Documentation
OROGUS

(Outil de Rendu OpenGL de l'UdeS)

Table des matières

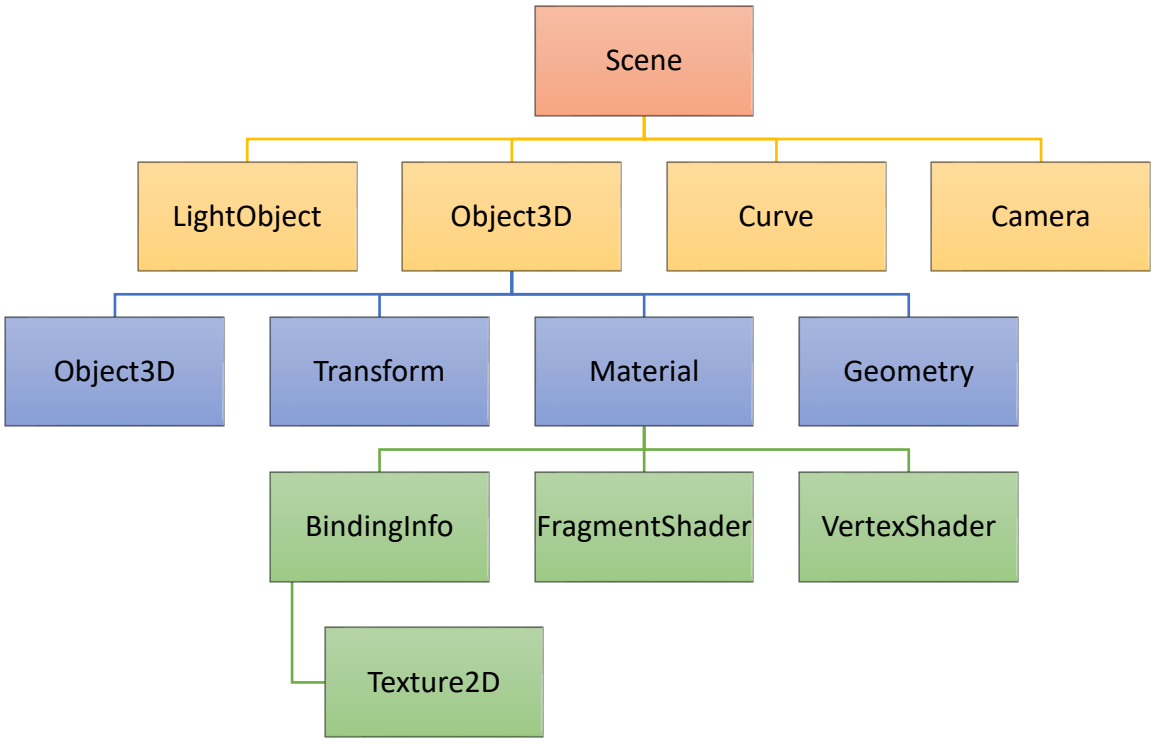
Présentation

OROGUS est le noyau graphique utilisé dans le cours IMN401 — Infographie et jeu vidéo. Ce noyau a été conçu dans l'optique de vous présenter une structure de code qui peut ressembler à ce qui existe dans l'industrie, tout en restant assez simple à suivre et à modifier dans le cadre du cours. Les moteurs de jeu récent sont beaucoup plus complexes et implémentent une grande quantité d'algorithmes de rendu de haute performance qui dépasse le cadre de ce cours. La structure de base et les objets à manipuler dans OROGUS sont cependant assez similaires avec ces moteurs commerciaux pour vous permettre de saisir la structure générale de ces outils et vous permettre de comprendre le pipeline graphique complet pour faire du rendu 3D.

Le présent document se veut un compagnon nécessaire pour effectuer les devoirs du cours. Vous y retrouverez toute l'information pertinente sur la structure du code, les classes et les formats utilisés.

