ECOLE SUPERIEURE D'INGENIEURS DE RENNES

UNIVERSITÉ DE RENNES 1

AtomicGL 2

DOCUMENTATION UTILISATEUR

Projet AtomicGL 2 |2015 - 2016

ESIR 2 | Imagerie Numérique

BILLEL HELALI – LUCAS FOUCAULT – IFIC GOUDÉ – ALEXIS ROCA

Table des matières

INTRODUCTION **2**

PAGE WEB **3**

Head 3

Body 3

SCÈNE **3**

Lights 3

ID 4

COLOR 4

POSITION 4

DIRECTION 4

RADIUS 4

RADIOSITY 4

Shaders 5

Textures 5

Shapes 5

SceneGraph 6

ROOT 6

TRANSFORM 6

OBJECT3D 6

SHADER **7**

Vertex 7

ATTRIBUTES 7

UNIFORMS 7

VARYING 7

OUTPUT 7

Fragment 8

PRECISION 8

UNIFORMS 9

VARYING 9

OUTPUT 9

OBJET 3D **9**

TEXTURE **9**

INTRODUCTION

Le moteur graphique AtomicGL 2 est un moteur Web basé sur WebGL. Il a pour volonté d'être simplissime d'un niveau atomique et pour autant très efficace.

Dans ce document, vous allez apprendre à utiliser ce moteur et vous verrez à quel point il est simple de créer sa première scène en 3D.

Avant de commencer, laissons-nous vous offrir quelques informations historiques sur ce moteur : La première version a été créé par Rémi Cozot dans un but éducatif. Son objectif principal était de faire apprendre à ses étudiants en Imagerie Numérique à programmer des Shaders. Il a fait le choix d'utiliser le langage de programmation JavaScript pour, d'une part facilité l'intégration Web, et d'autre part initier les étudiants à ce langage de plus en plus utilisé dans le monde des entreprises. Cette première version se nommait AtomicGL et était très peu structuré. Un code unique HTML englobait les multiples balises JavaScript en allant des Shaders jusqu'au graphe de scène. Quelques classes JavaScript étaient déjà présentes comme la gestion de l'horloge ou le contrôle de la caméra ; néanmoins le code restait compliqué à appréhender.

La programmation de la seconde version d'AtomicGL fut l'objet d'un projet étudiant. L'objectif premier était de clarifier et structurer le code. Les différents blocs vous seront présentés par la suite. Un second objectif était d'améliorer les capacités de ce moteur, ajouter des lumières et programmer des shaders basiques.

Vous pouvez maintenant commencer votre apprentissage.

PAGE WEB

ATOMICGL.HTML

Head

L'entête du fichier HTML contient toutes les classes JavaScript à importer dans le projet. Toutes nouvelles classes utiles au projet doivent être importées en suivant cette syntaxe :

```
<script type="text/javascript" src="classeJavaScript.js"></script>
```

Body

Le corps du fichier HTML contient la définition de la scène Atomic désirée ainsi que l'appel à la méthode *webGLStart()* pour exécuter le programme. La définition de la scène se construit de cette façon :

```
<div id="scene" style="visibility:hidden; height:0%;">maScene</div>
```

L'appel à la méthode *webGLStart()* se fait simplement :

```
<script type="text/javascript">webGLStart();</script>
```

SCÈNE

SCENE.XML

Lights

La première chose à définir dans notre scène sont les lumières. Il existe trois types de lumières et elles se déclarent comme ceci :

Lumière Ponctuelle :

```
<POINTLIGHT id="PointLight0" color="1.0,1.0,1.0" position="0.0,0.0,0.0" radioisity="1.0,1.0,1.0"></POINTLIGHT>
```

Lumière Directionnelle :

```
<DIRECTIONNALLIGHT id="DirecionnalLight0" color="1.0,1.0,1.0" direction="0.0,0.0,0.0" radioisity ="1.0,1.0,1.0"></DIRECTIONNALLIGHT>
```

Lumière Spot :

```
<SPOTLIGHT id="SpotLight0" color="1.0,1.0,1.0" position="0.0,0.0,0.0" direction="0.0,0.0,0.0" radius="1.0" radioisity ="1.0,1.0,1.0"></SPOTLIGHT>
```

ID

L'identifiant d'une lumière doit strictement respecter cette notation. Pour une lumière ponctuelle, l'identifiant doit se composer de ***PointLight*** suivi du numéro de la lumière associée. Pour la première lumière ponctuelle, l'identifiant sera ***PointLight0***, si une deuxième lumière ponctuelle est ajoutée dans la scène, son identifiant sera ***PointLight1***, et ainsi de suite.

