Quick3D

Gestion et affichage de scènes 3D

Ce document présente les fonctionnalités de Quick3D. Les concepts présentés n’entrent pas dans les détails de l’implémentation. Pour plus de détails, voir les commentaires dans les fichiers de définition des classes et le code lui-même.

2015

Guillaume DARIER

ATERMES

08/10/2015

Table des matières

[1 Glossaire 3](#_Toc434591352)

[2 Introduction 4](#_Toc434591353)

[3 Unités 4](#_Toc434591354)

[3.1 Conventions des systèmes de coordonnées 4](#_Toc434591355)

[3.1.1 Système de coordonnées affines local 4](#_Toc434591356)

[3.1.2 Système de coordonnées géocentriques 4](#_Toc434591357)

[3.1.3 Système de coordonnées sphériques 5](#_Toc434591358)

[4 Composants 6](#_Toc434591359)

[4.1 Boite englobante (Bounding box) 6](#_Toc434591360)

[4.2 Propriétés de position, rotation et mise à l’échelle 6](#_Toc434591361)

[4.3 Propriétés de parenté 6](#_Toc434591362)

[4.4 Propriétés physiques 7](#_Toc434591363)

[4.5 Rendu 3D 7](#_Toc434591364)

[4.5.1 La classe de maillage de polygone : CMesh 7](#_Toc434591365)

[4.5.2 La caméra et le viewport 8](#_Toc434591366)

[4.5.3 La pyramide de visualisation (Frustum) 9](#_Toc434591367)

[4.5.4 Méthode de rendu : CCamera::render() 10](#_Toc434591368)

[5 Terrain 11](#_Toc434591369)

[5.1 Le terrain géo-référencé 11](#_Toc434591370)

[5.2 La grille d’altitudes 12](#_Toc434591371)

[6 Matériaux 13](#_Toc434591372)

[6.1 Généralités 13](#_Toc434591373)

[6.2 Paramètres génériques 13](#_Toc434591374)

[6.3 Textures 13](#_Toc434591375)

[6.4 Matériaux de tuile satellite 14](#_Toc434591376)

[7 Shaders 15](#_Toc434591377)

[8 Gestionnaire de scènes 17](#_Toc434591378)

[8.1 L’usine de composants 17](#_Toc434591379)

[8.2 Le lecteur de scènes au format XML 17](#_Toc434591380)

[9 Rendu de matrice panoramique 21](#_Toc434591381)

[10 Cas typique d’utilisation de Quick3D 23](#_Toc434591382)

# 1 Glossaire

|  |  |
| --- | --- |
| Chunk | Morceau de terrain |
| Coordonnée affine | Coordonnée dans l’espace affine définie par trois éléments : X, Y et Z. |
| Coordonnée sphérique | Coordonnée sur un sphéroïde définie par trois éléments : Latitude, Longitude et Altitude. |
| Frustum | Pyramide de visualisation d’une caméra |
| GPU | Graphics Processing Unit. Processeur mathématique multitâches dédié principalement aux graphismes 3D. |
| Height field | Grille d’altitudes. |
| Mesh | Maillage de polygones. |
| NOLL | North-Oriented Local-Level. Un repère dont l’axe avant pointe vers le nord géographique, l’axe dessus pointe vers l’espace et l’axe droite est perpendiculaire aux deux premiers. |
| Quad-key | Clé qui identifie une tuile d’image satellite pour une géo-localisation et un niveau de détails donné. |
| Shader | Programme en C exécuté sur un GPU. Sert à calculer la couleur finale d’un pixel dans un rendu de scène 3D. |
| Shadow map | Une carte de profondeur servant à déterminer si un fragment d’objet (pixel) est illuminé ou non par une lumière, autrement dit s’il est dans l’ombre. |
| WGS84 | World Geodesic System 1984. Permet de représenter un ellipsoïde modélisant la terre. |

# 2 Introduction

**Quick3D** est un moteur de traitement et d’affichage d’objets en trois dimensions.

Il permet l’affichage en temps réel de scènes comportant des composants 3D ou d’exécuter des calculs particuliers sur base d’une scène 3D, comme le calcul d’une matrice panoramique de profondeur.

# 3 Unités

Pour les distances, l’unité est le mètre.  
Pour les angles hors géo-localisations, l’unité est le radian.  
Pour les angles de géo-localisation (latitude, longitude), l’unité est le degré.

## 3.1 Conventions des systèmes de coordonnées

Il existe trois systèmes de coordonnées dans **Quick3D**.

Les coordonnées affines locales, les coordonnée géocentriques et les coordonnées sphériques (géo-référencées).

### 3.1.1 Système de coordonnées affines local

La coordonnée affine est représentée sur trois axes. Il s’agit du vecteur en trois dimensions X, Y, et Z. La classe associée à ce système est **Vector3** dans **AtsUtilities**.

La convention utilisée pour les trois coordonnées est la suivante :

* X : Cet axe pointe vers la droite du point de référence
* Y : Cet axe pointe vers le haut du point de référence
* Z : Cet axe pointe vers l’avant du point de référence

C’est la convention dite du peintre, en comparaison de la convention de l’architecte dans laquelle Y pointe vers l’avant et Z vers le haut.

Pour transformer ces vecteurs, la classe **Matrix4** dans **AtsUtilities** permet de générer des matrices de translation (**Matrix4::MakeTranslation()**), de rotation (**Matrix4::MakeRotation()**) et de changement d’échelle (**Matrix4::MakeScale()**). Les matrices peuvent être multipliées entre elles afin d’accumuler des transformations.

Le repère peut être soit local à un point géo-référencé, soit local à un composant comme dans le cas d’un composant qui a un parent.

### 3.1.2 Système de coordonnées géocentriques

Ce système à pour origine le centre de la terre et s’exprime en coordonnées affines (X, Y, Z).  
Ainsi, l’intersection du méridien 0 et de l’équateur se trouve en <0, 0, -RT> ou RT = rayon de la terre à l’équateur.

Il sert principalement d’intermédiaire entre le système sphérique et le système affine local.

### 3.1.3 Système de coordonnées sphériques

Ce modèle est basé sur **WGS84**, ce qui permet d’afficher un modèle terrestre précis. La classe associée à ce système est **Geoloc** dans **AtsUtilities**.

