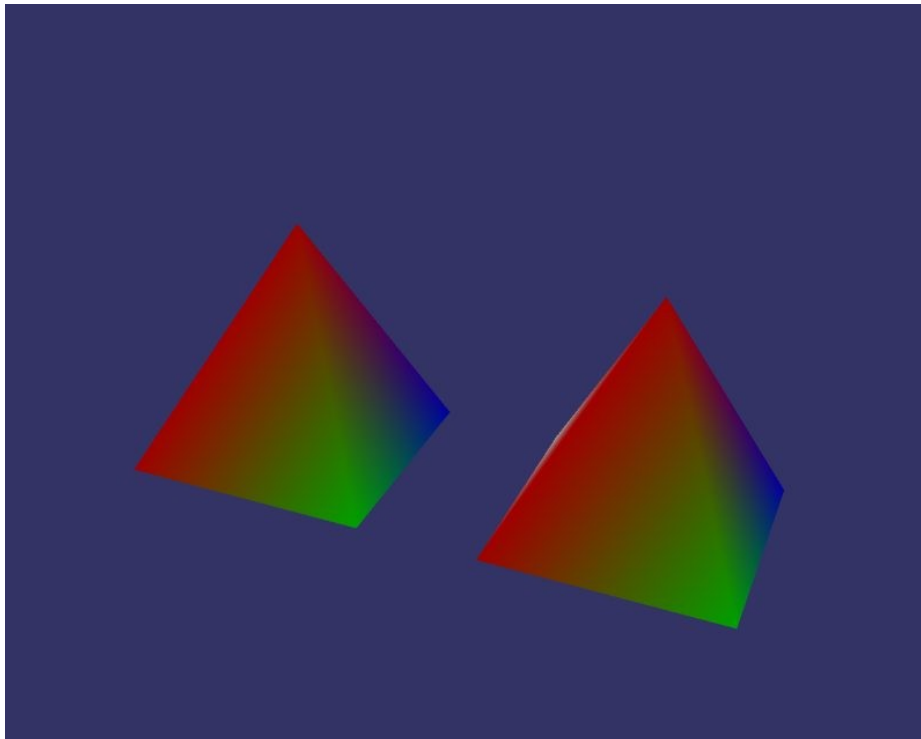


Travailler avec Open Scene Graph

1. Afficher une forme géométrique simple

Traduction par Thomas Baquet



Source: <http://www.openscenegraph.org/projects/osg/wiki/Support/Tutorials/> -by **Joseph Sullivan**

Traduction: Février 2008

Sources: 2003-2006

NdT: Avant de commencer

Il est possible qu'il y ait encore des erreurs de traduction, si c'est le cas, prévenez-moi. Aucune responsabilité ne peut cependant être engagée contre le traducteur ou l'auteur suite à ces erreurs, ou à quelque erreur à la lecture de la traduction.

Il y a eu également eu des mots impossibles à retraduire en les mettant en contexte dans la traduction (je pense à des mots comme « geometry » par exemple qui est employé comme nom simple). Dans ces cas, j'ai soit employé le mot en anglais, soit tourné la phrase de façon à garder le même sens.

En lisant ce document, le lecteur prend pleinement conscience de ce qui vient d'être écrit.

Introduction

Cette section présente quelques méthodes qui peuvent être utilisées pour créer des primitives géométriques. Il y a plusieurs méthodes pour travailler avec les formes géométriques: au plus bas niveau avec les primitives se basant sur OpenGL; à un niveau intermédiaire avec les formes basiques d'Osg et à un haut niveau, à partir de fichiers. Nous verrons dans ce tutoriel le niveau le plus bas. Cette méthode permet non seulement le plus de flexibilité, mais (malheureusement) requiert le plus code. Normalement une scène est chargée via des fichiers: le plus gros de l'effort de la définition des sommets est fait par le file loader.

Avant de commencer

Une petite explication sur les différentes classes s'avère utile:

La classe « Geode »:

La classe geode est dérivée de la classe « node ». Les noeuds (et donc la classe geode) peuvent être ajoutés à une scène comme des feuilles. Les instances de la classe peuvent avoir différent nombre de classes « drawables » qui lui sont associés.

La hiérarchie des classes « drawable »:

La classe de base est purement virtuelle, avec six classes dérivées concrètes ([référence](#)). la classe « geometry » peut avoir des vertices (ou des données de vertices) associées directement ou peuvent avoir un numéro de l'instance « primitiveSet » associées avec.

Les vertices et leurs attributs (couleurs, normales, coordonnées de textures) sont stockées dans des tableaux. Plusieurs vertices peuvent avoir la même couleur, la même normale ou les mêmes coordonnées de texture, et un tableau peut être utilisé pour mapper des tableaux de vertices sur des tableaux de couleurs (ou de normales, ou encore de coordonnées de textures).