La même rigueur syntaxique doit être appliquée aux autres types de lumière.

COLOR

La couleur se compose des trois composantes RGB. En théorie, une composante se définit avec une valeur entre 0 et 1, et la somme des trois composantes doit être égale à 1. En théorie, pour une lumière blanche, l'attribut *Color* doit être égal à ***(0.33,0.33,0.33)***. Ceci dit, la gestion de la couleur nous permet de proportionner la contribution de chaque composante comme on le souhaite. C'est-à-dire que l'on peut définir une couleur très Rouge, légèrement bleutée de cette manière : ***(10.0,0.0,0.5)***

POSITION

La position globale dans la scène doit être donnée ici. La lumière directionnelle n'a pas de position à proprement parler car elle est censée représenter la lumière du soleil, et éclairer la globalité de la scène en provenance d'une direction donnée.

DIRECTION

La direction est relative à la lumière. Dans le cas de la lumière spot, cette direction est donnée en fonction de la position du spot. Pour une lumière ponctuelle, aucune direction n'est à fournir car ce type de lumière est omnidirectionnelle.

RADIUS

Cet attribut n'existe que pour le spot. Il détermine le rayon éclairé par le spot à une distance relative de 1 depuis la position du spot. Exemple, si un spot positionné à 10 au-dessus d'un plan et que sa direction pointe vers ce sol. Un *Radius* de 1 dessinera un cercle lumineux sur le sol d'un rayon de 10.

RADIOSITY

La radiosité se compose de trois composante alpha, beta et gamma qui gèrent la propagation des ondes lumineuses dans la scène.

Shaders

Dans la version 2.0 d'AtomicGL, seulement les Shaders de matériaux sont disponibles. Les Shaders multi passe sont en prévision mais non fonctionnels pour le moment.

Un *Shader* se définit comme suit :

```
<XMATSHADER id="monProg">monShader.xml</XMATSHADER>
```

L'identifiant sera celui utilisé pour appliquer un shader à un objet.

Textures

Une texture se définit comme suit :

```
<TEXTURE id="maText" type="color ">maTexture.png</TEXTURE>
```

L'identifiant sera celui utilisé pour appliquer une texture à un objet.

Le type doit être mit à color, c'est aussi en prévision des différents Shaders autre que ceux de matériaux que le type de la texture à une importance. Cependant, les autres Shaders n'étant pas compatibles pour le moment, seulement le type color pour les textures est admis.

Shapes

Une bibliothèque d'objet standard est disponible. Cette dernière comprend le *Cube,* le *Cylindre,* le *Plan en XY,* le *Plan en XZ* et la *Sphère*. Ceux-ci ont des attributs particuliers pour chacun, voici leur syntaxe :

```
<CUBE id="monCube" height="10" width="10" depth="10"></CUBE>
<CYLINDER id="monCylindre" height="10" rad="1" lat="10" long="10"></CYLINDER>
<XYPLANE id="monPlanXY" height="10" width="10" xrow="10" yrow="10"></XYPLANE>
<XZPLANE id="monPlanXZ" height="10" width="10" xrow="10" zrow="10"></XZPLANE>
<SPHERE id="maSphere" rad="10" lat="10" long="10"></SPHERE>
```

D'autres objets peuvent être importés depuis l'extérieur. Ce sont soit des objets de type .obj (exportés depuis Sketchup, 3DS… au format obj) soit de type .js (tous les points, les triangles, les normales… sont définies dans le constructeur de la classe JavaScript.

```
<SHAPE id="monObjet" type="js">monObjetJS()</SHAPE>
<SHAPE id="monObjet" type="obj">monObjetObj</SHAPE>
```

Pour les objets JavaScript, le constructeur de la classe est appelé, c'est pour cela que les parenthèses sont nécessaire. Pour les obj, juste le nom de l'objet est nécessaire, un parsage est automatiquement exécuté à la suite de cette déclaration.

L'identifiant sera celui utilisé dans le graphe de scène. D'autres paramètres optionnelles peuvent être ajoutés aux objets. Le paramètre *color* est utile pour appliquer des shaders utilisant l'attribut Color. Le paramètre *tex* est utile pour appliquer des shaders utilisant des textures. Plusieurs textures peuvent être passé en paramètre séparé par des virgules. Si des textures sont appelés, les coordonnées de textures sont nécessaire. Voici la syntaxe pour paramétrer les objets.

```
color="1.0,1.0,1.0"
```

```
tex="maTexture0,maTexture1" uv="1.0,1.0"
```

SceneGraph

ROOT

Le graphe de scène se construit tout d'abord sur le nœud *Root* qui prend un certain nombre de paramètre. Premièrement, une texture pour la SkyBox doit être passée en paramètre, ainsi que sa dimension et le shader appliqué à la boîte. Ensuite, le type de caméra utilisé pour la scène, seulement la caméra *Walk* est implémentée pour le moment. Enfin, l'objet de navigation (c'est-à-dire le mesh sur lequel la caméra est autorisée à se déplacer) doit être l'identifiant de la shape désirée précédemment créée.

