Mécanisme du shader

Table des matières

[I. Introduction 1](#_Toc146287539)

[II. Distinction de cas 1](#_Toc146287540)

[1. Couleur uniforme et shading dur 2](#_Toc146287541)

[2. Texture et shading lisse 2](#_Toc146287542)

[3. Couleur uniforme, shading lisse 3](#_Toc146287543)

[4. Généralisation des cas 3](#_Toc146287544)

[III. Les éléments points, textes, lignes 5](#_Toc146287545)

[IV. Shader de sélection 5](#_Toc146287546)

[V. Organisation des données dans le buffer 6](#_Toc146287547)

# Introduction

Pour afficher un élément il faut écrire des shaders. Ce sont des programmes compilés sur la carte graphique qui permettent de construire une image à partir de données qui représentent des objets.

Ils existent plusieurs types de shaders qui ont des fonctionnalités bien particulières ainsi que des entrées et sorties différents. Ces shaders sont exécuter les uns a la suite des autres dans un ordre précis.

Dans la toolbox les shaders utiliser sont : le vertex shader, le geometry shader et le fragment shader.

Le schéma suivant représente les shaders utilisés avec leurs données d’entrées et de sorties. Les boites grises sont gérées par OpenGL.

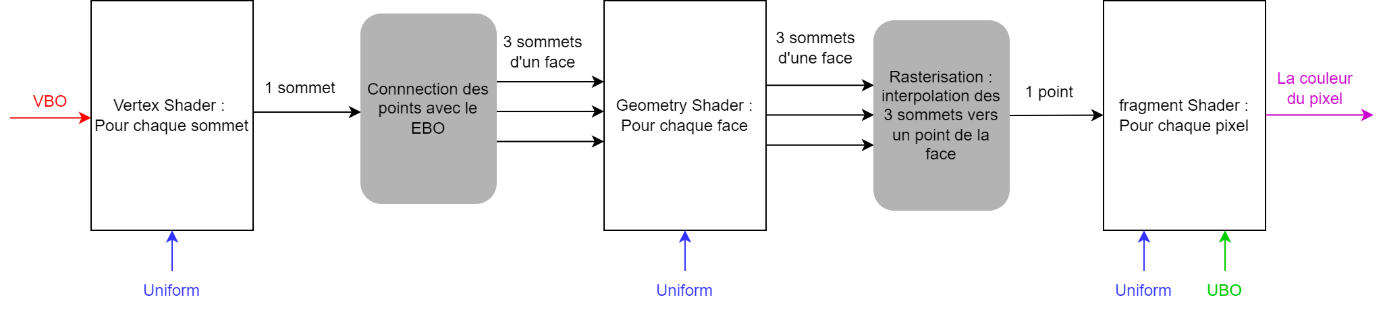


Figure : fonctionnement général shaders

Les shaders sont difficilement modulable. Un objet qui doit être afficher avec une couleur unique ne peut pas être réutiliser pour un objet qui est peint avec une texture.

Cependant leurs structures sont extrêmement similaires. C’est pour cela qu’il n’y a que trois programmes de shader.

* all.*type*.glsl est utiliser pour tous les éléments faces (des éléments 3D sur lesquels ont peut mettre de la lumière)
* noLight.*type*.glsl est utiliser pour les éléments points, lignes et texte (il ne peuvent pas être illuminé car ils n’ont pas de normales)
* drawId.*type*.glsl est utiliser pour dessiner les objets avec leurs Id avant la sélection, le résultat de ce shader n’est pas affichable mais est analyser pour trouver l’objet touché.

Selon le type de coloration ou de shading de chaque objet certaine ligne de chaque fichier seront supprimé pour correspondre au choix d’affichage de l’utilisateur.

# Distinction de cas

Maintenant nous allons isolés différents cas pour les éléments faces avec des modes d’affichages différents pour comprendre les raisons de la synthèse en programmes avec des lignes effaçable.

Dans les différents cas, le schéma présentera les données en entrée et sorties de chaque shader et son utilité.