La convention utilisée pour chacun des composants de la classe est la suivante :

* Latitude : Valeur de latitude sur la sphère en degrés
* Longitude : Valeur de longitude sur la sphère en degrés
* Altitude : Altitude sur la sphère en mètres

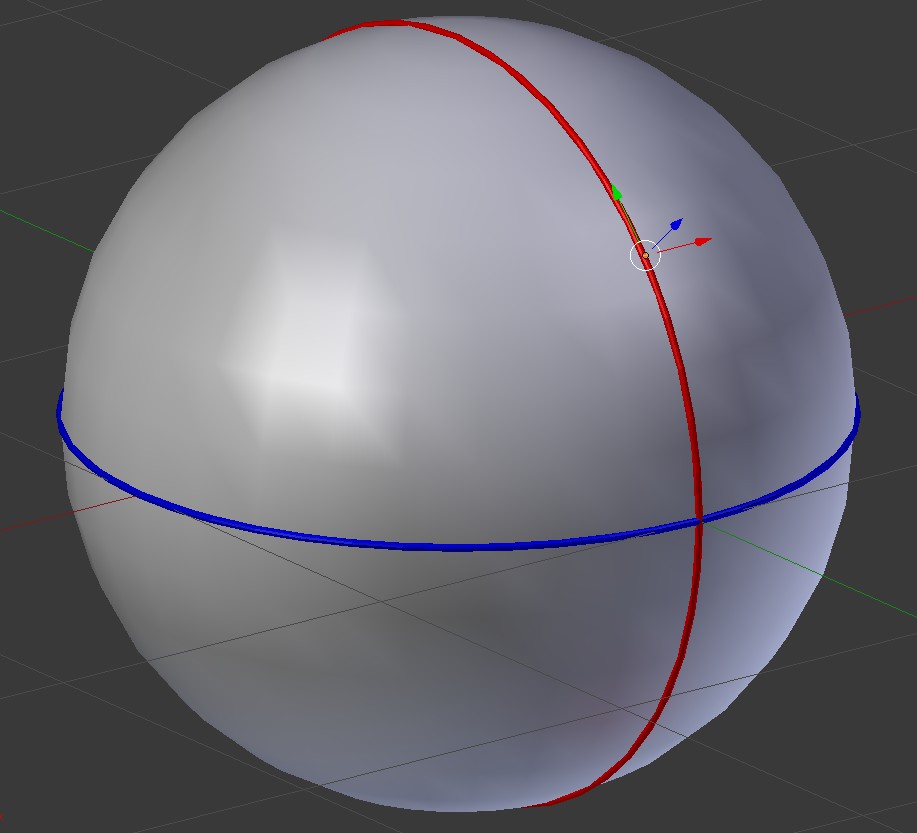
La classe **Geoloc** propose les méthodes nécessaires pour exécuter des changements de repère :

* Conversion d’une coordonnée sphérique en coordonnée géocentrique : **Geoloc::toVector3()**
* Conversion d’une coordonnée géocentrique en coordonnée sphérique : **Geoloc::Geoloc(const Vector3& vPosition3D)**
* Conversion d’une coordonnée sphérique en coordonnée affine locale, c'est-à-dire relative à une autre coordonnée sphérique : **Geoloc::toVector3(const Geoloc& gReference)**

Elle contient également une méthode qui permet de récupérer les axes NOLL (North-Oriented Local-Level) d’une géo-localisation donnée.  
Ceci est un ensemble de trois vecteurs géocentriques qui définissent les axes suivants :

* L’axe de l’avant (front) : Cet axe pointe vers le nord géographique
* L’axe du dessus (up) : Cet axe part du centre de la terre et passe par la geoloc de laquelle on demande l’axe NOLL
* L’axe de droite (right) : Cet axe est perpendiculaire aux deux précédents

L’axe NOLL est utilisé entre autre pour des calculs de physique sur les objets, comme la mise à jour de sa translation et rotation à partir de ses vecteurs d’accélération et d’accélération angulaire. Il est utilisé également pour connaitre le vecteur tangent des sommets de terrain (tangent par rapport à la surface du globe).



Un axe NOLL, l’axe vert (Z) pointe vers le nord.

# 4 Composants

Le composant (**CComponent**) est la classe de base de tout objet existant dans une scène de Quick3D, qu’il soit visible ou non.

Les propriétés du **CComponent** permettent de gérer la géo localisation, la rotation, les liens de parenté et la physique d’un objet 3D.  
Il contient également les définitions de méthodes propres à son affichage.

## 4.1 Boite englobante (Bounding box)

Tout composant possède une boite englobante. Celle-ci définit les limites de l’objet et est alignée sur les axes du monde. Ce type d’objet est souvent appelé une AABB (Axis Aligned Bounding Box) : boite englobante alignée sur les axes.

Elle permet de déterminer rapidement si un objet est visible par une caméra ou s’il est potentiellement traversé par un segment (**CRay**), comme dans le cas du ray-tracing.

Cette boite est accessible via la méthode **CComponent::getWorldBounds()** pour la version qui comporte les transformations de l’objet, ou **CComponent::getBounds()** pour la version locale à l’objet.

## 4.2 Propriétés de position, rotation et mise à l’échelle

Pour définir la position d’un composant, il existe une méthode de positionnement dans un repère affine local (X, Y, Z) et une méthode de positionnement dans le repère sphérique (**Geoloc**).

Si l’objet est un objet racine, à savoir qu’il n’a pas de parent, on utilisera le repère sphérique avec **CComponent::setGeoloc()** et **CComponent::getGeoloc()**.  
Si l’objet est un objet non-racine (qui a un parent), on utilisera le repère affine local avec **CComponent::setOriginPosition()** et **CComponent::getOriginPosition()**.

Les méthodes **CComponent::setPosition()** et **CComponent::getPosition()** sont réservés à l’animation et peuvent être vues comme des offsets ajoutés au valeurs « d’origine ».

De la même manière, les rotations et mises à l’échelle seront réglées avec **CComponent::setOriginRotation()** et **CComponent::setOriginScale()**.

## 4.3 Propriétés de parenté

Un composant peut être attaché à un autre, c’est-à-dire qu’il héritera des transformations de son parent. La propriété du nom du parent sert surtout lors de la lecture d’un fichier XML de scène : elle permettra de retrouver les composants par nom lorsque l’attachement se fera réellement.

La méthode **CComponent::setParent()** sert à définir le parent d’un objet.

Lorsque que la matrice de transformation d’un composant est calculée (d’après les vecteurs de translation, rotation et échelle), celle des enfants du composant est calculée récursivement.

