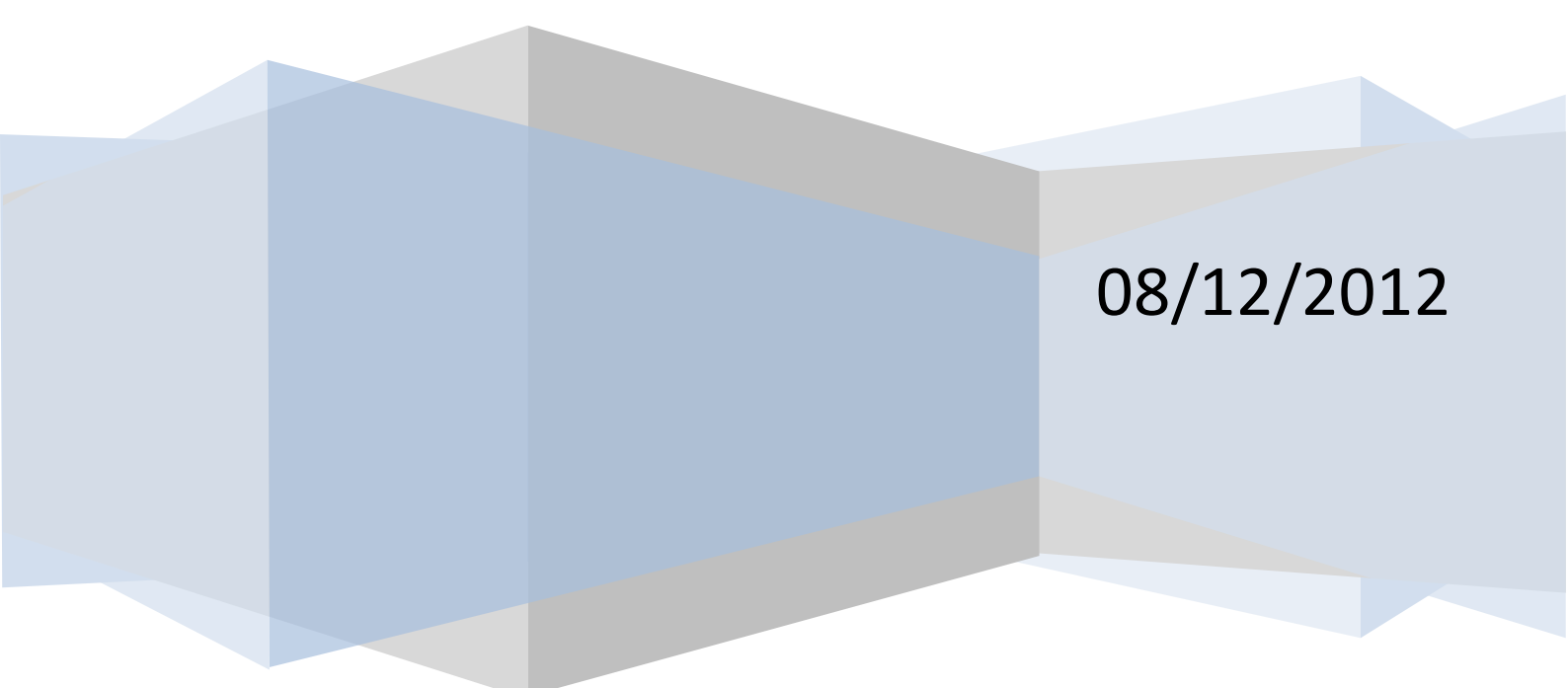


# NeoAxis engine

Affichage de données surfaciques

MONIER Vincent



08/12/2012

## Sommaire

---

NeoAxis: Affichage de données surfaciques.....	2
Définition des données à afficher .....	2
Rendu utilisé .....	2
Mise en œuvre .....	4
Mise en garde.....	4
Création du mesh .....	4
Conversion OgreXML vers OgreMESH .....	6
Création du *.type .....	7
Edition des HighMaterial .....	7
Ne pas utiliser de texture :.....	9
Lumière ambiante et ombres:.....	10
Transparence : .....	11
Culling : .....	11
Synthèse.....	11

---

## NeoAxis: Affichage de données surfaciques

---

### Définition des données à afficher

---

On considère que l'on dispose d'une surface définie par un ou plusieurs mesh (modèle) 3D, au format OgreXML et que l'on souhaite afficher, sur cette surface, des données. Ces données peuvent être à 1, 2 ou 3 dimensions. En tout point de la surface, on souhaite donc avoir une représentation de la valeur de la donnée.