## Couleur uniforme et shading dur

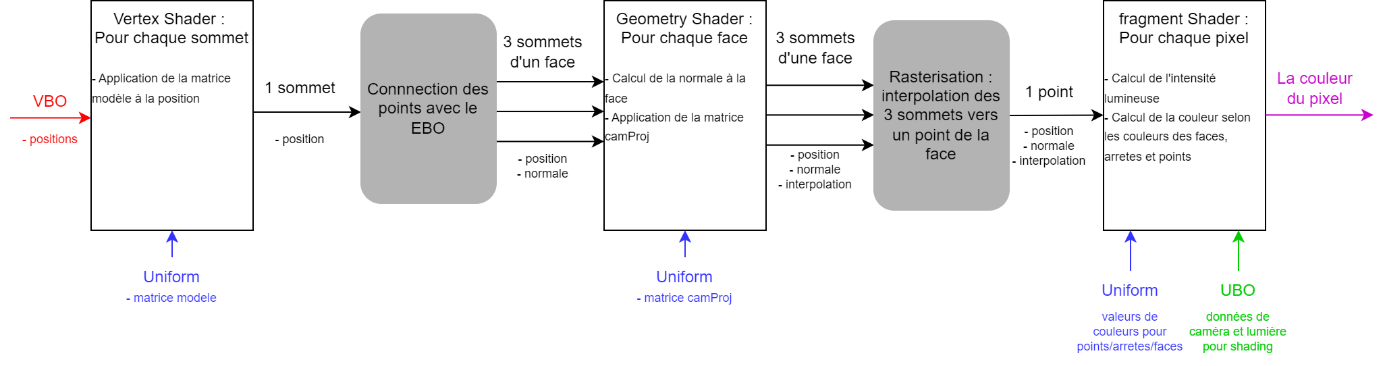


Figure : Enchainement des données dans un cas simple

La matrice modèle est appliquée dès le début car elle positionne les points dans le repère monde. Le VBO contient les points dans un repère relatif à lui-même.

L’application de la matrice camProj déforme l’objet (exemple : vue perspective). Le calcul de la normale doit donc être fait avant cette déformation.

La valeur d’interpolation est utilisée pour dessiner les lignes et les points des objets.

## Texture et shading lisse

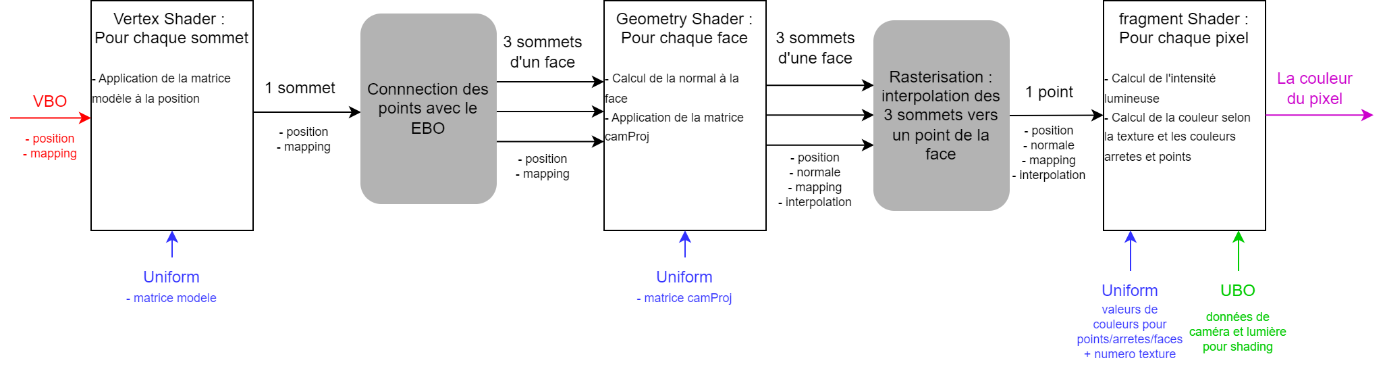


Figure : Enchainement des données avec une texture

Ce cas est très similaire à celui d’un objet coloré par sommet. Il suffit de faire passer l’information de couleur d’un shader à un autre plutôt que l’information de texture. En effet cette donnée ne sert que pour la coloration du pixel dans le fragment shader.

