SA4L - Programmation parallèle OpenGL

Géométrie de caméra

1 Espaces

1.1 Espace objet

Lorsqu'on charge un modèle 3D à partir d'un fichier, on récupère les coordonnées des sommets qui le compose (figure 1). Ces coordonnées sont données dans un système d'axes attaché au modèle. On appelle ce système d'axes, l'espace objet.

1.2 Espace monde

L'espace monde est l'espace à partir duquel on positionne les différents objets. On peut placer plusieurs fois un même modèle à des positions différentes dans l'espace monde.
Pour passer des coordonnées objets aux coordonnées mondes on utilise la matrice *ModelMatrix*

$$\begin{pmatrix} x_w \\ y_w \\ z_w \\ t_w \end{pmatrix} = ModelMatrix \begin{pmatrix} x_o \\ y_o \\ z_o \\ 1 \end{pmatrix}$$

Cette matrice est une combinaison de translation et de rotation.

1.3 Espace caméra

Pour pouvoir être projetés sur le plan image, il faut connaitre les coordonnées des sommets dans l'espace caméra. Pour passer des coordonnées mondes aux coordonnées objets on utilise la matrice *ViewMatrix*.

$$\begin{pmatrix} x_{cam} \\ y_{cam} \\ z_{cam} \\ t_{cam} \end{pmatrix} = ViewMatrix \begin{pmatrix} x_w \\ y_w \\ z_w \\ t_w \end{pmatrix}$$

Cette matrice est une combinaison de translation et de rotation. On combine parfois les matrices *ModelMatrix* et *ViewMatrix*.

$$ModelViewMatrix = ViewMatrix \cdot ModelMatrix$$

1.4 Espace de *clipping*

L'espace de clipping est celui qui reçoit les sommets projetés par la caméra. En synthèse d'image la matrice de projection (*ProjectionMatrix*) est une matrice 4×4. Les sommets projetés ont donc toujours 4 composantes.

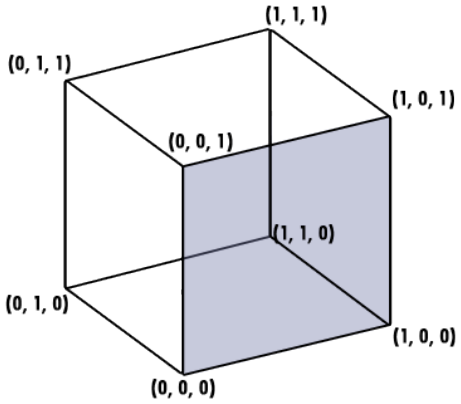

Figure 1: Coordonnées des sommets d'un modèle

$$\begin{pmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ t_{clip} \end{pmatrix} = ProjectionMatrix \begin{pmatrix} x_{cam} \\ y_{cam} \\ z_{cam} \\ t_{cam} \end{pmatrix}$$

Les composantes x_{clip}, y_{clip} et z_{clip} sont utilisées dans les processus de *clipping*. Ce processus sert à éliminer les sommets en dehors du champ de vision de la caméra. En pratique, tous les sommets ayant une valeur de x_{clip}, y_{clip} ou z_{clip} en dehors de l'interval $[0, 1]$ sont éliminés.

1.5 *Normalized Device Coordinates* (*NDC*)

C'est ici qu'on sort des coordonnées homogènes.

$$\begin{pmatrix} x_{ndc} \\ y_{ndc} \\ z_{ndc} \end{pmatrix} = \begin{pmatrix} x_{clip}/t_{clip} \\ y_{clip}/t_{clip} \\ z_{clip}/t_{clip} \end{pmatrix}$$

En synthèse d'image, cette opération est appelée *perspective divide*.

1.6 Espace fenêtre

Les coordonnées *NDC* sont transformées en coordonées sur l'écran via la Viewport Transform.

$$\begin{pmatrix} x_{win} \\ y_{win} \\ z_{win} \end{pmatrix} = \begin{pmatrix} \frac{width}{2} x_{ndc} + x + \frac{width}{2} \\ \frac{height}{2} y_{ndc} + x + \frac{height}{2} \\ \frac{far-near}{2} z_{ndc} + x + \frac{far-near}{2} \end{pmatrix}$$

La composante z_{win} sert dans le processus de *z-buffer*.

2 Pipeline graphique

Le fonctionnement des cartes graphiques est organisé sous forme d'un pipeline. le pipeline graphique est une succèssion d'étapes qui aboutissent à la production d'une image (figure 2).