Pour une donnée à 1 dimension, il est conseillé d'utiliser un dégradé du blanc au noir, le noir étant la valeur la plus basse pour la donnée. Pour une donnée à 3 dimensions, il est conseillé d'utiliser les couleurs Rouge, Verte et Bleue pour représenter la valeur de chacune de ces dimensions. Pour des données à 2 dimensions, le développeur devra mettre en place un système permettant à l'utilisateur de choisir quel couple de couleur utiliser (Rouge/Vert, Rouge/Bleu ou Bleu/Vert) pour représenter chacune de ces dimensions.

Dans ce document, on pourra considérer :

- Soit que la donnée est définie par une fonction de l'espace, donc calculable en tout point de l'espace et donc, en tout point de la surface du modèle 3D
- Soit que la donnée est définie pour un ensemble fini de points de l'espace et que, par linéarisation, on pourra estimer la valeur de la donnée en tout point de l'espace

### Rendu utilisé

---

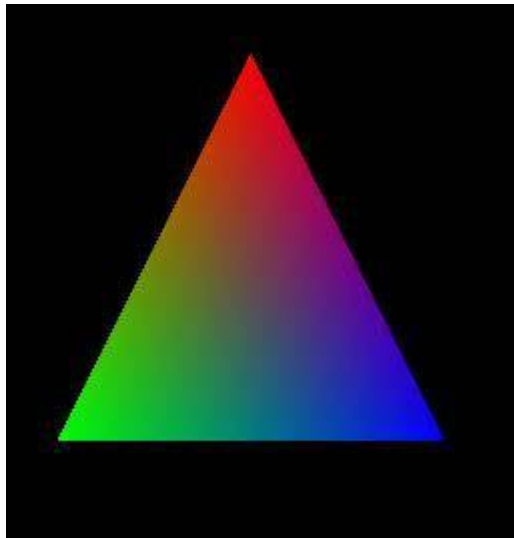
Pour afficher la donnée sur la surface d'un modèle 3D, le système suivant sera utilisé

- Pour chaque sommet S
  - Calculer la valeur V de la donnée en S
  - Calculer la couleur C associée à V
  - Enregistrer que le sommet S est de couleur C

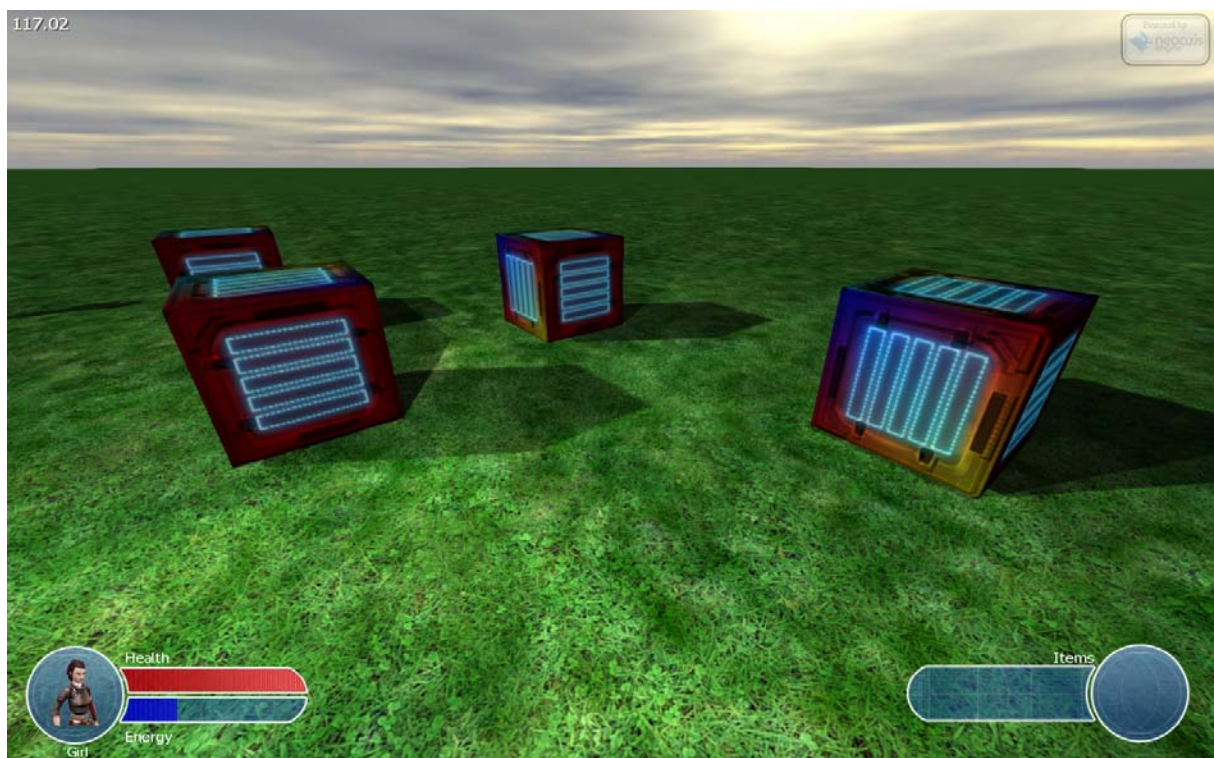
Ensuite, lorsque NeoAxis affichera le modèle 3D, le calcul suivant sera réalisé :