Pour que le shader sache quelle texture appliquée sur l’objet, il faut lui donner son numéro en uniform. Ce passage de donnée est transparent pour l’utilisateur, le programme s’en occupe automatiquement.

## Couleur uniforme, shading lisse

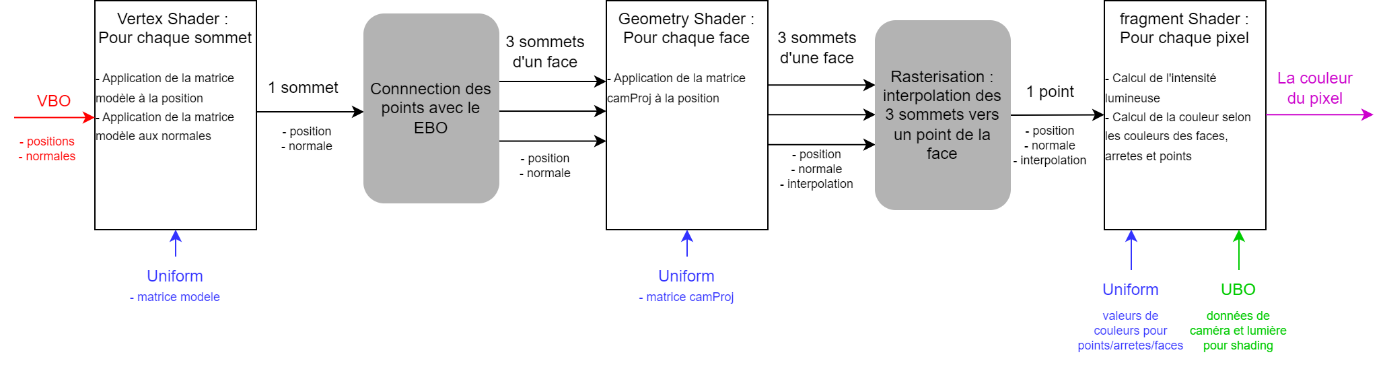


Figure : Enchainement des données pour un affichage lisse

La matrice modèle doit être appliqué aux positions mais aussi aux normales. Si l’objet est tourné par la matrice modèle, logiquement, les normales doivent subir la même rotation.

Le geometry shader est inutile car l’application de la matrice CamProj aurait pu être faite dans le vertex shader. Cependant cette façon de faire à l’avantage de reprendre le même schéma que les autres cas ce qui permet de généraliser tous les cas.

## Généralisation des cas

Comme dit précédemment, il est obligatoire de faire un programme par cas, pour autant les programmes varient uniquement de quelques lignes car la structure reste la même.

C’est pour cette raison qu’il n’y a qu’un seul programme dont certaines lignes sont supprimées pour s’adapter aux choix de l’utilisateur.

Tableau : Vertex shader dans le cas général

|  |  |
| --- | --- |
| Une image contenant texte, capture d’écran, Police, conception  Description générée automatiquement | La partie vertex shader reste très simple. Elle applique la matrice modèle aux positions puis les envoie à l’étape suivante.  Si le VBO contient des couleurs ou du mapping, c’est données sont simplement envoyé à l’étape suivante.  Si le VBO contient des normales alors il faut leur appliquer la matrice modèle avant de les transmettre à l’étape suivante. |

Tableau : Geometry shader dans le cas général

|  |  |
| --- | --- |
|  | C’est dans le geometry shader que l’on choisi l’affichage lisse ou dur.  Si on veut un affichage dur, les normales sont calculées à partir des positions selon un produit vectoriel.  Si l’affichage est lisse, il suffit de récupéré les normales du VBO passé en input. |

