TP Rasterisation - Compte Rendu

Luc Anchling et Timothée Teyssier

10 Octobre 2023

1 Introduction

Grâce à l'informatique, de nombreuses personnes se sont intéressées à la production d'images à l'aide d'ordinateurs. La production d'images par ordinateur permet de créer de nombreuses applications dans un certain nombre de domaines tels que l'imagerie médicale, les jeux vidéos, l'animation, le cinéma ou bien même la réalité virtuelle.

En ce qui concerne la production d'image, deux possibilités s'offrent alors à nous. La première consiste à se focaliser sur la production d'images rapide, en grande quantité sans forcément obtenir la meilleure qualité pour optimiser le temps de calcul sur machine. Ce processus est par exemple utilisé dans les jeux vidéos, où nous cherchons à obtenir un temps de calcul faible pour maximiser le nombre d'images affichées au joueur par seconde. La deuxième consiste à produire des images plus réalistes, plus jolies mais qui sont plus coûteuses à créer. On utilise cette méthode pour l'animation ou le cinéma.

Lors de ce TP, nous nous sommes intéressés au rendu de la rasterisation et le rendu projectif. La rasterisation est un procédé qui consiste à convertir une image vectorielle en une image matricielle. Cela signifie que nous pouvons transformer une scène 3D en une image (pixels) sur un écran. Dans ce TP, nous implémentons un programme permettant d'afficher dans un premier temps un segment, puis un triangle, puis un mesh tout en y ajoutant la couleur, la texture et l'illumination. Le tout pour comprendre et mettre en place les différentes étapes intervenant dans une API tel que *OpenGL*.

2 Affichage d'un segment

2.1 Affichage d'un segment vertical ou horizontal

Dans un premier temps, nous cherchons à afficher un segment sur notre image. Nous devons tout d'abord définir nos deux points en créant des **ivec2** permettant ainsi de renseigner les positions 2D de nos deux pixels. Nous souhaitons également tracer ce segment en ajoutant un dégradé de couleur du noir au blanc (le premier pixel contient donc **color(0.0f,0.0f,0.0f)** et le deuxième pixel **color(1.0f,1.0f,1.0f)**). Nous penserons également à bien convertir nos images **.ppm** en **.png** ce qui nous permettra de stocker nos images ayant une plus faible taille et ne perdant aucune informations contrairement à **.jpg** ou **.jpeg**. Nous créons notre ligne grâce à la fonction **drawline** qui permet de construire une ligne discrète entre deux points.

On calcule également la couleur en fonction de la position du pixel dans le segment, ce qui nous permet ainsi de créer notre dégradé entre le noir de notre premier pixel et le blanc de notre deuxième pixel. Après l'avoir créée, nous dessinons cette ligne sur un objet **image** dérivant de la classe **image basic**. Les couleurs sont stockées en internes grâce à 3 floats R,G,B (même si nous pouvons créer une couleur avec qu'un seul float (gray), la couleur sera stockée avec 3 floats identiques : gray,gray,gray).

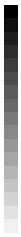

Figure 1: Segment vertical entre deux pixels

2.2 Affichage d'un segment dans le premier octant

Nous nous intéressons maintenant à tracer un segment en diagonale dans le premier octant, nous utilisons la structure **ligne discrete** afin d'obtenir la liste des coordonnées de tous les points de la diagonale.

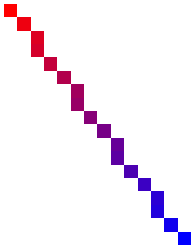

Figure 2: Segment entre deux pixels se trouvant dans le premier octant

2.3 Affichage d'un segment dans n'importe quel octant

Nous cherchons maintenant à afficher un segment entre deux pixels ne se trouvant pas dans le premier octant, pour cela nous utilisons l'algorithme de Bresenham, qui va nous permettre de ramener le problème au cas d'un calcul du premier octant pour ensuite le re-projeter dans son octant d'origine.

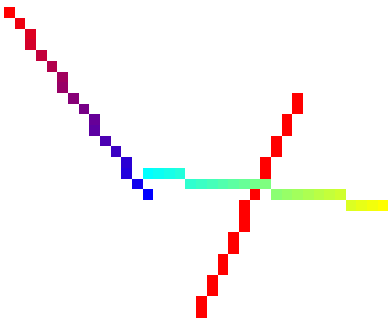

Figure 3: Segment entre deux pixels se trouvant dans n'importe quel octant

2.4 Interpolation des couleurs

Nous implémentons également une méthode permettant de tracer une ligne entre deux points *p0* et *p1* telle que la couleur varie linéairement entre *c0* et *c1*. Nous allons associer un poids variant de 0 à 1 qui sera le paramètre d'interpolation. Cela nous permet d'observer nos segments avec une couleur qui varie linéairement tout au long du segment.