Dans certain cas, il peut être nécessaire de lier un composant à un autre pour pouvoir les associer dans l’arbre des composants d’une scène, mais sans que le composant hérite des transformations du parent. C’est le cas des terrains, qui sont construits sous la forme d’un quad-tree mais sont positionnés de manière absolue par rapport au centre de la terre.  
Si l’on souhaite qu’un composant avec parent n’hérite pas des transformations de ce dernier, on appellera la méthode **CComponent::setIgnoreParentsForTransform(bool)**.

## 4.4 Propriétés physiques

Quick3D intègre des calculs simples de physique sur les composants.  
Au niveau des collisions, le terrain est géré de manière précise pour que les véhicules puissent y rouler de manière réaliste. Pour ce qui est des collisions entre objet, seules les collisions par sphère sont gérées, même si un enum permet de définir des collisions par boite ou par mesh.

Un composant doit avoir une masse. On la règle, en kilogrammes, avec **CComponent::setMass\_kg()**.

Pour faire bouger un composant, une force ou un couple devra être appliqué à celui-ci.  
Pour appliquer une force, on appellera **CComponent::addForce\_kg()** ou **CComponent::addLocalForce\_kg()** qui comme leur nom l’indique, attendent des valeurs exprimées en kilogrammes.  
Pour appliquer un couple, on appellera **CComponent::addTorque\_kg()** ou **CComponent::addLocalTorque\_kg()**.

Un composant peut être exclu des calculs de physique en utilisant **CComponent::sleep()**, ou inclus en appelant **CComponent::wakeup()**. De la même manière, il peut être exclu ou inclus des collisions avec **CComponent::setCollisions()**.

Etant donné que les collisions ne sont pas gérées de manière précise, un modèle réaliste de véhicule terrestre n’est pas possible. La rotation d’un tel véhicule est calculée simplement en cherchant l’altitude du sol sous ses quatres roues et en prenant une moyenne de ces altitudes sur deux axes pour orienter le véhicule.

## 4.5 Rendu 3D

### 4.5.1 La classe de maillage de polygone : CMesh

Un composant doit hériter de **CMesh** pour pouvoir être rendu (pour affichage temps réel ou calcul de matrice), ou implémenter les méthodes de rendu de **CComponent** avec des instructions **OpenGL**.

Les primitives géométriques de **Quick3D** (sphère, cône, boite) héritent de **CMesh** et construisent leur maillage dans leur constructeur respectif.

Les objets chargés depuis le disque sous la forme d’un **.obj** sont des **CMesh**. C’est le **COBJLoader** qui se charge de les lire. Il est relativement aisé de créer un loader pour un autre format d’objet en copiant le comportement de **COBJLoader**.

Afin de pouvoir partager des maillages (comprendre que plusieurs objets dessinent le même maillage) une classe intermédiaire peut être utilisée : **CMeshInstance**. Elle contient simplement une référence vers un **CMesh**. Elle est notamment utilisée par le **CAutoTerrain** pour afficher des éléments de végétation.

Le manager de ressources (**CRessourcesManager**) retourne un **CMeshInstance** dans sa méthode **share()**. Cela permet de réutiliser le même maillage dans plusieurs objets une fois qu’il a été chargé. De la même manière, un matériau peut être partagé via le manager de ressources.

### 4.5.2 La caméra et le viewport

Pour faire un rendu, Quick3D a besoin de deux éléments importants (hormis les composants visuels) :

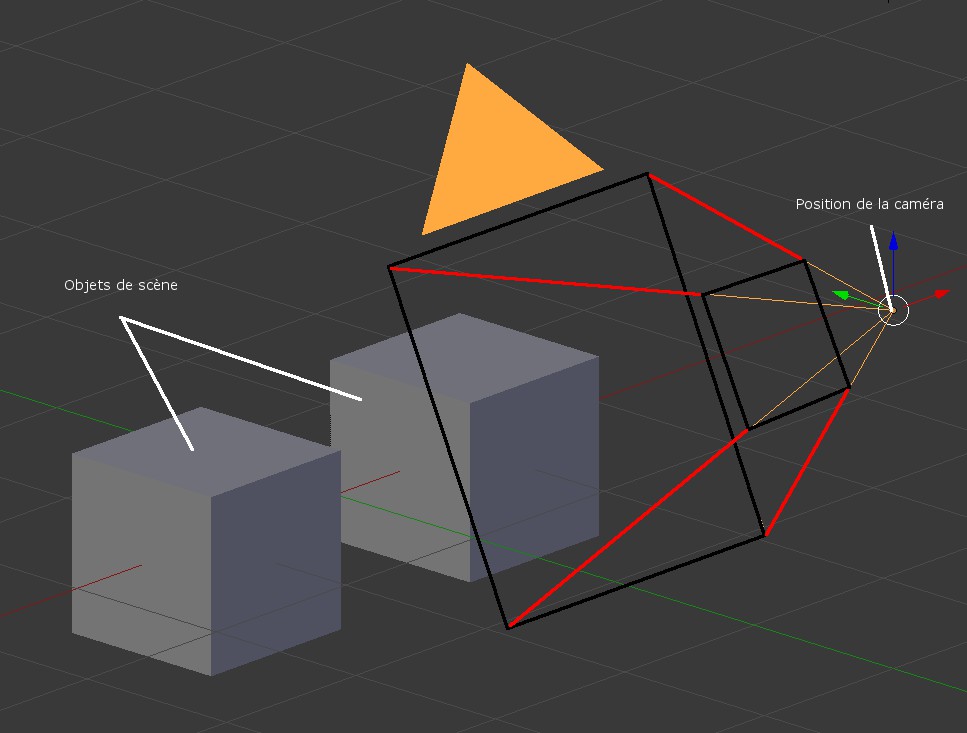
* Une caméra, qui va définir le point de vue, les matrices de transformation et de projection et la pyramide de visualisation (Frustum)
* Un viewport, qui est une surface 2D. La plupart du temps, le viewport fait la taille exacte de l’image finale de rendu. Dans certains cas, on veut afficher plusieurs vues de la scène dans une même image, comme dans Hardware Simulator où l’on visualise deux caméras simultanément. D’où l’utilité du viewport.

Ces deux éléments doivent donc absolument être créés et configurés avant tout rendu.