La classe « PrimitiveSet »:

Cette classe permet de dessiner les primitives de base d'Opengl: POINTS, LINES, LINE_STRIP, LINE_LOOP,... QUADS,... POLYGON.

Le Code

Cette section va nous permettre de créer une vue d'une scène que nous créerons, un « groupe » d'instance qui servira comme noeud principal de la scène, un noeud géométrique (geode) qui s'occupera des drawables, et une instance géométrique qui associe les vertices et les données de vertex. (Ici, la forme à dessiner dans le rendu est une pyramide à quatre cotés.)

```
...
int main()
{
    ...
    osgProducer::Viewer viewer;
    osg::Group* root = new osg::Group();
    osg::Geode* pyramidGeode = new osg::Geode();
    osg::Geometry* pyramidGeometry = new osg::Geometry();
```

Ensuite nous associons [la géométrie] pyramidGeometry à pyramidGeode, et nous ajoutons ce dernier au noeud principal de la scène.

```
pyramidGeode>addDrawable(pyramidGeometry);
root>addChild(pyramidGeode);
```

On déclare un tableau de vertices. Chaque vertex sera représenté par un vecteur (instance de la classe Vec3). Une instance de osg::Vec3Array peut être utilisée pour les stocker. Comme osg::Vec3Array est dérivée de la classe vector de la STL, nous pouvons utiliser la méthode push_back pour ajouter les éléments du tableau. Puisque push_back ajoute les éléments à la fin du tableau, l'index du premier élément sera 0, la seconde entrée 1, etc. Nous utilisons ici le système de coordonnées right-handed¹ avec ici la cote vers le haut². Quatre points représentent la base (de cote égale à zéro) sur les cinq requis pour créer une simple pyramide.

```
osg::Vec3Array* pyramidVertices = new osg::Vec3Array;
pyramidVertices>push_back( osg::Vec3( 0, 0, 0 ) ); // devant à gauche
pyramidVertices>push_back( osg::Vec3(10, 0, 0 ) ); // devant à droite
pyramidVertices>push_back( osg::Vec3(10,10, 0 ) ); // derrière à droite
pyramidVertices>push_back( osg::Vec3( 0,10, 0 ) ); // derrière à gauche
pyramidVertices>push_back( osg::Vec3( 5, 5,10 ) ); // sommet
```

En associant cet ensemble de vertices avec pyramidGeometry elle-même associée avec [le geode] pyramidGeode, nous l'ajoutons à la scène.

```
pyramidGeometry>setVertexArray( pyramidVertices );
```

1 Ndt: il existe deux types de système de coordonnées: le right-handed et le left-handed. Cette appellation est due à un petit moyen de reconnaître facilement lequel correspond à quoi: lorsque l'on utilise le pouce comme l'axe des abscisses, l'index comme celui des ordonnées, et le majeur comme celui des cotes, on peut observer que la direction du majeur change. Faites le test pour mieux comprendre.

2 Ndt: Ici, la cote et l'ordonnée sont échangées selon le modèle mathématique.

Ensuite, nous créons une primitive et l'ajoutons à pyramidGeometry. Nous utilisons les quatre premiers points de la pyramide pour définir la base, en utilisant une instance de la classe DrawElementsUInt. Cette classe étant également dérivée de la classe vector de la STL, la méthode push_back peut être utilisée pour ajouter les éléments dans un ordre séquentiel. Pour assurer une bonne suppression des faces arrières (backface culling), les vertices doivent être spécifiées dans le sens des aiguilles d'une montre. Les arguments pour le constructeur sont les énumérations du type de primitive (semblable aux types d'énumération d'OpenGL, ici GL_QUADS), et l'index où commencer dans le tableau de vertices.

```
osg::DrawElementsUInt* pyramidBase =
    new osg::DrawElementsUInt(osg::PrimitiveSet::QUADS, 0);
pyramidBase>push_back(3);
pyramidBase>push_back(2);
pyramidBase>push_back(1);
pyramidBase>push_back(0);
pyramidGeometry>addPrimitiveSet(pyramidBase);
```

On répète la même chose pour chaque côté de la pyramide. Les vertices sont toujours spécifiées dans le sens horlogique.

```
osg::DrawElementsUInt* pyramidFaceOne =
    new osg::DrawElementsUInt(osg::PrimitiveSet::TRIANGLES, 0);
pyramidFaceOne>push_back(0);
pyramidFaceOne>push_back(1);
pyramidFaceOne>push_back(4);
pyramidGeometry>addPrimitiveSet(pyramidFaceOne);

osg::DrawElementsUInt* pyramidFaceTwo =
    new osg::DrawElementsUInt(osg::PrimitiveSet::TRIANGLES, 0);
pyramidFaceTwo>push_back(1);
pyramidFaceTwo>push_back(2);
pyramidFaceTwo>push_back(4);
pyramidGeometry>addPrimitiveSet(pyramidFaceTwo);

osg::DrawElementsUInt* pyramidFaceThree =
    new osg::DrawElementsUInt(osg::PrimitiveSet::TRIANGLES, 0);
pyramidFaceThree>push_back(2);
pyramidFaceThree>push_back(3);
pyramidFaceThree>push_back(4);
pyramidGeometry>addPrimitiveSet(pyramidFaceThree);

osg::DrawElementsUInt* pyramidFaceFour =
    new osg::DrawElementsUInt(osg::PrimitiveSet::TRIANGLES, 0);
pyramidFaceFour>push_back(3);
pyramidFaceFour>push_back(0);
pyramidFaceFour>push_back(4);
pyramidGeometry>addPrimitiveSet(pyramidFaceFour);
```