Structure générale

Le code d'OROGUS est divisé de façon à retrouver facilement les classes auxquelles on veut accéder. D'un côté, les ressources à utiliser (textures, géométrie, matériel) et de l'autre la scène et tout ce qui gravite autour. Finalement, tous les types et les classes utilitaires sont situés dans le répertoire « Utilities ». Le diagramme suivant montre les liens entre les différentes classes du noyau graphique.



Une scène est composée d'une caméra (*Camera*), d'une liste d'objet (*Object3D*), d'une liste de courbes (*Curve*) et d'une liste de lumière (*LightObject*). Les lumières peuvent être de trois types, soit une lumière directionnelle (*DirectionalLight*), une lumière ponctuelle (*PointLight*) ou une lumière projecteur (*SpotLight*).

Un objet est composé d'une géométrie (*Geometry*) qui représente un maillage composé de Vertex qui forment des triangles. Un objet possède aussi une transformation (*Transform*) qui indique sa position, sa rotation et sa mise à l'échelle dans le monde. Un objet peut aussi contenir une liste d'objet enfant afin de faire une hiérarchie entre des objets. Finalement, un objet possède aussi un matériel (*Material*). Il est possible pour un objet de ne pas avoir de matériel s'il possède aussi un objet parent. Dans ce cas, le matériel du parent sera utilisé pour l'objet. Le matériel représente l'apparence visuelle d'un objet. Un matériel est composé de deux programmes qui s'exécutent sur le GPU pour calculer la couleur de l'objet, soit un *VertexShader* et un *FragmentShader*. Chacun de ces programmes peut avoir besoin de certaines valeurs spécifiques pour effectuer leur calcul. Chacune de ces valeurs spécifiques est sauvegardée dans le matériel à l'aide d'une information d'attache (*BindingInfo*). Le *BindingInfo* est une structure privée au matériel lorsqu'un ajout d'attache est demandé. Il est possible d'attacher plusieurs types de valeurs différentes sur un matériel. Les plus simples sont les valeurs entières ou réelles, les vecteurs en trois ou quatre dimensions ou encore des couleurs. Il est aussi possible d'attacher des images (*Texture2D*).

Le programme principal s'occupe de créer la fenêtre de rendu et de faire le chargement d'une scène. Il contient aussi la boucle de rendu principale. Dans cette boucle, la méthode de rendu de la scène est appelée. C'est cette méthode qui s'occupe d'appeler les méthodes de rendu des différents objets qui compose la scène. Chaque objet s'occupe de lier le matériel et la géométrie requise pour effectuer le bon rendu. Chaque objet s'occupe aussi d'appeler la méthode de rendu de ses objets enfants afin de respecter la structure en arbre de la scène. La scène s'occupe aussi de faire le rendu des lumières aux fins de visualisation. Chaque matériel s'occupe aussi d'assigner l'ensemble des valeurs requises afin que les programmes de nuanceurs (*shaders*) s'exécutent correctement.

Gestion des ressources

L'ensemble des ressources du noyau graphique est géré par des gestionnaires dédiés afin d'éviter de recharger en mémoire plusieurs fois les mêmes ressources. Il y a trois gestionnaires dédiés et un gestionnaire général afin de simplifier quelques-unes des opérations.

Les géométries sont gérées par le gestionnaire de géométries (*GeometryManager*). Ce gestionnaire fourni des méthodes pour charger et décharger des géométries à partir d'un nom de fichier. Si la géométrie a déjà été chargée précédemment, le gestionnaire retournera un pointeur vers la géométrie originale. Il est important de libérer les ressources qui ne sont plus utilisées en utilisant la méthode « *unload* » du gestionnaire. Cette méthode s'assure de garder le compte des références sur une géométrie particulière et libère la mémoire seulement lorsque la dernière référence a été libérée.

Les textures sont aussi gérées par un gestionnaire (*TextureManager*). Le même principe que le gestionnaire de géométrie s'applique.

Le dernier gestionnaire s'occupe de la gestion des programmes de nuanceurs (*ShaderManager*). Les différents programmes sont utilisés par les matériels. Les matériels ne sont pas gérés par un gestionnaire puisque chaque instance de matériel peut contenir des valeurs particulières pour les « *uniforms* » dans un nuanceur. Le matériel utilise le gestionnaire de nuanceurs pour la gestion de ceux-ci et s'assure que le même programme ne soit pas instancié plus qu'une fois sur le GPU.