- Pour chaque pixel de la surface du modèle 3D
  - Calculer la lumière L reçue en ce pixel
  - Calculer la couleur C associée à la valeur de la donnée pour ce pixel (linéarisation)
  - Multiplier L et C pour définir la couleur finale F de ce pixel
  - Multiplier F par la valeur T de la texture en ce pixel ( $F * T = P$ )
  - Afficher le résultat final P

On peut utiliser une texture blanche pour le modèle 3D, et ne pas faire de calcul de lumière. En ce cas, la valeur finale P est égale à la couleur C de la donnée.



*Dans ce triangle, chaque sommet se voit attribué une couleur (dans l'ordre horaire, Rouge, Bleu, Vert). La couleur de chaque pixel du triangle est calculée par linéarisation des couleurs aux sommets.*



*La donnée est définie en chaque sommet de la surface du modèle 3D. Si le même modèle 3D est réutilisé plusieurs fois, chacun aura la même couleur en ses sommets que les autres (autrement dit, si deux cubes devaient avoir des couleurs différentes en leurs sommets, il faudrait faire deux modèles 3D c'est à dire deux fichiers \*.type différents).*

## Mise en œuvre

Pour réaliser ce rendu, il faut utiliser une propriété des verticles : la « colours\_diffuse ». Cet attribut peut être défini pour chaque sommet d'un modèle 3D. Il s'agit d'une couleur RGBA (Rouge Vert Bleu Alpha), dont chaque composante s'étale entre 0 et 1. Nous n'utiliserons, dans notre cas, que les composantes RGB : la valeur alpha (transparence) n'étant pas pertinente.

Si le dégradé doit être en noir et blanc, il suffit d'appliquer les mêmes valeurs aux composantes Rouge Verte et Bleue.

La mise en œuvre se déroule en plusieurs étapes, décrites ci-dessous.

## Mise en garde

Il n'est pas possible de modifier la valeur « colours\_diffuse » en temps réel. Il ne sera donc pas possible de changer les couleurs des sommets en temps réel. La donnée doit donc être connue AVANT de lancer l'environnement 3D dans NeoAxis. Si la donnée est modifiée pendant que NeoAxis affiche la scène 3D, il faudra alors recharger l'environnement 3D (cela se fait en rechargeant le fichier \*.map associé à l'environnement).

Il n'est donc pas possible de modifier directement les données en temps réel. Toutefois, on pourra essayer la méthode suivante si les données venaient à être modifiées :

- Créer un nouveau modèle 3D selon les étapes décrites ci-dessous, modèle indépendant du modèle déjà existant et déjà en cours d'affichage dans NeoAxis
- Indiquer à NeoAxis d'utiliser dorénavant ce nouveau modèle
- Implémenter un code dans NeoAxis qui sera capable de « rediriger » tous les appels à l'ancien modèle 3D vers ce nouveau modèle 3D

Autrement dit, au lieu de modifier un modèle 3D existant, on en crée un autre, puis on supprime l'ancien modèle dans la scène 3D, et on le remplace par le nouveau modèle 3D. Ceci ne sera pas à faire souvent car ces opérations sont longues et coûteuses en temps de calcul.