Puis on déclare et on charge un tableau de Vec4 pour stocker les couleurs:

```
osg::Vec4Array* colors = new osg::Vec4Array;
colors>push_back(osg::Vec4(1.0f, 0.0f, 0.0f, 1.0f) ); //index 0 rouge
colors>push_back(osg::Vec4(0.0f, 1.0f, 0.0f, 1.0f) ); //index 1 vert
colors>push_back(osg::Vec4(0.0f, 0.0f, 1.0f, 1.0f) ); //index 2 bleu
```

```
colors>push_back(osg::Vec4(1.0f, 1.0f, 1.0f, 1.0f) ); //index 3 blanc
```

On déclare une variable qui associera les éléments du tableau de vertices aux éléments du tableau de couleurs. Elle aura le même nombre d'éléments que le nombre de vertices, et servira ainsi de lien entre les deux. Les entrées dans ce tableau correspondent aux éléments dans le tableau de vertices, et leurs valeurs correspondent aux index du tableau de couleur. C'est le même schéma pour les normales ou les coordonnées de textures.

Il est à noter que dans cet exemple, nous associons cinq sommets à quatre couleurs. Les éléments zéro (inférieur gauche) et quatre (sommet de la pyramide) du tableau sont tous les deux assignés à la couleur d'indice zéro.

```
osg::TemplateIndexArray
    <unsigned int, osg::Array::UIntArrayType,4,4> *colorIndexArray;
colorIndexArray =
    new osg::TemplateIndexArray<unsigned int,
osg::Array::UIntArrayType, 4,4>;

colorIndexArray>push_back(0); // vertex 0 assigné à la couleur 0
colorIndexArray>push_back(1); // vertex 1 assigné à la couleur 1
colorIndexArray>push_back(2); // vertex 2 assigné à la couleur 2
colorIndexArray>push_back(3); // vertex 3 assigné à la couleur 3
colorIndexArray>push_back(0); // vertex 4 assigné à la couleur 0
```

L'étape suivante est d'associer le tableau de couleur avec [la géométrie] pyramidGeometry, puis assigner à ce dernier les indices des couleurs et activer le mode de binding `_PER_VERTEX`.

```
pyramidGeometry>setColorArray(colors);
pyramidGeometry>setColorIndices(colorIndexArray);
pyramidGeometry>setColorBinding(osg::Geometry::BIND_PER_VERTEX);

osg::Vec2Array* texcoords = new osg::Vec2Array(5);
(*texcoords)[0].set(0.00f,0.0f);
(*texcoords)[1].set(0.25f,0.0f);
(*texcoords)[2].set(0.50f,0.0f);
(*texcoords)[3].set(0.75f,0.0f);
(*texcoords)[4].set(0.50f,1.0f);
pyramidGeometry>setTexCoordArray(0,texcoords);
```

Maintenant que nous avons créé un noeud géométrique et que nous l'avons ajouté à la scène, nous pouvons le réutiliser. Par exemple, si nous voulons avoir une seconde pyramide de 15 unités de côtés à droite de la première, nous pouvons ajouter ce geode comme enfant d'un noeud de transformation dans notre scène.

```
// Déclare et initialise un noeud de transformation.
osg::PositionAttitudeTransform* pyramidTwoXForm =
    new osg::PositionAttitudeTransform();

// Utilise la méthode 'addChild' de la classe osg::Group
// pour ajouter la transformation en tant qu'enfant du noeud principal
// et la pyramide comme celui de la transformation
root>addChild(pyramidTwoXForm);
pyramidTwoXForm>addChild(pyramidGeode);
```

```
// Déclare et initialise une instance de Vec3 pour changer la
// position de la pyramide
osg::Vec3 pyramidTwoPosition(15,0,0);
pyramidTwoXForm>setPosition( pyramidTwoPosition );
```

L'étape finale est de mettre en place la boucle principale et d'entrer dedans.

```
viewer.setUpViewer(osgProducer::Viewer::STANDARD_SETTINGS);
viewer.setSceneData( root );
viewer.realize();
while( !viewer.done() )
{
    viewer.sync();
    viewer.update();
    viewer.frame();
}
```

Bonne chance!