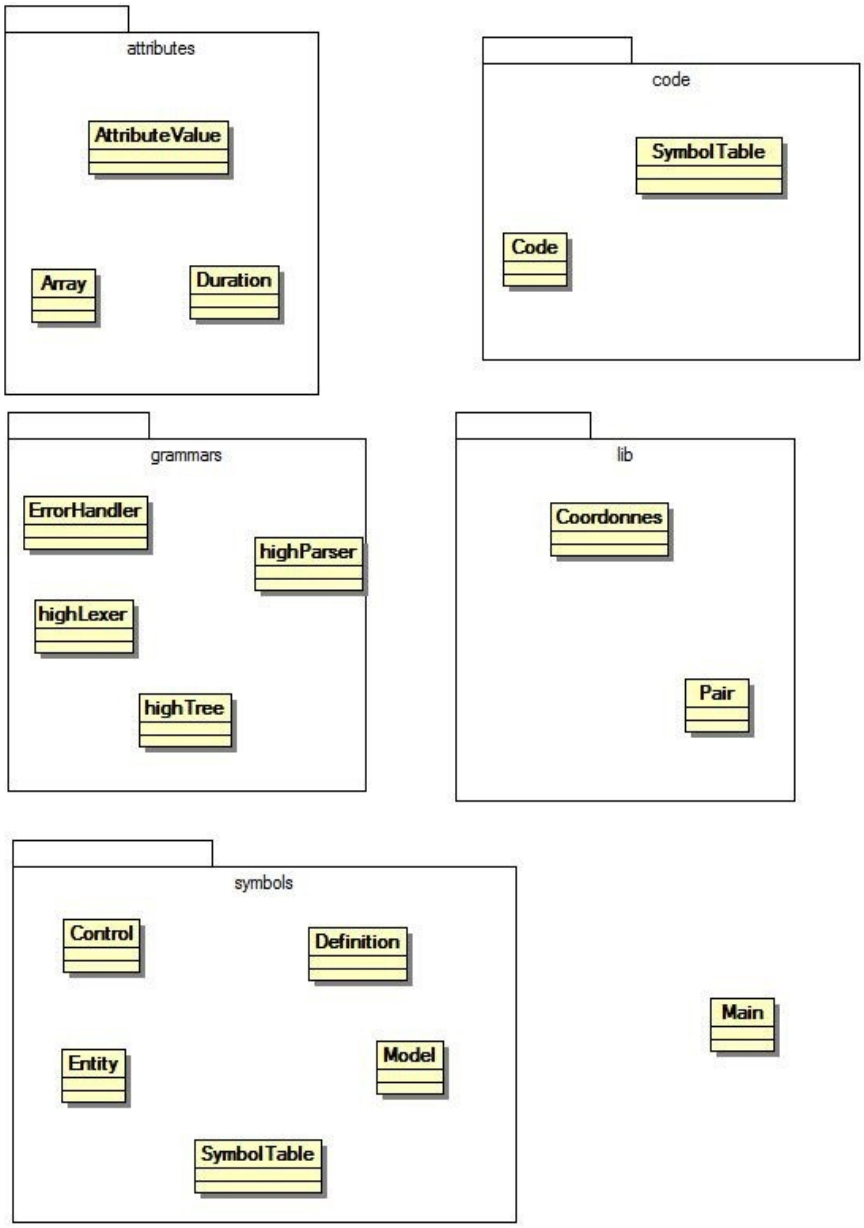
V. ANNEXE

1. Où trouver les sources

L'ensemble des fichiers du projet Mini3D est disponible à l'adresse suivante :
http://code.google.com/p/3dminigames/

2. Structure du compilateur

a. Architecture générale



b. Attributes

✔ **AttributeValue.java :**

Classe permettant de gérer le Type et la valeur des attributs dans les modèles. Par exemple : le Type prédéfini Character a un attribut life. Life est de type numeric et possède une valeur. Ces deux éléments sont exprimés grâce à un objet AttributeValue. La valeur de l'attribut est exprimée en code car la valeur n'est pas forcément exprimée de manière claire. Par exemple un numeric peut être de la forme 25+chasseur.mass

✔ **Duration.java :**

La classe Duration permet de stocker la valeur d'un attribut de type time. Elle contient la valeur (toujours sous forme de code) ainsi que l'unité (seconde, milliseconde,…).

✔ **Array.java :**

La classe Array permet de gérer la valeur d'un attribut de type array/list. Elle contient le modèle associé a la liste ainsi que le contenu de cette dernière.

c. Code

✔ **Code.java :**

Classe permettant de gérer le code JavaScript qui doit être généré. Elle contient aussi les méthodes statiques permettant la génération de bouts de code « complexe ». Par exemple : la méthode genRefreshLoop() permet de générer le code de la boucle de rafraichissement en JavaScript.

✔ **SymbolTable.java :**

Classe permettant de gérer la table des symboles du compilateur. Cette classe gère une hashTable qui contient des objets de type Symbol. Une seule instance de cette classe est créée lors de l'exécution du programme.

d. Grammars

✔ **ErrorHandler.java**

Gestionnaire d'erreurs du compilateur. Il utilise 3 types d'erreurs : WARNING, ERROR et FATAL. Les erreurs fatales sont déclenchées par ANTLR, mais aussi lors du traitement des données. Suite à ce déclenchement, le compilateur s'arrête et affiche l'erreur. Les erreurs « classiques » n'interrompent pas le compilateur. Et les warnings sont des alertes en cas d'incertitude du compilateur mais où le compilateur a quand même pris une décision. Elles n'arrêtent pas la compilation.