3 Affichage d'un triangle

3.1 Affichage d'un triangle de couleur unie

Après avoir pu tracer un segment dans n'importe quel octant, nous cherchons maintenant à tracer et afficher un triangle composé de trois points *p0*, *p1*, *p2* et d'une couleur unie *c*. Pour cela nous devons apprendre à remplir le triangle de la couleur unie *c*. Nous utilisons ainsi l'algorithme scanline qui consiste à parcourir les 3 cotés du triangle, et, pour chaque ligne rencontrée (paramètre y) stocker la position la plus à gauche et la plus à droite (paramètre x minimal et maximal) sur le coté du triangle. Ensuite chaque ligne appartenant au triangle est parcourue et un segment horizontal est tracé entre le point le plus à gauche et le point le plus à droite.

Nous pouvons créer un scanline grâce à la fonction **triangle scanline factory** prenant en paramètres les trois sommets du triangle *p0*, *p1*, *p2* et la couleur **color c** pour les trois sommets. Si nous affichons en sortie dans le terminal le scanline, nous allons obtenir une liste de tous les points appartenant au triangle contenant pour chaque point, la position et la couleur.

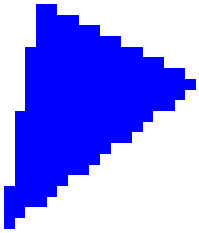

Figure 4: Triangle de couleur unie

Nous pouvons également ajouter une couleur différente à chaque sommet, la couleur sera interpolée dans la fonction en prenant pour chaque ligne du triangle la couleur du point le plus à gauche et la couleur du point le plus à droite, puis la couleur va varier linéairement sur cette ligne (segment). Nous pouvons observer le résultat sur la figure 5.

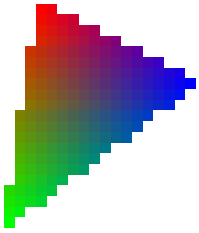

Figure 5: Triangle avec une couleur différente pour chaque sommet

4 Gestion de la profondeur

Pour l'instant, les triangles sont définis dans un espace 2D et sont affichés dans l'ordre de leur appel respectif. Nous allons maintenant décider d'ajouter un paramètre permettant de gérer la profondeur et ainsi de pouvoir placer nous mêmes les triangles que l'on souhaite au premier plan et ainsi pour les autres

plans. Pour cela nous utilisons une carte de profondeur appelée **zbuffer** qui va permettre de stocker pour chaque pixel un float correspondant à la profondeur de l'objet se trouvant au premier plan sur le pixel. Les valeurs de z (profondeur) sont stockées en float entre -1 et 1 sachant que -1 correspond au plan le plus proche et 1 au plan le plus loin. Par défaut, la valeur initialisée du *zbuffer* est à 1 soit le plan le plus profond.

Pour ajouter la profondeur, nous devons déjà modifier la fonction **draw point** de manière à ce que le point ne s'affiche que si la profondeur à afficher est inférieure à la profondeur déjà contenu dans le *zbuffer* à la même position.

On gère l'interpolation de la profondeur de la même façon que nous interpolons la couleur dans la fonction **draw line depth**. Nous pouvons apercevoir sur la figure 6 deux triangles positionnés de manière à ce qu'ils se croisent, cela nous permet de montrer l'importance de la profondeur lors de l'affichage prochain de mesh contenant beaucoup plus de triangles à afficher.

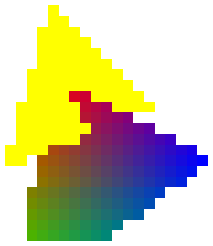

Figure 6: Z-fighting entre deux triangles

5 Projection

Pour l'instant, nous affichions des triangles définis dans un espace 2D. Nous allons maintenant considérer des triangles ayant des coordonnées dans l'espace 3D, nous souhaitons être capable d'afficher ces triangles dans l'espace 2D. Nous allons introduire trois matrices (Model, View, Projection) qui vont nous permettre de réaliser notre projection :

- **Model** : gestion des transformations directes sur l'objet (rotation, translation), cela permettra de placer notre objet.
- **View** : gestion du placement de la caméra en utilisant des rotations et des translations.
- **Projection** : gestion de la projection de la scène 3D dans l'image 2D.