## Création du mesh

D'abord, il faut créer le ou les mesh(es). Un mesh est, pour rappel, un ensemble de surfaces chacune découpée en triangles et sommets. Un modèle 3D est un assemblage d'un ou plusieurs mesh liés à un HighMaterial (shader), ainsi que d'un modèle physique, de sons et d'effets visuels. La liste des composantes avec leur position est définie dans le fichier \*.type du modèle 3D.

Une fois le mesh créé, classiquement via un éditeur 3D ou via un algorithme automatique, il faut l'exporter au format OgreXML pour pouvoir le manipuler.

Le fichier XML contient un ou plusieurs « vertexbuffer ». Un vertexbuffer est un élément XML qui contient la définition de tous les sommets (vertex) du modèle 3D. Pour utiliser les « vertexcolor » (ou « colours\_diffuse »), il faut d'abord déclarer leur utilisation dans les attributs du vertexbuffer. Pour ce faire, ajoutez l'attribut suivant à la balise « vertexbuffer » : `colours_diffuse="true"`

Ensuite, il faut déclarer la couleur pour chacun des vertex. Pour ce faire, ajoutez la balise « `<colour_diffuse value="R.RRR V.VVV B.BBB" />` » où R.RRR est la valeur de la composante rouge, V.VVV la valeur verte, et B.BBB la bleue. Ce tag doit être ajouté pour chacun des vertex ! S'il n'est pas défini pour l'un des vertex, le modèle 3D fera planter le SDK (NeoAxis) lorsqu'il sera affiché.

Attention ! Le nom de la balise est bien « colour\_diffuse », sans « s » après « colour », alors que l'attribut à ajouter pour vertexbuffer est « colours\_diffuse », avec un « s ».

Si, pour un même submesh, plusieurs vertexbuffer sont définis, alors il suffit de définir les vertexcolor dans un seul de ces vertexbuffer. En revanche, si le mesh possède plusieurs submeshes, il faudra définir les vertexcolor pour chacun de ces submesh.

## Submesh1

- vertexbuffer
- Position
- Normal
- Colour\_diffuse
- vertexbuffer
- Tangent
- Texcoords 1
- Texcoords 2

## Submesh2

- vertexbuffer
- Position
- Normal
- Texcoords 1
- Colour\_diffuse

*S'il y a plusieurs submeshes, il faut définir les vertexcolor une fois pour chaque submesh ; si un submesh possède plusieurs vertexbuffer, il ne faudra ajouter les colour\_diffuse que pour un seul de ces vertexbuffer (généralement, celui qui définit les positions des vertex).*

```

GunBase.mesh.xml
289      <face v1="852" v2="853" v3="854" />
290      <face v1="855" v2="856" v3="857" />
291      <face v1="858" v2="859" v3="860" />
292      <face v1="861" v2="862" v3="863" />
293      <face v1="864" v2="865" v3="866" />
294      <face v1="867" v2="868" v3="869" />
295      <face v1="870" v2="871" v3="872" />
296      <face v1="873" v2="874" v3="875" />
297      <face v1="876" v2="877" v3="878" />
298      <face v1="879" v2="880" v3="881" />
299  </faces>
300  <geometry vertexcount="882">
301    <vertexbuffer colours_diffuse="true" positions="true" normals="true" texture_coord_dimensi
302    <vertex>
303      <position x="1.15085" y="0.17138" z="0.0725217" />
304      <normal x="0.0993088" y="0.917861" z="0.384278" />
305      <texcoord u="0.325457" v="0.150116" />
306      <texcoord u="14.7548" v="-5.23017" />
307      <tangent x="-0.995049" y="0.0931699" z="0.0346107" />
308      <colour_diffuse value="0.0029667110190489 0.81653781193267 0.64301769979795" />
309    </vertex>
310    <vertex>
311      <position x="1.15085" y="0.184984" z="0.00523562" />
312      <normal x="0.0992855" y="0.995059" z="3.59347e-007" />
313      <texcoord u="0.325457" v="0.157102" />
314      <texcoord u="16.0144" v="0.999997" />
315      <tangent x="-0.995059" y="0.0992855" z="-1.4983e-005" />
316      <colour_diffuse value="0.0029667110190489 0.83712032412974 0.51047047459101" />
317    </vertex>
318    <vertex>
319      <position x="0.344657" y="0.193385" z="0.00523562" />
320      <normal x="0.0104201" y="0.999946" z="2.56713e-007" />
321      <texcoord u="0.400493" v="0.157101" />
322      <texcoord u="16.7923" v="1" />
323      <tangent x="-0.999946" y="0.0104201" z="-1.49556e-005" />
324      <colour_diffuse value="0.99079620956045 0.84933613762509 0.51047047459101" />
325    </vertex>
326    <vertex>
327      <position x="0.344657" y="0.193385" z="0.00523562" />
328      <normal x="0.0104201" y="0.999946" z="2.56713e-007" />