On aura ce style de syntaxe :

```
<ROOT skybox="skyTexture" skysize="100.0" skyshader="monShader" camera="walk"
navmesh="monNavMesh"></ROOT>
```

À l'intérieur de ces balises *Root,* on pourra trouver trois sorte de nœuds différents.

TRANSFORM

Ce nœud correspond à une transformation dans le graphe de scène. Il comporte un identifiant, une translation, un axe de rotation et un angle.

```
<TRANSFORM id="maTransform" translate="10.0,0.0,0.0" rotaxis="1.0,0.0,0.0"
angle="45.0"></TRANSFORM >
```

OBJECT3D

Un objet 3D se caractérise par un identifiant, un shader et prend en attribut une shape.

```
<OBJECT3D id="monObjet" shader="diffuseTextureShader">maSphere0</OBJECT3D>
```

SHADER

SHADER.XML

Les Shaders doivent obligatoirement être sauvegardés dans le dossier shader.

Vertex

Toute la déclaration du Vertex Shader est comprise entre ces balises :

```
<VERTEX></VERTEX>
```

ATTRIBUTES

Tous les attributs appelé par le vertex sont à déclarer dans ces balises :

```
<ATTRIBUTES></ATTRIBUTES>
```

La déclaration d'un attribut doit contenir le nom de la fonction de l'attribut, pour la position, l'attribut devra contenir « position ». On pourra obtenir ce genre d'attribut :

```
attribute vec3 aVertexPosition ;
attribute vec3 aVertexColor ;
```

UNIFORMS

Les uniformes sont gérés strictement de la même manière.

```
<UNIFORMS></UNIFORMS>
```

Les trois matrices (Modèle View, Projection et Normal) sont disponibles en uniforme.

```
uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;
uniform mat4 uNMatrix;
```

VARYING

Les varyings sont gérés strictement de la même manière.

```
<VARYING></VARYING>
```

OUTPUT

L'output prend le code à proprement parler du Vertex. Il comporte le *main()* et définit le paramètre gl_Position.

```
<OUTPUT></OUTPUT>
```

Fragment

Toute la déclaration du Fragment Shader est comprise entre ces balises :

```
<FRAGMENT></FRAGMENT>
```

PRECISION

La précision vérifie la compilation du Shader grâce à ces quelques lignes :

```
<PRECISION>
	#ifdef GL_ES
		precision highp float;
	#endif
</PRECISION>
```

La précision prend, en outre, pour un shader utilisant les lumières dynamiquement, le nombre de lumière de chaque type ainsi que leur structure.

```
#define NB_POINTLIGHTS
#define NB_SPOTLIGHTS
#define NB_DIRECTIONNALLIGHTS

struct pointLight {
	vec3 position;
	vec3 color;
	vec3 intensity; };

struct spotLight {
	vec3 position;
	vec3 direction;
	vec3 color;
	float radius;
	vec3 intensity; };

struct directionnalLight {
	vec3 direction;
	vec3 color;
	vec3 intensity; };
```

UNIFORMS

Les uniformes sont gérés comme dans le vertex.

```
<UNIFORMS></UNIFORMS>
```

Pour l'utilisation des lumières dynamiques, il est utile de déclarer des tableaux d'uniformes de la taille du nombre de lumière par type.

```
uniform pointLight pointLights[NB_POINTLIGHTS];
uniform spotLight spotLights[NB_SPOTLIGHTS];
uniform directionnalLight directionnalLights[NB_DIRECTIONNALLIGHTS];
```

VARYING

Les varyings sont gérés comme dans le vertex.

```
<VARYING></VARYING>
```

OUTPUT

L'output prend le code à proprement parler du Fragment. Il comporte le *main()* et définit le paramètre gl_FragColor.

```
<OUTPUT></OUTPUT>
```

OBJET 3D

Tous les objets, qu'ils soient au format .js ou .obj doivent être sauvegardés dans le dossier objets.

TEXTURE

Toutes les textures doivent être au format .png et sauvegardées dans le dossier textures.