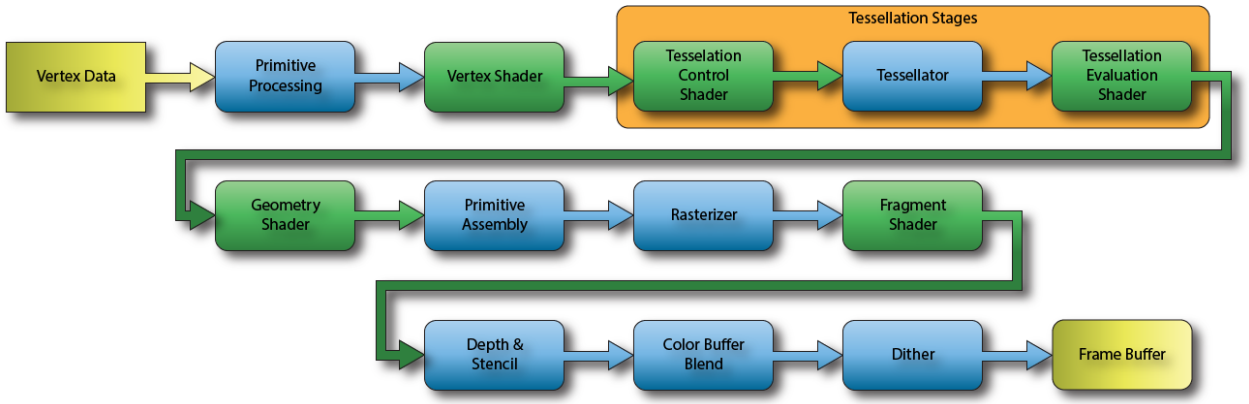

Figure 2: Pipeline OpenGL 4.X

Dans ce pipeline, nous allons détailler les étapes qui nous intéresse le plus.

2.1 *Vertex Data*

Il ne s'agit pas d'une étape à proprement parlé mais bien des données de base sur lesquels va travailler le pipeline. Les *Vertex Data* contiennent toutes sorte de données associées aux sommets. On peut, par exemple citer, les coordonnées de position, les coordonnées de texture, les normales, des couleurs, etc…

2.2 *Vertex Shader*

Le *Vertex Shader* est responsable de la production des coordonnées de clipping. C'est dans le *Vertex Shader* qu'on multiplie les coordonnées des *Vertex Data* par les matrices *ModelMatrix*, *ViewMatrix* et *ProjectionMatrix*.
Plus généralement, on peut y faire tous les calculs qui doivent être effectués sur chaque sommet.

2.3 *Rasterizer*

Sur base des sommets, on définit des primitives à afficher. Les primitives de base sont les points, les segments et les triangles. Pour afficher une primitive, il faut calculer quels pixels de l'image en font partie. Ce processus s'appelle la *rasterization* (figure 3).
Ce calcul se base sur les sommets projetés par le *Vertex Shader* et interpole les données des sommets pour les passer au *Fragment Shader*

2.4 *Fragment Shader*

Le *Fragment Shader* est responsable du calcul de la couleur d'un pixel. Pour effectuer se calcul, il se base sur les données des sommets transmises par *Vertex Shader* et interpolées par le *Rasterizer*.
Plus généralement, on y effectue tous les calculs devant être réalisés sur chaque pixel.

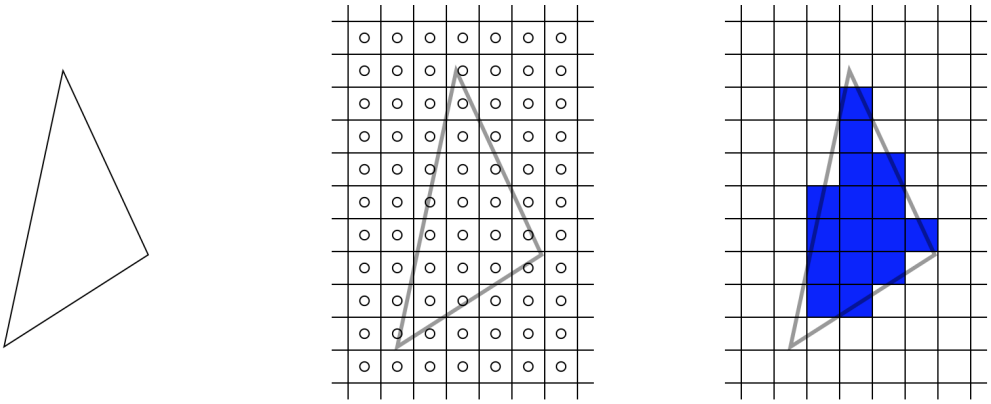

Figure 3: Rasterizer