```

Dans cet exemple de fichier OgreXML, `colour_diffuse` a été rajouté pour chaque vertex définissant le tag « position » (qui est la coordonnée XYZ du sommet dans le repère local du mesh).

## Conversion OgreXML vers OgreMESH

Pour pouvoir utiliser le fichier OgreXML comme mesh, il vous faudra d'abord convertir le fichier XML en fichier \*.mesh. Pour ce faire, utilisez l'outil « OgreXML converter », fourni sur le site d'Ogre.

Cet outil est régulièrement mis à jour, il est donc conseillé de vérifier régulièrement la disponibilité d'une nouvelle version. La rétro-compatibilité est certifiée par l'équipe de développement d'Ogre (autrement dit, si vous utilisez une version de NeoAxis qui est prévue pour utiliser des fichiers mesh version 1.5, vous pourrez quand même y utiliser des fichiers mesh de version supérieure).

L'outil fonctionne en stand-alone : il n'est pas nécessaire d'installer Ogre ou NeoAxis pour l'utiliser.

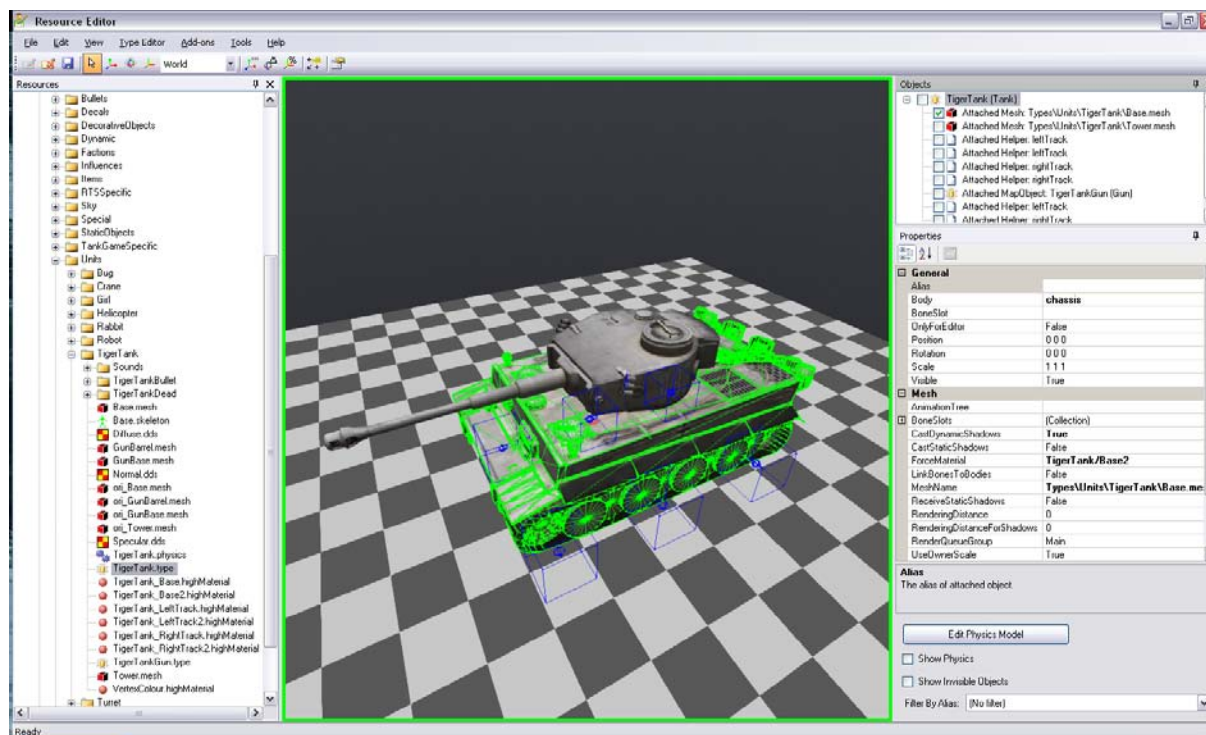
Pour réaliser la conversion, vous pouvez simplement déplacer le fichier XML jusque sur l'icône du programme « Ogre XML converter ». Le programme convertira alors le fichier XML. Il est également possible d'utiliser le convertisseur via la ligne de commandes.