### 4.5.3 La pyramide de visualisation (Frustum)

Cette pyramide représente l’ensemble des plans formés par la caméra.  
Rappel : Un plan en 3D est défini par 3 points dans l’espace, ou par un vecteur normal au plan et une distance. Dans notre cas, c’est la deuxième méthode qui est employée.  
Soit Vc la position de la caméra, V1 le coin supérieur gauche de la « fenêtre » de la caméra, V2 le coin supérieur droit de la fenêtre, V3 le coin inférieur gauche de la fenêtre, V4 le coin inférieur droit de la fenêtre, les plans sont définis par :

* Plan supérieur : Vc, V2, V1
* Plan inférieur : Vc, V3, V4
* Plan droit : Vc, V4, V2
* Plan gauche : Vc, V1, V3
* Plan proche : Une distance minimum avec le vecteur inverse de direction de la caméra
* Plan lointain : Une distance maximum et le vecteur de direction de la caméra



*Le viewing frustum d’une caméra  
Axe rouge = X, axe vert = Z, axe bleu = Y*

La méthode **CCamera::computeFrustum()** calcule ces plans.

Ne seront effectivement dessinés que les objets contenus partiellement ou entièrement dans le frustum.

### 4.5.4 Méthode de rendu : CCamera::render()

Cette méthode exécute le rendu d’une scène.  
Voici un résumé de ses actions :

* Préparation des flags **OpenGL**
* Doit-on faire un rendu des ombres portées ? (paramètre shader quality de la scène)
  + Oui :
    - Y a t-il au moins une lumière dans la scène ?
      * Oui :
        + Indication aux shaders que l’on fait du rendu d’ombre portée
        + La lumière est considérée comme une caméra
        + Changement de position de la lumière pour la placer à une bonne distance de la caméra (pour que l’ombre portée couvre une zone suffisamment grande)
        + Préparation des matrices de transformation et de projection
        + Création du frustum de la lumière
        + Préparation du viewport et des flags **OpenGL**
        + Rendu de chaque objet visible de la scène
        + Restauration de la position d’origine de la lumière
        + Indication aux shaders que le rendu d’ombre portée est terminé
* Préparation du viewport **OpenGL** (à ne pas confondre avec le viewport **Quick3D**)
* Dans le cas où on gère les ombres portées, création d’une matrice de transformation et de projection pour les ombres (qui sera transmise aux shaders)
* Récupération FOV caméra et distances minimum et maximum de rendu (pour créer le frustum)
* Préparation des matrices de transformation et de projection
* Préparation du contexte de rendu qui sera transmis aux objets rendus
* Mise à jour des objets avant rendu selon le contexte de rendu
* Rendu de chaque objet selon le contexte de rendu (appel de leur méthode **paint()**)
* Restauration des flags **OpenGL**

Le paramètre « shader quality » de la scène est très utilisé dans les shaders. Il va déterminer si on va utiliser ou non les textures de couleur diffuse, si on va ou non faire du rendu de nuage, si la mer va être rendue de manière basique ou avec réflexions, etc…

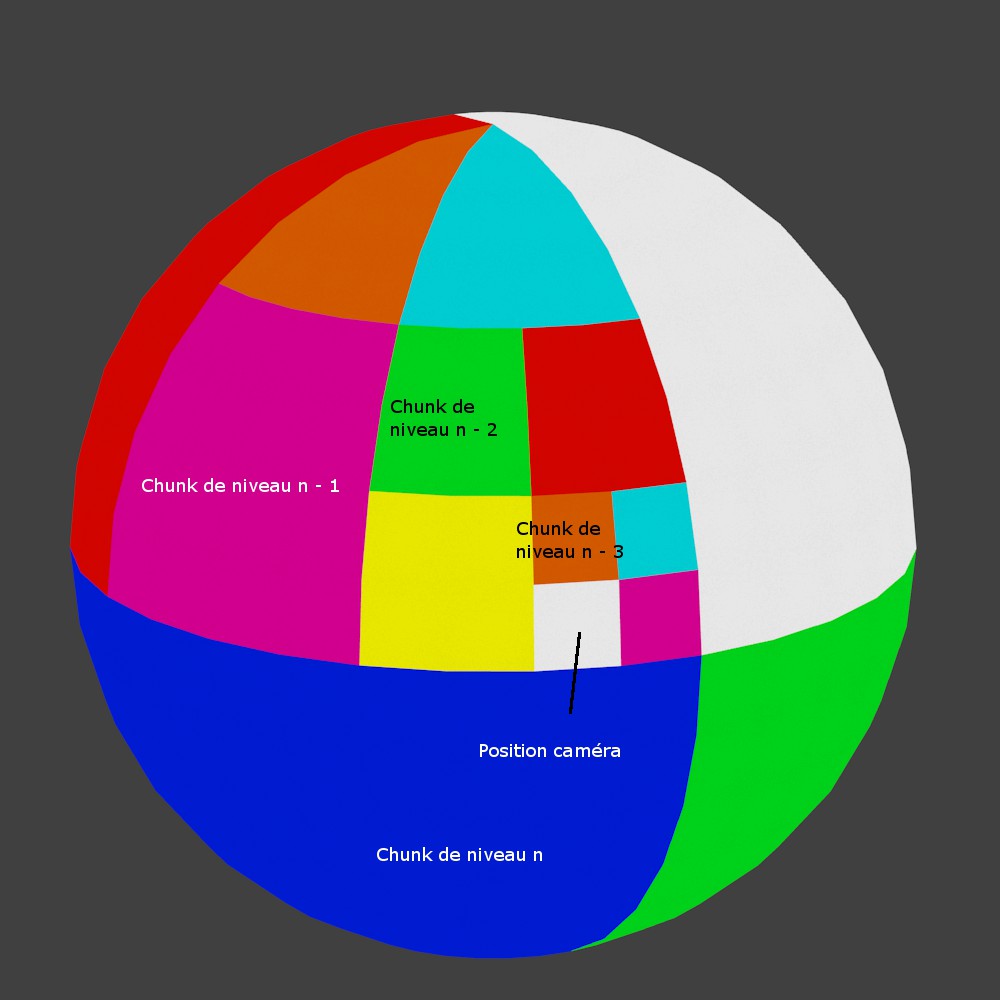
Les ombres portées sont calculées de manière à créer ce qu’on appelle une « shadow map ».  
Une shadow map est en réalité une carte de profondeur. C’est un rendu de la scène en prenant comme point de vue la position d’une lumière.  
Lorsque le rendu final est executé, la position du pixel d’un objet est transférée dans le repère de la shadow map (repère défini par sa matrice de transformation et de projection), et ainsi la distance stockée dans la shadow map permet de savoir si le pixel est « caché » par rapport à la lumière, d’où l’ombre portée.

# 5 Terrain

## 5.1 Le terrain géo-référencé

La classe gérant l’affichage d’un globe est **CAutoTerrain**.