Chacun des gestionnaires est implanté sous forme de singleton afin de permettre un accès direct à n'importe quel endroit dans le code. De plus, afin de s'assurer que le compte des références sur les objets est le bon, il est impératif de ne pas créer de ressources directement avec l'opérateur « *new* », mais de passer par les méthodes « *load* » des gestionnaires.

Il existe aussi un gestionnaire qui gère les gestionnaires afin de simplifier l'initialisation des autres gestionnaires lors du démarrage du noyau. Même si l'utilité peut être discutable dans un noyau aussi simple, elle est très utile dans de plus gros moteurs graphiques lorsque les gestionnaires peuvent avoir des dépendances avec d'autres parties du code et que l'initialisation doit se faire dans un ordre particulier.

Scène

Le noyau fourni utile un concept de scène pour gérer les objets et les propriétés. Chaque scène est spécifiée à l'aide d'un fichier écrit en utilisant la notation XML. XML est une notation permettant de définir des données sous forme d'arbre de façon claire et lisible. Chaque information est écrite dans un « élément ». Un élément commence par le caractère « < » suivi du nom de l'élément, d'une liste d'attribut au format « att="valeur" ». La fin de l'élément se termine avec le caractère « > ». Il est ensuite possible d'écrire directement les éléments enfants de cet élément au même format. Lorsque les enfants sont terminés, il faut fermer l'élément parent avec la balise « </nomElement> où « nomElement » est le même nom que la balise parent. Si un élément n'a pas d'enfant, il est possible de le terminer directement en finissant la balise avec la séquence « /> » à la place du « > » standard. Voici un exemple qui spécifie une partie des paramètres d'une caméra dans une scène.

```
<camera>
  <far value="1000"/>
  <near value="0.01"/>
</camera>
```

L'élément caméra n'a pas d'attributs. Il possède deux éléments enfants qui ont chacun un attribut nommé « *value* ». Ces deux éléments ne possèdent pas d'enfants et sont donc terminés directement par le symbole « /> ».

Voici la structure générale d'un fichier représentant une scène :

```
<scene>
  <properties>
    …
  </properties>
  <lights>
    …
  </lights>
  <objects>
    …
  </objects>
  <curves>
    …
  </curves>
</scene>
```

Comme on peut le voir, un élément `scene` contient quatre sous éléments, soit `properties`, `lights`, `objects` et `curves`. L'élément `properties` contient comme éléments enfants différentes propriétés de la scène. Toutes les propriétés sont facultatives et des valeurs par défaut seront utilisées. Tous les chemins d'accès doivent être relatifs à l'endroit où est le fichier scène. Toutes les scènes devraient se retrouver dans le dossier Output.

Élément – properties

Voici une description de l'ensemble des propriétés d'une scène que vous pouvez spécifier et les valeurs par défaut associées si vous ne les spécifiez pas.

`<camera>` : Élément complexe représentant une caméra. L'élément `camera` possède plusieurs sous éléments qui représentent chacun une propriété de la caméra :

```
  <camera>
    <far value="1000"/>
    <near value="0.01"/>
    <fieldOfView value="45"/>
    <position x="0" y="0" z="-5"/>
    <lookAt x="0" y="0" z="0"/>
    <up x="0" y="1" z="0"/>
  </camera>
```

Les valeurs des propriétés `far` et `near` sont spécifiées en mètres. La valeur de la propriété `fieldOfView` est spécifiée en degrés. Les propriétés `position` et `lookAt` sont des points. Chaque composante d'un point est spécifiée en mètre.

`<ambientColor r="0" g="0" b="0">` : Couleur ambiante de base de la scène, au format RGB. Les valeurs sont des nombres réels, compris entre 0 et 1. Pour écrire une valeur entière comprises entre 0 et 255 pour chaque composante, utilisez les attributs `red`, `green` et `blue`.