Tableau : Fragment shader dans le cas général

|  |  |
| --- | --- |
|  | C’est ici que l’on choisi la couleur de l’objet en fonction du mode choisi (UNIFORME, TEXTURE, PAR\_SOMMET).  On ajoute les arrêtes et les points en fonction du choix de l’utilisateur passé en Uniform.  On calcule l’intensité lumineuse avec la normale et les données dans l’UBO. Il n’y a pas de distinction de calcul entre un affichage lisse ou dur. |

# Les éléments points, textes, lignes

Les éléments lignes et points ne sont pas influencer par la lumière car on ne peut pas calculer de normale. Pour simplifier le code j’ai aussi décidé que les éléments textes ne serait pas soumis aux variations de lumière.

Le programme se trouve nettement simplifier

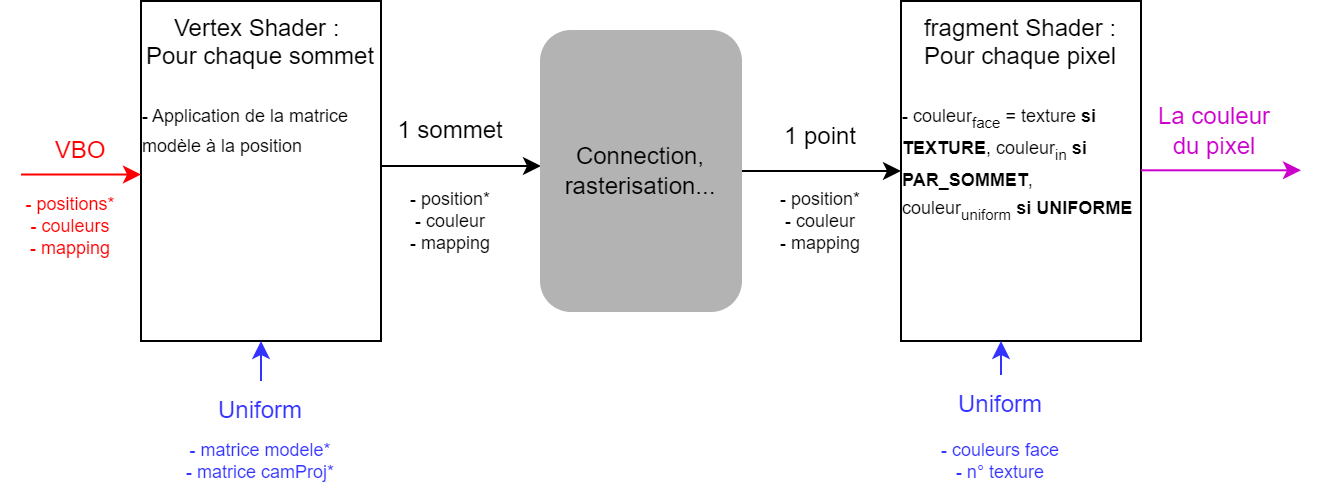


Figure : Enchainement des données pour les objets lignes, points et textes (noLight)

# Shader de sélection

Le shader de sélection est encore plus simple puisqu’on colore l’objet avec son id passé en uniform.