Une fois le modèle converti, il est conseillé de créer un sous-dossier dans « .../NeoAxis/Bin/Data », sous-dossier qui sera dédié au modèle 3D. Déplacez le ou les fichiers \*.mesh dans ce dossier. Ce dossier contiendra également vos textures, votre ou vos fichiers Highmaterial (créez-les dès maintenant comme des shaders classiques, via l'éditeur de ressources), ainsi que le fichier \*.type du modèle 3D (cf étape suivante).

## Création du \*.type

Une fois le mesh créé, il faut créer le fichier \*.type associé. Il est conseillé de créer, d'abord, les fichiers HighMaterial (shaders) du modèle 3D, ce qui vous permettra de vérifier que le modèle est correctement texturé. Dans l'exemple actuel, le modèle utilisé est un modèle de char déjà existant dans le SDK. La création des HighMaterial ne sera donc pas détaillée puisque ces fichiers existent déjà.

Pour créer le fichier \*.type, utilisez l'éditeur de ressource, puis « New Ressource », « Type File », et attachez-y le ou les meshes qui composent le modèle 3D.



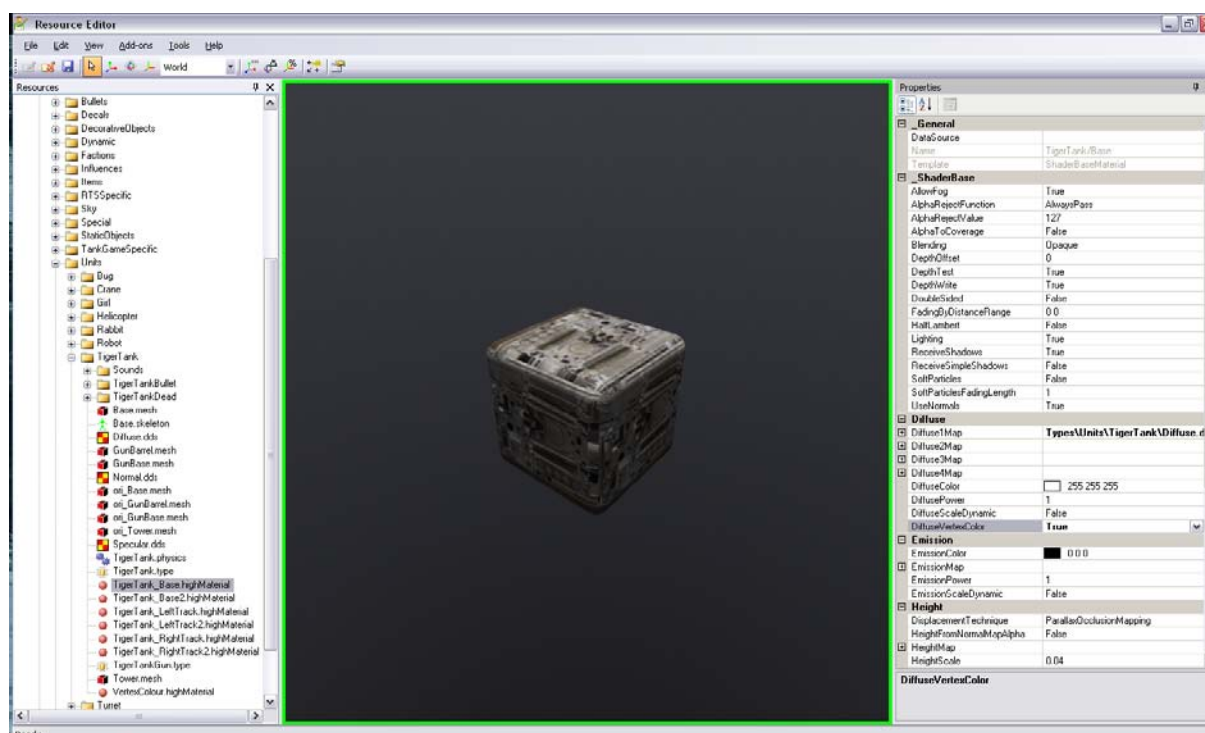
*Ce modèle 3D possède plusieurs meshes, mais également un sous-modèle 3D (Gun), un modèle physique (défini dans les propriétés du modèle 3D) et des aides, entités invisibles permettant de faciliter la manipulation du modèle 3D dans le script de la map (Helpers).*