`<ambientPower x="1" y="1" z="1">` : Puissance de la couleur ambiante de base de la scène. Spécifié sous forme de vecteur avec composante x, y et z avec des valeurs entre 0 et 1.

```
<material>
  …
</material>
```
L'information sur le matériel de base à utiliser sur les objets s'ils n'en définissent pas un explicitement. Si aucun matériel n'est indiqué, il n'y a pas de matériel par défaut. La définition complète d'un matériel est donnée dans la section sur les objets.

```
<transform>
  …
</transform>
```

Les transformations géométriques à appliquer sur chacun des objets de la scène. Ce nœud est optionnel. Si aucune transformation n'est indiquée, la scène reste centrée à l'origine, orientée sur les axes canoniques X, Y et Z. La définition complète des transformations est donnée dans la section sur les objets.

Élément – lights

L'élément `lights` peut contenir une suite d'élément `<light>`. Il peut y avoir trois types de lumière différents à utiliser. Voici la description de chacun des types. La distinction entre les types provient de l'attribut *type* de l'élément `light`.

```
<light type="point">
  <position x="0" y="0" z="0"/>
  <ambient r="0" g="0" b="0"/>
  <diffuse r="0" g="0" b="0"/>
  <specular r="0" g="0" b="0"/>
  <constant value="0"/>
  <linear value="0"/>
  <quadratic value="0"/>
</light>
```
Ce type de lumière représente une lumière ponctuelle (*PointLight*). Les propriétés `ambient`, `diffuse` et `specular` représente respectivement les couleurs ambiante, diffuse et spéculaire de la lumière, représentées comme couleur RGB. Les valeurs sont des nombres réels positifs. Pour écrire une valeur entière comprises entre 0 et 255 pour chaque composante, utilisez les attributs `red`, `green` et `blue`. La propriété `position` représente la position de la lumière dans le monde. La position est un point dont les composantes sont spécifiées en mètre. Les propriétés `constant`, `linear` et `quadratic` représente respectivement les facteurs d'atténuations constants, linéaires et quadratiques. Ces valeurs n'ont pas d'unité.

```
<light type="directional">
  <direction x="0" y="0" z="0"/>
  <ambient r="0" g="0" b="0"/>
  <diffuse r="0" g="0" b="0"/>
  <specular r="0" g="0" b="0"/>
</light>
```
Ce type de lumière représente une lumière directionnelle (*DirectionalLight*). Les propriétés `ambient`, `diffuse` et `specular` représente respectivement les couleurs ambiante, diffuse et spéculaire de la lumière, représentées comme couleur RGB. Les valeurs sont des nombres réels positifs. Pour écrire une valeur entière comprises entre 0 et 255 pour chaque composante, utilisez les attributs `red`, `green` et `blue`. La propriété `direction` représente la direction de la lumière. La direction est un vecteur dont les composantes n'ont pas d'unité. Le vecteur n'a pas à être

normalisé. La direction devrait toujours être spécifiée, car la direction par défaut est le vecteur nul.

```
<light type="spot">
  <position x="0" y="0" z="0"/>
  <direction x="0" y="0" z="0"/>
  <ambient r="0" g="0" b="0"/>
  <diffuse r="0" g="0" b="0"/>
  <specular r="0" g="0" b="0"/>
  <angle value="10"/>
</light>
```

Ce type de lumière représente une lumière de type projecteur (*SpotLight*). Les propriétés `ambient`, `diffuse` et `specular` représente respectivement les couleurs ambiante, diffuse et spéculaire de la lumière, représentées comme couleur RGB. Les valeurs sont des nombres réels positifs. Pour écrire une valeur entière comprises entre 0 et 255 pour chaque composante, utilisez les attributs `red`, `green` et `blue`. La propriété `position` représente la position de la lumière dans le monde. La position est un point dont les composantes sont spécifiées en mètre. La propriété `direction` représente la direction du faisceau. La direction est un vecteur dont les composantes n'ont pas d'unité. Le vecteur n'a pas à être normalisé. La direction devrait toujours être spécifiée, car la direction par défaut est le vecteur nul. La propriété `angle` représente la moitié de l'ouverture du faisceau, en degrés.