✔ **highLexer.java, highParser.java et highTree.java**

Classe du parser, lexer et la classe exécutive du compilateur. Ces trois fichiers sont générés à l'aide d'ANTLR et des fichiers high.g et highTree.g (qui se situent dans le même répertoire).

e. lib

✔ **Coordonnees.java**

Classe permettant de gérer les coordonnées d'un model lorsque l'utilisateur utilise le mot clé « coordinate » par exemple. En gros quand l'utilisateur doit saisir les valeurs des 3 coordonnées d'un coup.

✔ **Pair.java**

Classe permettant de créer des Paires (association de deux objets).

f. Symbols

✔ **Symbol.java**

Interface implémentée par model, entity, definition et control. Cette interface regroupe toutes les signatures des méthodes utiles lors de l'utilisation de symbols récupérés par l'intermédiaire de la symbolTable.

✔ **Model.java**

Classe permettant de définir tous les « Types » chargés au démarrage ou créés par l'utilisateur. Cette classe se charge de parser et de charger dans la SymbolTable les « Types » prédéfinis.

✔ **Entity.java**

Classe permettant de définir toutes les Entités créées par l'utilisateur.

✔ **Definition.java**

Classe permettant de définir toutes les Définitions créées par l'utilisateur.

✔ **Control.java**

Classe permettant de définir toutes les Commandes créées par l'utilisateur.

g. Autres classes

✔ **Main.java**

Classe principale permettant de compiler un fichier pour 3DWIGS.

3. IndexedDB

IndexedDB étant récente et encore en évolution pour l'implémentation de la technologie dans les navigateurs, on trouve pour le moment très peu de documentation. Voici un résumé des possibilités actuelles sur Firefox (4.0 Beta 10) et Chrome (dev 11).

a. Charger l'API

```
« window.indexedDB = window.mozIndexedDB; »
```

b. Créer et ouvrir la base de données

```
« var idbRequest = window.indexedDB.open("m3d","3DWIGS project"); »
```

c. Tester si base de données déjà existante

A chaque création de table on doit mettre à jour la version de la base de données, ici on placera toujours la version à "1". On peut alors tester si la version a été initialisée afin de savoir si l'utilisateur a déjà sa base de données créée.

```
« idbRequest.onsuccess = function(event) {
      db = idbRequest.result;
      var e= idbRequest;
      if(e.result.version != "1"){
                  //création des tables
      }else{ //chargement des données
      }
}
idbRequest.onerror = function(e){log("Error: IndexedDB init");}; »
```

d. Créer une table

```
« var request = db.setVersion('1');
request.onerror = function(e){
      log("Error: IndexedDB create table");
};
request.onsuccess = function(e) {
      db.createObjectStore('/nom_table/', {keyPath: 'id'});
} »
```

e. Ajouter un object dans une table

```
« var objectStore = db.transaction([],
IDBTransaction.READ_WRITE).objectStore("/nom_table/");
var request = objectStore.add({/format json exemple: name: "toto", sexe:
"m"/}, id: "/id_object/"});
request.onerror = function(e){
      log("Error: IndexedDB setObject");
};
request.onsuccess = function(event) {log("Add object k");}
; »
```

f. Modifier un object

```
« var objectStore = db.transaction([],
IDBTransaction.READ_WRITE).objectStore("/nom_table/");
objectStore.delete(/id_object/);
var request = objectStore.add({/format json exemple: name: "toto", sexe:
"m"/}, id: "/id_object/"});
request.onerror = function(e){log("Error: IndexedDB update object");};
request.onsuccess = function(event) {log("Update object k");}
; »
```

g. Supprimer un object

```
« var objectStore = db.transaction([],
IDBTransaction.READ_WRITE).objectStore("/nom_table/");
if (objectStore.delete) {
      var request = objectStore.delete(/id_object/);
} else {
      var request = objectStore.remove(/id_object/);
} »
```

h. Supprimer une table

```
« if (db.deleteObjectStore) {
      db.deleteObjectStore('/nom_table/');
} else {
      db.removeObjectStore('/nom_table/');
} »
```

i. Charger un object

Pour récupérer les infos d'un object en particulier:

```
« var objectStore = db.transaction([],
IDBTransaction.READ_WRITE).objectStore("/nom_table/'");
var request = objectStore.get("/id_object/");
request.onsuccess=function(e){
      //request.result permet d'accéder aux infos de l'object
}
request.onerror = function(e){log("Error: IndexedDB load");};
»
```

Pour récupérer les infos de tous les objects d'une table:

```
« var request = db.transaction([],
IDBTransaction.READ_ONLY).objectStore("/nom_table/").openCursor();
var tmpArr = [];
request.onsuccess = function(e) {
      var cursor = request.result; if (!cursor) {
            var len = tmpArr.length; for(var i=0; i<len; i++){
```

```
                    /tmpArri? permet d'accéder aux infos de l'objet i/
                }
            }
        tmpArr.push(cursor.value); cursor.continue();
        }
        request.onerror = function(e){log("Error: IndexedDB load");};
        »
```

j. Non implémentés et problèmes

L'API synchrone n'est toujours pas implémentée. Pour l'ajout et modification d'un object, la fonction put() n'est pas implémentée sur Firefox. **L'API asynchrone ne permet pas d'avoir une fonction 'get'.**