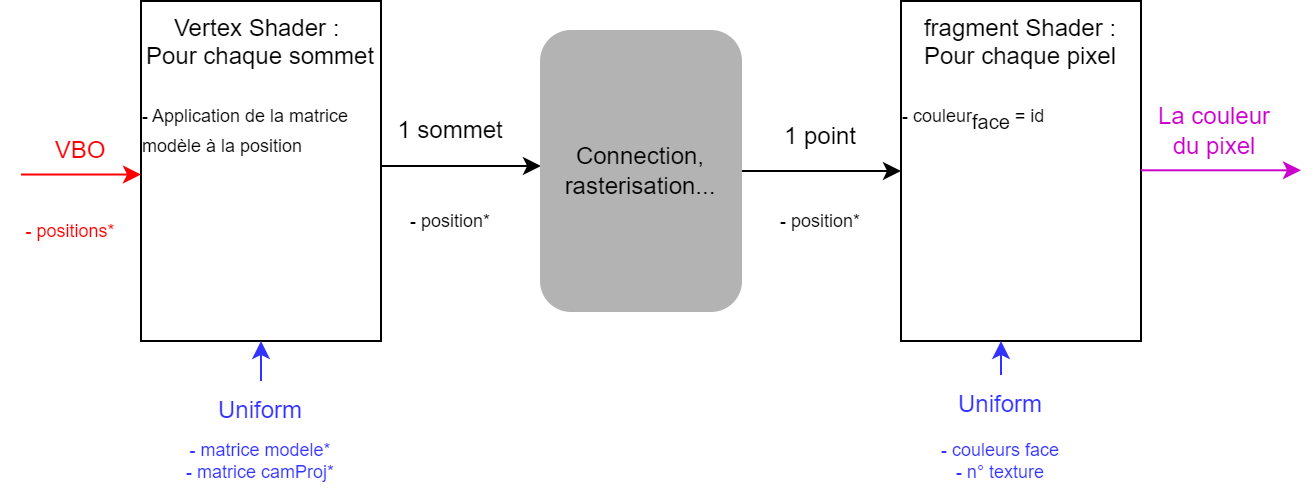


Figure : Enchainement des données pour le shader de sélection (drawId)

# Organisation des données dans le buffer

Les données concernant un objet sont stockées dans une structure OpenGL appelé VAO (Vertex Array Object)

Ces structures contiennent en réalité trois type de données :

* Le VBO (Vertex Buffer Object) : données de tous les sommets (tableau 2D)
* Les vertex attributes : information sur l’organisation du VBO
* Le EBO (Element Buffer Object) : liste de connectivité (tableau 1D)

Un sommet peut contenir plusieurs types d’informations : la position, la couleur, … C’est pour cette raison que les vertex attributes sont nécessaires pour indiquer combien de données sont nécessaires pour chaque information.

Chaque information est numérotée pour être récupérer dans le shader. Ces valeurs ne sont utiles que à OpenGL.

Dans la toolbox, 4 informations peuvent être précisés pour chaque sommet : Sa position (id : 1), sa couleur (id : 2), le mapping pour la texture (id : 3), sa normale (id : 4).

Les données sont placées à la suite dans un tableau 2D.

Quelques exemples :

*Carré simple :*

VBO :

|  |  |  |  |
| --- | --- | --- | --- |
|  | Position x | Position y | Position z |
| Sommet 0 | -1 | 1 | 0 |
| Sommet 1 | 1 | 1 | 0 |
| Sommet 2 | 1 | -1 | 0 |
| Sommet 3 | -1 | -1 | 0 |

EBO :

0 1 2 2 3 0

Vertex Attrib :

id : 1, type d’information : float, nombre : 3, position : 0

*Triangle avec texture & norme :*

VBO :

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Pos  X | Pos  Y | Pos  Z | Mapping X | Mapping Y | Norm X | Norm Y | Norm Z |
| Sommet 0 | -1 | -0.5 | 0 | 0 | 0 | -1 | -0.5 | 0 |
| Sommet 1 | 1 | -0.5 | 0 | 1 | 0 | -1 | 0.5 | 0 |
| Sommet 2 | 0 | 1 | 0 | 0.5 | 1 | 0 | 1 | 0 |

EBO :

0 1 2 2 3 0

Vertex Attrib :

id : 1, type d’information : float, nombre : 3, position : 0

id : 3, type d’information : float, nombre : 2, position : 4

id : 4, type d’information : float, nombre : 3, position : 6

Enfin, l’information de normale n’est présente que pour les normales aux points et peut être calculer automatiquement. Ce calcule est basé sur la normale à la face, pondérée par l’angle.

Dans le cas des normales à la face, le calcul se fait très simplement à partir des valeurs des positions dans le geometry shader. Le stockage des normales à la face impliquerait de stocker plusieurs fois le même sommet. Car un point appartenant à plusieurs face doit être stocké avec les normales de chaque face qu’il construit.