Élément – objects

L'élément `objects` peut contenir une suite d'élément `<object>`. Chaque objet définit un ensemble de propriétés qui sont utilisées pour afficher correctement ce dernier dans la scène. Chaque objet contient un sous-élément `geometry` et `material`. Optionnellement, les sous-éléments `transform` et `children` peuvent être présent. L'exemple suivant vous montre la structure générale de définition d'un objet.

```
<object name="Unknown">
  <geometry type="XYZ">
    …
  </geometry>
  <material>
    …
  </material>
  <transform>
    …
  </transform>
  <children>
    <object name="NomEnfant">
      …
    </object>
  </children>
</object>
```

Un élément `geometry` doit contenir un attribut *type* qui peut avoir soit la valeur `forme`, soit la valeur `fichier`. Si le type est `forme`, l'élément `geometry` doit avoir un seul sous-élément nommé `forme`. Ce nouvel élément doit avoir un attribut nommé *type* qui spécifiera le type de forme à

utiliser comme géométrie pour l'objet. Voici la liste des formes possibles et les attributs associés à chaque forme :

```
<forme type="cube" width="1" height="1" depth="1"/>

<forme type="cube" size="1"/>

<forme type="sphere" stacks="10" slices="10" radius="1"/>

<forme type="grid" width="10" depth="10" n="10" m="10" uRepeat="1" vRepeat="1"/>

<forme type="triangle"/>

<forme type="cylinder" topRadius="1" bottomRadius="1" height="1" slices="10"
stacks="10"/>
```

Chacun de ces éléments peut contenir un sous-élément nommé `<color r="1" g="1" b="1"/>` pour spécifier la couleur de la forme si aucune texture n'est utilisée dans les nuanceurs. C'est utile principalement pour les nuanceurs minimaux qui ne font pas d'illumination si on ne veut pas utiliser des matériaux complexes. Chaque attribut permet de donner l'information pertinente aux fonctions trouvées dans la classe *GeometryHelper* afin de créer les bonnes géométries. La forme triangle affiche un triangle en coordonnées homogènes directement. À utiliser pour déboguer seulement avec le bon nuanceur.

Il est aussi possible d'utiliser des modèles plus complexes pour les objets. Ces modèles doivent être des géométries au format .OBJ. Pour les utiliser, il faut que le *type* de la géométrie soit `fichier`. Le sous-élément de l'élément `geometry` doit être spécifié de cette façon :

```
<fichier name="path"/>
```

En plus d'une géométrie, un objet doit spécifier un matériel à utiliser pour faire son affichage correctement. Le matériel est composé d'un `vertexShader` et d'un `fragmentShader`. Si aucun nuanceur n'est spécifié, les nuanceurs par défaut seront utilisés (*BaseVertexShader.vs* pour le *VertexShader* et *BaseColorNoLitFragmentShader.fs* pour le *FragmentShader*).

Un matériel est défini comme ceci :

```
<material>
  <vertexShader name="Shaders/BaseVertexShader.vs"/>
  <fragmentShader name="Shaders/BaseColorNoLitFragmentShader.fs">
    <uniform name="uniformName" type="float" value="0"/>
    <uniform name="uniformName" type="int" value="0"/>
    <uniform name="uniformName" type="vec3" x="0" y="0" z="0"/>
    <uniform name="uniformName" type="vec4" x="0" y="0" z="0" w="0"/>
    <uniform name="uniformName" type="color" r="0" g="0" b="0" a="0"/>
    <texture name="sample2DName" value="path"/>
  </fragmentShader>
</material>
```

Chacun des nuanceurs peut spécifier les valeurs des attributs uniformes et des *samplers* (textures) qu'il utilise. Vous devez spécifier le *nom* de l'attribut dans le nuanceur, son *type* et la *valeur* à lui assigner lors de l'exécution si c'est une valeur uniforme autre qu'une texture. Si c'est une texture

que vous voulez spécifier, vous devez créer un élément `texture` et indiquer le chemin d'accès vers l'image à utiliser. Les éléments `uniform` et `texture` peuvent être mis autant comme sous-élément d'un `vertexShader` ou d'un `fragmentShader`.