Vous avez donc, maintenant, un modèle 3D classique, correctement texturé, mais pour l'instant, les vertexcolor sont ignorées. Il faut éditer le HighMaterial pour les prendre en compte.

## Edition des HighMaterial

Reprenez les Highmaterial créés précédemment. Ouvrez-les en édition dans l'éditeur de ressource, et modifiez le paramètre « DiffuseVertexColor » pour le définir à « true » (zone « DiffuseMap ») :





Définissez « DiffuseVertexColor » à true pour que le shader prenne en compte les vertexcolors.

Maintenant, si vous affichez le modèle 3D dans l'environnement 3D, les vertexcolor sont prises en compte :



La couleur issue de chaque sommet est multipliée par la valeur de la texture et par la couleur issue de l'éclairage diffus ambiant pour donner ce rendu.

## Ne pas utiliser de texture :

« Diffuse1Map » peut être mis à « null » pour ne pas utiliser de texture (la couleur utilisée est alors celle définie par « DiffuseColor » : mettez du blanc pur pour que la couleur finale, dans le moteur 3D, soit celle issue des couleurs des sommets du modèle 3D).



*Le modèle de gauche utilise une texture, pas celui de droite (qui est quand même ombré).*



*Ici, sans texture, seul l'éclairage influence la couleur du rendu.*

## Lumière ambiante et ombres:

« Lighting » peut être mis à false pour ignorer le calcul de lumière diffuse et spéculaire, et donc, ne pas laisser la lumière ambiante influencer la couleur du rendu final. « ReceiveShadows » peut aussi être mis à false pour que le modèle 3D ne soit pas influencé par sa propre ombre.



*Il est possible, via le HighMaterial lié au mesh, de ne plus calculer les ombres. La couleur est maintenant directement représentative de la donnée associée à la surface du modèle 3D.*



## Transparence :

Pour ajouter de la transparence, définissez « DiffuseColor » sur « 255 255 255 A » où A est la valeur d'opacité de la texture, entre 0 (totalement transparente donc invisible) et 255 (totalement opaque) ; n'oubliez pas de modifier également le paramètre « Blending » dans « \_ShaderBase » pour le mettre sur « AlphaAdd » (superposition) ou « AlphaBlend » (mélange, autrement dit, transparence classique).



*Il est possible de rendre le HighMaterial à demi transparent. En plaçant ce « fantôme » au même endroit que le modèle 3D normal (c'est-à-dire celui texturé), il est possible d'afficher la donnée « par-dessus » le modèle normal.*

## Culling :

Dans la section « \_ShaderBase », vous pouvez définir « DoubleSided » à true pour que les triangles du modèle 3D soient aussi visibles « de dos ». Vous ne verrez une différence que si vous avez également utilisé la transparence (la transparence vous permet de voir à travers le mesh, et donc, de voir certains triangles « de dos »).

## Synthèse

- La couleur ne peut être définie que pour les sommets du modèle 3D (pour définir la couleur pour un triangle, il suffit de la définir pour chacun des sommets)
- Si le modèle n'est pas assez détaillé pour afficher les données, il faut ajouter des sommets aux endroits où on souhaite plus de précision
- La couleur des sommets ne peut pas être éditée en temps réel
- Ne pas oublier de définir « DiffuseVertexColor » à « true » dans le HighMaterial
- Il est nécessaire de passer par le format OgreXML puis le convertisseur vers ogreMESH pour pouvoir définir les couleurs