Un **CAutoTerrain** peut être vu comme un arbre dont chaque feuille est une portion de terrain du globe. Chaque feuille, si nécessaire, peut être subdivisée en quatre autres feuilles qui contiendront des versions plus définies du terrain, et ce jusqu’à atteindre le niveau 0, au sol. Le niveau maximum est celui visible depuis l’espace, à une certaine distance.



*Représentation du découpage du globe en quad-tree.  
N représente le niveau maximum de subdivision.*

La classe **CWorldChunk** représente une portion de terrain. Le chunk racine est le globe entier. Pour générer les feuilles du chunk racine, le globe est divisé en quatre, et ainsi de suite à mesure que l’on descend dans l’arbre.

Le chunk est subdivisé si la caméra est suffisamment proche de lui, auquel cas on a besoin de plus de définition. Lorsque ses quatre chunks enfants sont prêts à l’affichage, il efface son propre terrain. Afin de préserver la mémoire, les chunks doivent être « nettoyés » régulièrement. A chaque fois qu’un chunk est utilisé pour affichage, sa date de dernière utilisation est mise à jour. Si cette date devient trop ancienne, le chunk est vidé. Si les quatres sous-chunks d’un chunk sont vides, celui-ci les détruit et reconstruit son propre terrain si nécessaire.

Tous les **CMesh** des chunks ont le même nombre de polygones : le nombre de sommets d’un chunk est paramétrable au niveau du **CAutoTerrain**. Ce qui définira la résolution d’une zone est simplement le nombre de chunks affichés.

Lorsque le chunk se construit, les géo-localisation de ses extrémités peuvent être modifiées par le matériau qu’il utilise. Dans le cas d’un **CTiledMaterial** par exemple : ce matériau télécharge des tuiles qui utilisent la projection de Mercator. Or un chunk est construit de manière linéaire en latitude et longitude. Pour que les tuiles satellite soient placées au bon endroit, le matériau calcul un décalage en latitude et longitude pour le chunk, via la méthode **CMaterial::transformGeoloc()**.

Un chunk est constitué d’un **CMesh** pour le sol, éventuellement un **CMesh** pour l’eau, et peut contenir d’autres composants comme de la végétation.  
Lorsqu’il est créé, son relief est généré soit immédiatement, soit dans un **CWorke**r (une thread), en fonction des paramètres de son **CAutoTerrain**. Pour générer le relief, il fait appel à un **CHeightField** (voir plus bas).  
Dans le cas d’une génération de matrice panoramique, le terrain est généré immédiatement, car les données devront toutes être présentes lors des appels de méthode de rendu.

## 5.2 La grille d’altitudes

Pour avoir du relief, un terrain a besoin d’une source de données. Cette source de données, dans **Quick3D**, est appelée **CHeightField**.

Un « height field » est une grille d’altitudes.  
Sa fonction principale est de fournir une altitude pour un point donné. Il peut également fournir d’autres informations comme la rigidité du terrain à l’endroit spécifié.  
Il peut être procédural. Dans ce cas, des fonctions génèreront les altitudes à l’endroit demandé.  
Ou il peut être basé sur un fichier de modèle numérique de terrain (MNT), comme le SRTM ou le BIL.  
Il pourrait aussi être basé sur un serveur fournisseur d’altitudes, les possibilités sont infinies.

Concernant le terrain procédural, la classe **CGeneratedField** est utilisée. Elle fait appel à une **CGenerateFunction** qui peut être lue dans le fichier XML de définition de scène.  
Une **CGenerateFunction** peut être une opération arithmétique simple ou faire appel à des fonctions de bruit de Perlin, d’érosion, etc…

# 6 Matériaux

## 6.1 Généralités

Les matériaux définissent l’apparence d’un objet. La classe **CMaterial** est la base de toute classe de matériau.

Cette classe, ainsi que ses descendantes, est responsable de transférer des paramètres de rendu à un shader (voir plus bas) via des instructions du wrapper **OpenGL** de **Qt**. Elle le fait dans sa méthode **activate()**. Cette méthode est appelée à chaque fois qu’une instance de matériau est utilisée pour le rendu d’un **CMesh**.

La classe gère également les éventuelles textures du matériau.

## 6.2 Paramètres génériques

Ambient : Couleur ambiante du matériau. La couleur visible sans aucune source lumineuse.  
Diffuse : Couleur diffuse du matériau. C’est celle qui est propagée dans toutes les directions par une source lumineuse.  
Specular : Couleur spéculaire du matériau. On peut voir ça comme l’éclat d’un objet. C’est une couleur réfléchie sous un certain angle entre œil <-> objet <-> lumière.  
SelfIllumination : C’est un facteur qui multiplie la couleur diffuse. Par défaut, il est à zéro.  
Shininess : Le facteur de réflexion spéculaire. Plus il est élevé, plus l’objet est brillant, et plus le reflet spéculaire observé est petit.  
Reflection Taux de réflexion de l’environnement (à savoir le ciel)  
ReflectionSteepness Il agit un peu comme le shininess du spéculaire. Plus il est élevé, plus la réflexion nécessitera un angle aigu entre œil <-> objet <-> lumière pour être visible. Permet entre autre de faire une réflexion réaliste pour l’eau.  
IRFactor Taux d’illumination infrarouge.  
DiffuseTextures Liste de textures utilisées pour la couleur diffuse (elles sont mixées via des facteurs à chaque sommet de polygone). Elles sont prioritaires sur la couleur diffuse.

## 6.3 Textures

Les textures sont des images bitmap qui définissent (entre autre) la couleur d’un objet. La plupart du temps, une texture servira à définir la couleur diffuse (diffuse map), mais elle peut aussi servir à définir le taux de lumière spéculaire (specular map) ou encore le relief d’un matériau (bump map et normal map). Actuellement dans **Quick3D**, seule la diffuse map est gérée par les shaders.

La texture est entièrement gérée par la clase **CMaterial**. Elle peut provenir du disque ou des ressources de l’exécutable.

Si l’on souhaite simplement déclarer une texture de couleur diffuse, on appellera **CMaterial::addDiffuseTexture()**.

Si l’on souhaite créer une texture pour usage particulier, on appellera **CMaterial::createTexture()**, qui prend en entrée une **QImage**.

Des matériaux animés peuvent être développés, ils se serviront alors de la propriété statique **CMaterial::m\_dTime()** que l’application fera évoluer dans le temps.