Vous pouvez aussi spécifier autant de transformations géométriques que vous le voulez sur un objet afin de le transformer. Chaque transformation doit être spécifiée dans l'élément `transform`. Voici les transformations possibles à appliquer sur un objet :

```
<translation x="0" y="0" z="0"/>

<scale x="1" y="1" z="1"/>

<rotation angle="0" axeX="0" axeY="0" axeZ="0"/>

<rotationX angle="0"/>

<rotationY angle="0"/>

<rotationZ angle="0"/>
```

Les angles sont toujours spécifiés en degrés. La `rotation` représente la rotation d'un *angle* par rapport à un *axe* particulier. Les éléments `rotationX`, `rotationY` et `rotationZ` représentent une rotation d'un *angle* spécifié respectivement autour de l'axe X, l'axe Y et l'axe Z. Les transformations spécifiées sont effectuées dans l'ordre indiqué.

Un objet peut aussi définir une hiérarchie parent-enfant avec d'autres objets. Si c'est le cas, l'élément `children` sera présent dans l'objet. L'élément `children` contiendra une liste d'éléments `objet` qui donneront la définition de l'objet. Il n'est pas nécessaire de spécifier un matériel pour les objets enfants. Si aucun matériel n'est spécifié, celui du parent sera utilisé. Chacune des transformations d'un enfant sera effectuée en lien avec celle de son parent, c'est-à-dire que si le parent se déplace, les enfants le suivront.

Élément — curves

Le dernier type d'élément pouvant être ajouté à une scène est une courbe. L'élément `curves` contient une liste d'élément `curve`. Voici la définition d'un élément `curve` :

```
<curve type="hermite" name="Unknown">
  <color r="1" g="1" b="1"/>
  <precision value="10"/>
  <controlPoints>
    <point x="0" y="0" z="0"/>
  </controlPoints>
</curve>
```

Les courbes peuvent avoir les *types* suivants : *hermite*, *bezier*, *bspline* ou *catmullrom*. Chaque courbe définit une liste de *points de contrôles* dans le monde qui seront utilisés pour calculer la courbe dans son ensemble. La couleur de la courbe, ainsi que la précision du tracé (le nombre de vertex à utiliser) sont aussi spécifiés.

Mathématiques

L'ensemble des valeurs manipulées dans OROGUS sont associées à des unités. Les distances sont spécifiées en mètres, le temps en seconde, les angles en degrés, etc. Les opérations d'algèbres

linéaires sont donc adaptées pour donner les bons résultats. Les classes Vector2, Vector3 et Vector4 sont disponibles et représentent des vecteurs mathématiques respectivement en 2 dimensions, en 3 dimensions et en 4 dimensions. Le type Point3 représente une position dans l'espace en 3 dimensions. Les types Matrix3x3 et Matrix4x4 représentent des matrices carrées à trois ou quatre lignes/colonnes respectivement. Tous ces types sont générique (*template*) sur le type d'unité à utiliser pour les composantes. L'ensemble des opérations mathématiques sont disponibles (multiplication, addition, etc.) si les unités des composantes sont compatibles. Il est possible d'additionner deux vecteurs contenant des unités de distances, mais pas un vecteur de distance et un vecteur de temps.

Commandes

Il y a plusieurs raccourcis clavier disponibles dans OROGUS. Pour réinitialiser la caméra, il suffit d'appuyer sur la touche « Escape ». Pour affiche les objets en fil de fer, il faut faire la touche 2. Pour revenir au mode normal, la touche 1. Pour bouger la caméra, il faut utiliser les touches W, A, S, et D pour tourner respectivement vers le haut, vers la gauche, vers le bas et vers la droite. La touche R permet de recharger le fichier de scène est les nuanceurs afin de rester plus rapidement les modifications apportées à une scène. Finalement, la souris peut aussi être utilisé pour bouger la caméra. En appuyant sur le bouton de gauche, vous pourrez faire tourner la caméra. La roulette de la souris permet de « zoomer » sur le point d'intérêt de la caméra.