Coordonnées projetées = Projection * View * Model * Coordonnées objet
Nous pouvons ainsi obtenir le résultat suivant.

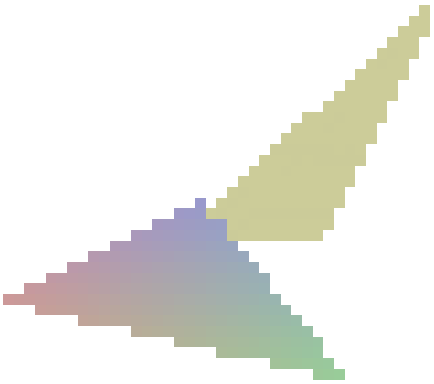

Figure 7: Projection d'un triangle 3D sur une image 2D

Nous pouvons alors constaté que toutes les propriétés que nous avons ajouté jusqu'à maintenant fonctionnent lorsque nous projetons des triangles en coordonnées 3D sur une image 2D. Il nous manque plus qu'à gérer l'illumination.

6 Illumination de Phong

Pour cela nous utilisons l'illumination de Phong qui va permettre de calculer la couleur d'un sommet suivant 3 paramètres : l'illumination ambiante I_a (éclairage homogène), diffuse I_d (effet de profondeur) et spéculaire I_s (effet de brillance). Nous calculons l'illumination de chaque sommet dans la fonction *vertex shader*. Nous ajoutons trois normales pour nos trois sommets. Nous pourrons l'observer facilement sur l'affichage du mesh dans la partie suivante.

7 Affichage d'un Maillage

Nous cherchons maintenant à afficher un maillage, c'est-à-dire un ensemble de triangles ayant des coordonnées 3D, pour cet exemple nous prendrons comme exemple le fichier *dino.obj*. Pour afficher le mesh, nous savons déjà comment afficher des triangles ayant des coordonnées 3D en y ajoutant la couleur, la profondeur, l'illumination, sachant qu'un maillage est un ensemble de triangles 3D, nous créons une boucle qui va au fur et à mesure ajouter chaque triangle dans la scène 3D que nous allons projeter sur notre image 2D. Nous obtenons ainsi le résultat suivant.

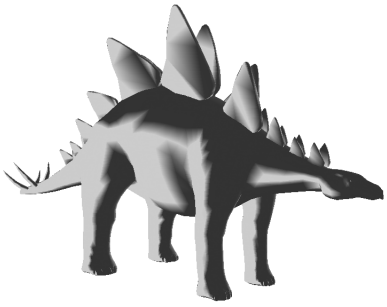

Figure 8: Affichage d'un maillage, dino.obj sans texture

8 Texture

Nous pouvons également ajouter la texture à notre mesh, nous rajoutons donc l'information de la texture à chaque point 3D afin de l'interpoler avec la lumière. Pour récupérer les informations de textures (coordonnées de textures) nous utilisons la classe *texture*. Ces coordonnées de textures sont à interpoler pour obtenir une texture ayant des détails précis et non-lisses. Nous pouvons alors observer le résultat sur la figure 9.

Figure 9: Affichage d'un maillage, dino.obj avec texture

9 Plan

Concernant la texture pour le modèle *plane.obj*, on obtient la figure 10. On observe sur la figure 10 que notre texture est mal affichée sur le plan, en effet ici nous pouvons voir une diagonale bien visible partant du bas à gauche jusqu'à droite en haut, nous obtenons ce résultat car le plan est un carré, divisé en deux triangles, les deux triangles ont pu être bien affichés, or cependant la texture n'est pas bonne, puisque nous devrions obtenir un damier. Nous n'avons pas réussi à déterminer d'où pouvait venir le problème.

Figure 10: Affichage d'un maillage, plane.obj

10 Conclusion

Lors de ce tp nous avons appris une première méthode permettant ainsi de créer une image à partir d'une scène 3D. Nous avons appris à tracer un segment, puis un triangle, nous avons également appris à remplir ce triangle, tout en y ajoutant la texture si voulue. Nous avons pu afficher des maillages tels que le dinosaure ou bien le plan. Cependant cette méthode à certaines limites pour tracer une sphère, dans le but de tracer une sphère, nous sommes obligé de créer un maillage de la sphère à partir de triangles ce qui permet d'obtenir un très bon réalisme au niveau de nos résultas.