## 6.4 Matériaux de tuile satellite

La class **CTiledMaterial** est un matériau particulier qui télécharge à la volée des tuiles depuis un fournisseur HTTP. Actuellement, seul le fournisseur **Bing** est supporté.

Le principe est de calculer une « Quad key » qui est l’interprétation directe d’une géo-localisation et d’un niveau de détail donné. On obtient un numéro qui représente une tuile sur le serveur **Bing**. Une demande de téléchargement est faite si la tuile n’existe pas en cache disque. Tant que la tuile n’est pas là, on se contente d’afficher du blanc. Dès que la tuile est disponible, le matériau l’utilise en tant que texture diffuse via sa classe mère **CMaterial**.

# 7 Shaders

Un shader est un programme C qui s’exécute sur un GPU.  
Sa tâche est simple : calculer une couleur pour un pixel donné.

Pour chaque triangle composant la scène, **OpenGL** va exécuter la séquence suivante :

* Calcul des coordonnées écran des sommets du triangle ainsi que toute valeur utile pour un sommet de triangle (vecteur normal par exemple)
* Pour chaque sommet du triangle :
  + Appel du « Vertex shader » fourni par l’application
    - Le shader calcule les informations dont il a besoin pour chaque sommet, lesquelles vont être linéairement interpolées par **OpenGL** dans la séquence suivante  
      (Eclairage au niveau du sommet, déplacement du sommet selon un algorithme pour former des vagues, …)
* Pour chaque ligne horizontale (scan line) du triangle :
  + Appel du « Fragment shader » fourni par l’application
    - Le shader dispose des informations calculées dans le Vertex shader
    - Le shader calcul la couleur du pixel et la retourne

Quand un Vertex shader est appelé, il dispose d’autant d’informations sur le pixel que nécessaire. Ces informations sont fournies par le code de **Quick3D** via l’API **OpenGL** de **Qt**.  
Ces informations sont, entre autres :

* La position absolue du sommet
* Le vecteur normal au sommet
* L’altitude du sommet
* La position absolue de la caméra
* L’altitude de la caméra
* Les positions et couleurs des lumières
* Les attributs de matériau du sommet et du polygone associé
* Les coordonnées de texture du sommet
* Etc…

En utilisant toutes ces informations, le Fragment shader va calculer la couleur diffuse de l’objet au niveau du pixel, sa couleur spéculaire (reflets de lumière), sa réflexion (reflet de ciel), sa couleur ambiante et générer une couleur finale qu’**OpenGL** va stocker dans l’image finale. Le shader peut également faire appel à une série d’algorithmes pour générer des perturbations de la surface, comme pour l’océan.

Dans **Quick3D** il n’existe actuellement qu’un seul shader qui gère tous les effets des matériaux. Il est décomposé en deux fichiers :

* VS\_Standard.c Le « Vertex shader »
* FS\_Standard.c Le « Fragment shader », appelé dans d’autres API « Pixel shader »

En entête de ces fichiers sont définies des variables :

* « Uniform » : Elles sont globales et renseignées par :
  + **CMaterial** pour les propriétés du matériau de l’objet dessiné
  + **CMesh** pour les matrices de transformation et projection
  + **CGLWidgetScene** pour les propriétés d’environnement (lumières, brume, etc…).
* « Varying » : Ce sont des valeurs calculées par le « Vertex shader » et elles vont être interpolées par **OpenGL** pour chaque pixel du polygone dessiné pour nourrir le « Fragment shader ».
* « Attribute » : Ces données proviennent des buffers de vertex et de polygone préparés dans une instance de la classe **CMesh**. Elles contiennent les attributs de chaque sommet de polygone ainsi que les indices de sommet de chaque polygone.

Pour le code en lui-même, se référer à la documentation de GLSL (GL Shading Language) et aux formules mathématiques disponibles sur Internet :

* Eclairage
* Bruit de Perlin
* Turbulence
* Ray marching
* Brume
* Etc…

Références :

Référence GLSL : <https://www.opengl.org/documentation/glsl/>  
Explicatif du pipeline de rendu : <https://open.gl/drawing>  
Exemples de shaders : <http://glslsandbox.com/>

# 8 Gestionnaire de scènes

## 8.1 L’usine de composants

Les différents types de composants peuvent être instanciés par nom de classe. C’est la **CComponentFactory** qui est responsable de cette tâche.

Pour rendre un composant instanciable via un fichier de scènes, il doit être connu par la factory. Pour ce faire, on appellera la méthode **CComponentFactory::registerComponent()**.

Note :  
Une application exploitant Quick3D doit faire un appel et un seul à CComponentFactory::registerCoreComponents(). Ceci permet à la factory de connaître tous les composants de base du moteur.

## 8.2 Le lecteur de scènes au format XML

Le **CComponentLoader** est responsable du chargement d’une scène via un fichier XML.

Le principe de chargement est le suivant :

* Pour chaque nœud de composant de la scène :
  + Instancier le composant avec son nom de classe via la **CComponentFactory**
  + Définir le nom de parent du composant
  + Appeler la méthode **loadParameters()** du composant : ce dernier sera responsable de la lecture de tous ses paramètres.
* Après cette première passe, pour chaque composant en mémoire :
  + Appeler la méthode **solveLinks()** du composant afin qu’il puisse se lier à son parent

Voici un bref descriptif des nœuds possibles dans le fichier de scène :

<Scene>  
 *Balise de début de la scène, racine du document.* <Components>  
 *Balise de début de la liste de composants.* *Exemple de composant : Terrain.*  
 <Component Name="Terrain" Class="CAutoTerrain">  
 *Balise de début de composant.*  
 *Name : Nom du composant.*  
 *Class : Classe du composant (voir la* ***CComponentFactory****).*  
 <Parameters>  
 *Balise de début des paramètres de terrain.*  
 <General Levels="18" Resolution="31"/>  
 *Levels : Nombre de niveaux du terrain.*  
 *Resolution : résolution d’un chunk en nombre de vertex.*  
 <Material Type="bing"/>   
 *Type de matériau.*  
 *bing : Tuiles téléchargées sur le serveur de* ***Bing.***  
 *Si le type de matériau n’est pas spécifié, le terrain utilise un matériau  
 par défaut à base de textures mixées (texture blending).*  
 <Height Type="bil">  
 *Type de height field pour les altitudes.*  
 *bil : Instancie un* ***CBILField*** *qui gère des fichiers* ***.bil.***  
 *srtm : Instancie un CSRTMField qui gère les fichiers* ***.asc*** *et* ***.bin.*** *hgt : Instancie un* ***CHGTField*** *qui gère les fichiers* ***.hgt.*** *Pour obtenir un terrain généré, ne pas spécifier de type et ajouter  
 ce qui suit :*  
 <Value Type="xxx">  
 *Se référer à la classe* ***CGeneratedField*** *pour les types.*  
 *Exemple : Pour générer un terrain plat :*  
 *<Value Type="Constant" Value="0.0"/>*  
 </Height>  
 </Parameters>  
 *Balise de fin des paramètres de terrain.*  
 </Component>  
 *Balise de fin d’un composant.* Exemple de composant : Objet mobile sans parent.  
 <Component Name="Visitor" Class="CTerrestrialVehicle">  
 *Name : Nom du composant.*  
 *Class : Classe du composant (voir la* ***CComponentFactory****).*  
 <Geoloc Latitude="19.2380" Longitude="10.8965" Altitude="0.0"/>  
 *Géo-localisation de l’objet.*  
 <Rotation x="0.0" y="0.0" z="0.0"/>  
 *Rotation sur les trois axes du repère NOLL de l’objet.*  
 <Mesh Name="PVP.obj"/>  
 *Nom du mesh en ressources pour afficher cet objet. Actuellement,  
 pas de possibilité de lire un mesh depuis le système de fichiers.*  
 <IR Factor="0.8"/>  
 *Facteur de radiation infrarouge*  
 <Physics SpeedMS="4.0" TurnSpeedDS="30.0" Sleep="0" CollisionsSleep="0"/>  
 *Propriétés physiques de déplacement.*  
 *SpeedMS : Vitesse en mètres/seconde.*  
 *TurnSpeedDS : Vitesse de virage en degrés/seconde.  
 Sleep : Si vrai, aucun calcul de physique ne sera effectué  
 CollisionsSleep : Si vrai, aucun calcul de collisions ne sera effectué*   
  
 *L’élément suivant peut être ajouté pour faire suivre une trajectoire à un objet.*  
 *Il s’agit simplement d’une suite de points de passage géo-localisés.*  
 <Trajectory>  
 <Geoloc Latitude="19.2487" Longitude="10.8959" Altitude="0.0"/>  
 <Geoloc Latitude="19.2476" Longitude="10.8969" Altitude="0.0"/>  
 </Trajectory>  
 </Component> Exemple de composant : Objet avec parent.  
 <Component Name="VisitorDriver" Class="CCamera">  
 *Name : Nom du composant.*  
 *Class : Classe du composant (voir la* ***CComponentFactory****).*  
 <Parent Name="Man1"/>  
 *Propriétés de parenté.*  
 *Name : Nom du parent dont l’objet hérite des transformations.*  
 <Position x="0.0" y="3.2" z="-4.0"/>  
 *Position de l’objet dans le repère affine local du parent.*  
 <Rotation x="10.0" y="0.0" z="0.0"/>  
 *Rotation de l’objet dans le repère affine local du parent.*  
 </Component>  
 Exemple de composant : Contrôleur d’animation.  
 <Component Name="BDD0MastAnim" Class="CBasicAnimator" RotationTarget="BDD0Mast" Playing="true">  
 *Cet objet n’a pas besoin de position ni de parent car il n’est pas visible.*  
 *Name : Nom du composant.*  
 *Class : Classe du composant (voir la* ***CComponentFactory****).*  
 *RotationTarget : Composant cible pour l’animation de rotation  
 (Paramètre lu par* ***CController****).*  
 *Playing : Si true, l’animation est active, si false, l’animation est en pause  
 (Paramètre lu par* ***CAnimator****)*  
 <Rotation>  
 *Définition de l’animation de rotation.*  
 *Peut contenir autant de <Step> que nécessaire.*  
 <Step Pause="0.0">  
 *Pause : Le temps en secondes à attendre avant de commencer  
 ce pas d’animation.*  
 <Target x="0.0" y="0.0" z="2.0"/>  
 *La valeur cible à atteindre sur les trois axes.*  
 <Speed x="8.0" y="8.0" z="8.0"/>  
 *La vitesse de rotation en degrés par seconde.*  
 </Step>  
 <Step Pause="0.2">  
 <Target x="0.0" y="0.0" z="-2.0"/>  
 <Speed x="8.0" y="8.0" z="8.0"/>  
 </Step>  
 </Rotation>  
 <Translation>  
 *Idem que pour rotation mais en mètres/seconde.*  
 </Translation>  
 </Component>  
  
 </Components>  
 *Balise de fin de la liste de composants.*</Scene>  
 *Balise de fin de la scène.*

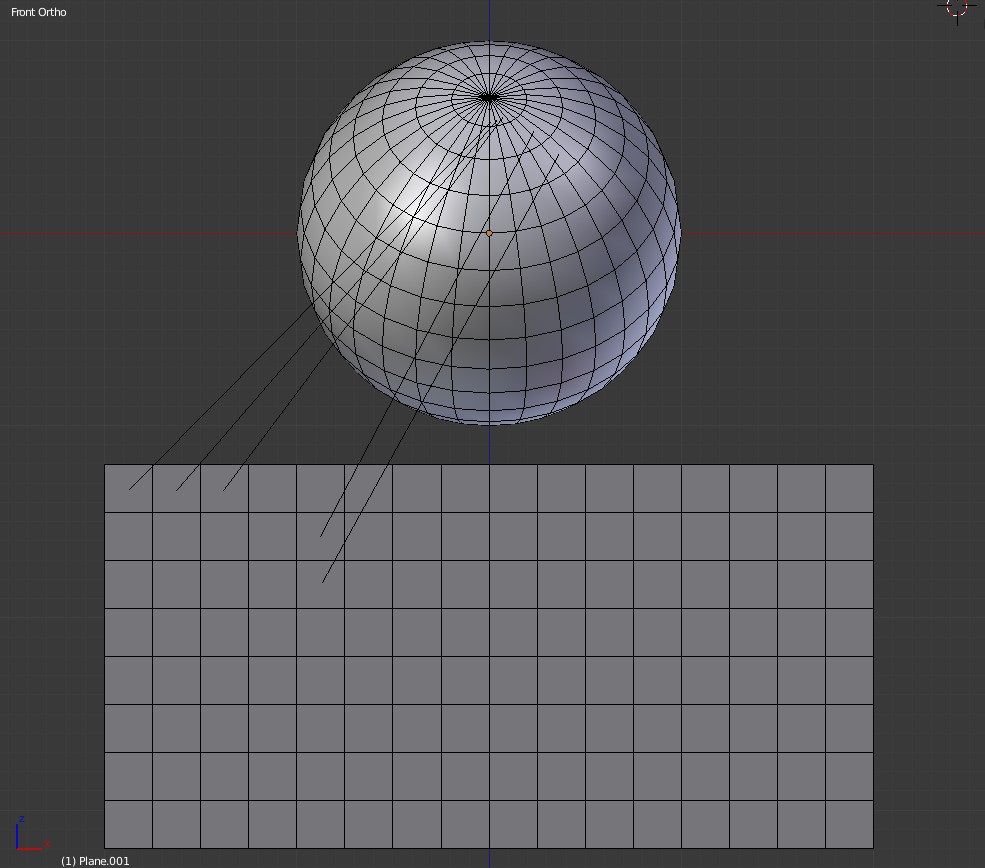
Note :  
Le nom d’un composant doit être unique, particulièrement si des enfants vont s’y attacher. C’est la propriété qu’utilise Quick3D pour lier les objets entre eux.

Chacun des paramètres a une valeur par défaut, ils peuvent donc être omis si souhaitable. Bien entendu, le nom de classe du composant fait exception, il doit toujours être spécifié.

# 9 Rendu de matrice panoramique

Afin de générer une matrice panoramique de profondeur, il s’agit de stocker les informations de profondeur lorsqu’**OpenGL** fait un rendu de la scène. Pour ce faire, le Fragment Shader standard de **Quick3D** comporte un flag lui indiquant s’il doit générer ou non ces informations de profondeur et d’angle d’incidence (**u\_depth\_computing** dans le fichier FS\_Standard.c).

Le rendu de la matrice est fait dans la classe **CCamera**, méthode **renderDepth\_CubeMapped()**.  
Etant donné qu’**OpenGL** ne sait pas faire de rendu panoramique de base, il s’agit de « balayer » la scène en faisant une multitude de petits rendus (512) qui vont être regroupés dans une image finale.



*La sphère représente la grille virtuelle à travers laquelle la caméra voit le monde. Le rectangle est l’image dans laquelle sont concaténés les mini-rendus de la scène.*

Note :  
La définition finale de la matrice dépend donc du nombre et de la résolution de mini-rendus de la scène. Sachant qu’en gisement l’image finale rendue par OpenGL fera 8192 pixels, s’il est prévu de générer une matrice plus résolue en gisement, le nombre de mini-rendus devra être augmenté.

Le principe de génération de matrice est le suivant :

* Préparation de la scène
  + Création du **CAutoTerrain** en spécifiant que sa construction doit être immédiate
  + Création de la caméra
* Appel de la méthode **CCamera::renderDepth\_CubeMapped()**
  + Création d’un viewport pour le rendu de la scène
  + Création d’une image finale pour regrouper les portions de scène rendues
    - Chaque portion de scène fait 256x256 pixels
    - Il y a 32 portions en gisement et 16 portions en site
  + Rendu de chaque portion de scène et transfert dans l’image finale
  + Itération sur chaque pixel de la matrice finale
    - Récupération de la distance dans l’image
    - Récupération du type de zone
    - Stockage de ces informations dans les matrices de sortie

Certaines valeurs pour le rendu sont défini par un **#define** dans le code de **CCamera** :

* RENDER\_MAP\_SIZE\_WIDTH Cette valeur définit la largeur du viewport à travers lequel sera vu chaque petit bout de scène rendu.
* RENDER\_SUBDIVISIONS\_PAN Nombre de subdivisions de la matrice panoramique en gisement. C'est-à-dire le nombre de petits rendus faits à travers le viewport et placés côte à côte dans l’image finale.
* RENDER\_SUBDIVISIONS\_TILT Nombre de subdivisions de la matrice panoramique en site.

Le point d’entrée des calculs de matrice se trouve dans la classe **CQuick3DUtilities**.  
Cette classe comporte deux méthodes de calcul de matrice :

* **computeSphericalDetectionMatrices\_RayTraced()**
* **computeSphericalDetectionMatrices\_CubeMapped()**

C’est la seconde méthode qui est actuellement utilisée dans BARIER car plus rapide.

Les deux méthodes acceptent en entrée une instance de **CPanoramicMatrixParams** (un conteneur de données d’entrée et de sortie pour calcul d’une matrice).  
Elles préparent une scène et une caméra et appellent à leur tour soit **CCamera::renderDepth\_RayTraced()**, soit **CCamera::renderDepth\_CubeMapped()**.  
Elles obtiennent en sortie des vecteurs de nombres qui contiennent la profondeur, l’indice de zone et l’angle d’incidence de chaque point de la matrice.  
Ces données sont fournies à la méthode **CQuick3DUtilities::computeMatrixImages()** qui créé des images de référence des matrices.  
Finalement, les vecteurs sont récupérés par l’appelant (comme le **LPCCService**) qui va les sauvegarder sur disque.

# 10 Cas typique d’utilisation de Quick3D

Voici les actions d’une application **Quick3D** standard :

* Appel de l’initialisation de **Quick3D** :
  + CComponentFactory::getInstance()->registerCoreComponents();
* Création d’un objet scène :
  + m\_pScene = new CGLWidgetScene();
* Création d’un viewport :
  + m\_pScene->getViewports()[0] = new CViewport(m\_pScene, true);
* Chargement des composants depuis un fichier de scènes :
  + QVector<CComponent\*> vComponents = CComponentLoader::getInstance()->load(m\_pScene, ...);
* Initialisation de la scène avec les composants chargés:
  + m\_pScene->init(vComponents);
* Assignation des grilles d’altitudes aux véhicules avec trajectoire
  + m\_pScene->autoResolveHeightFields();
* Création d’un widget d’affichage (hérité de QGraphicsView) :
  + m\_pView = new CView(m\_pScene, ui.Render1);
  + (ui.render1 est un **QFrame**)

Note :  
Il est tout à fait possible de ne pas utiliser le CComponentLoader et d’instancier chaque composant de manière programmatique. Il suffira de créer un vecteur de CComponent, de le remplir en instanciant les composants et de le fournir à C3DScene::init().

Sur timer, pour chaque image :

* Calcul du delta time en secondes (temps écoulé depuis la dernière image)
* Mise à jour de la scène :
  + m\_pScene->updateScene(dDeltaTime);
* Mise à jour du widget d’affichage :
  + m\_pView->update(dDeltaTime);
  + A son tour, la vue fera une mise à jour de chacun de ses viewports, ce qui provoquera des rendus de scène