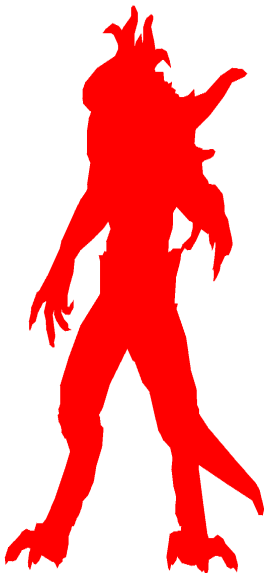
Bordeaux INP
ENSEIRB
MATMECA

Conception et optimisation d'un moteur 3D en logiciel

Module EN212



MOUHAGIR Ayoub
FERIAT Abdessamad

Encadré par :
CRENNE Jérémie

Introduction

Ce module a pour objectif d'introduire les rudiments de l'affichage 3D sans aucune bibliothèque tiers et les méthodes d'optimisation associées. Nous utiliserons quand même deux petites bibliothèques : SDL (Simple Direct Layer) pour ouvrir une fenêtre et la manipuler et STBI pour charger des images.

Le travail demandé consiste à réécrire les fonctions de bas niveau qui résument en quelque sorte les fonctionnalités de base des bibliothèques traditionnellement utilisées dans le domaine (OpenGL, Direct3D,..). Ce travail sera fait étape par étape en partant d'un squelette du programme proposé qu'on peut modifier et améliorer à notre guise.

08/04/2019

Après avoir étudié et traité les différents fichiers sources donnés dans le squelette pour avoir une idée générale du projet. Nous avons été demandés à manipuler des fichiers du format **.OBJ** qui contiennent la description d'une géométrie 3D. Ils sont à la fois simples et très communs dans le domaine. Les formes géométriques peuvent être définies par des polygones ou des surfaces décrites un ensemble de sommets (accompagné de coordonnées de textures et de normales en chaque sommet) et d'un ensemble de faces contenant les indices des coordonnées en question.

Notre première tâche consiste à récupérer toutes ces coordonnées et les stocker dans des listes prédéfinies. Pour ce faire, on a complété la fonction **ModelLoad** du fichier *model.c* qui permet simplement de changer la « forme » des données, d'une chaîne de caractères en un ensemble de vecteurs.

La seule difficulté que représente cette tâche est la manière dont on stocke les données dans les listes pour pouvoir retirer après les bonnes coordonnées de chaque sommet en parcourant les indices dans chaque face. En plus, comme on manipule des pointeurs, il est indispensable de louer la mémoire avec *malloc()* pour s'assurer à ne pas écraser des informations.

Enfin, on vérifie le bon fonctionnement de notre approche en parcourant et en affichant le contenu des différentes listes constituées après le chargement d'un fichier **.obj** (exemple : *head.obj*).

15/04/2019

L'affichage d'un dessin plus élaborée des objets en 3D passe irrémédiablement par l'utilisation de primitives d'affichage très simples : le point, la ligne et enfin le triangle. Ces primitives ont la particularité d'être appelées des millions de fois par seconde pour des scènes complexes. D'où la nécessité d'écrire des versions plus optimisées que possible.

Le point d'arrivée dans cette partie est d'afficher des triangles remplis d'une couleur en paramètre. Alors, on procède étape par étape vu qu'un triangle rempli est un ensemble de lignes, qui sont à leur tour constitués d'un ensemble de points.

On commence par l'implémentation de la fonction *WindowDrawPoint(x, y , r, g, b)* qui consiste simplement à remplir le **framebuffer** de la fenêtre avec la bonne couleur codée en **Alpha_rgb** : **((0xFF « 24) | (r « 16) | (g « 8) |**

b) (Alpha est pris pour l'instant égale à 0xFF). On vérifie le bon fonctionnement du notre implémentation en affichant une fenêtre à une couleur uniforme avec la fonction **WindowDrawClearColor** en parcourant tous les pixels de la fenêtre, et on appelle pour chaque pixel (x,y) notre fonction de **WindowDrawPoint**.

En deuxième étape, on essayera d'afficher une ligne entre deux points (x_0, y_0) et (x_1, y_1). La première idée qui vient à l'esprit c'est de calculer la pente de la ligne et puis évaluer les ordonnées/abscisses correspondantes à l'aide de l'équation caractéristique d'une ligne en parcourant les abscisses/ordonnées. Du coup, on aura tous les points (x,y) constituant la ligne qu'on affiche à l'aide de **WindowDrawPoint**. Enfin, lorsqu'on parcourt les abscisses ou les ordonnées, il faut faire attention de bien partir de la plus petite à la plus grande et aussi aux cas particuliers où les deux points ont la même abscisse ou la même ordonnée. Tous cela sera fait simplement à l'aide des tests.

Notre approche semble bien fonctionnelle avec une complexité tolérable. Ce code reste sujet d'une optimisation pour gagner en performances. C'est pour cela on utilisera l'algorithme de Bresenham qui détermine quels sont les points d'un plan discret qui doivent être tracés afin de former une approximation de segment de droite entre deux points donnés. Pour mettre en application notre implémentation, on affiche les modèles 3D en fil-de-fer (wireframe).

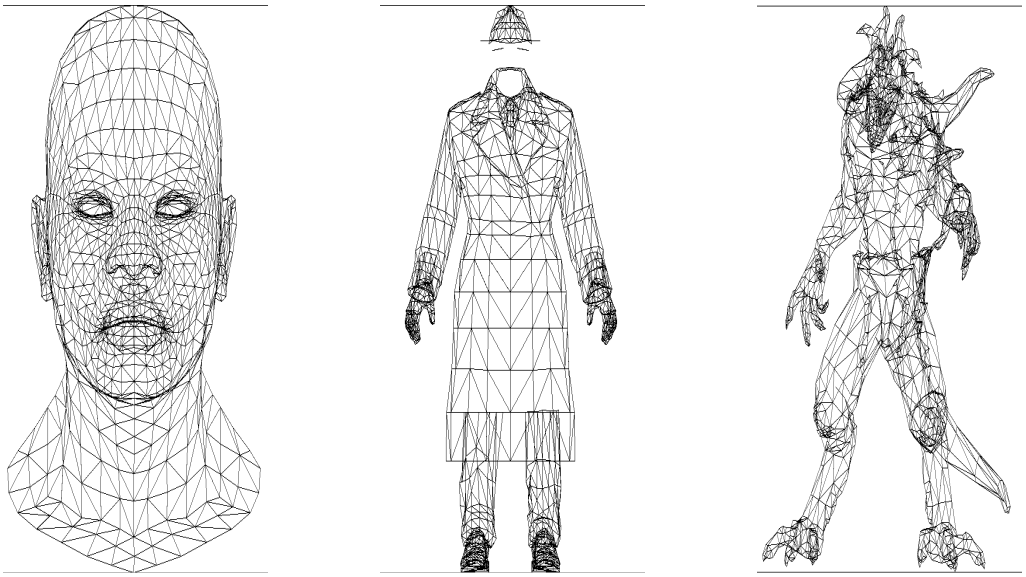


FIGURE 2 – Affichage des 3 modèles en fil-de-fer

21/04/2019

Maintenant que nous avons un programme pour dessiner une ligne entre deux points donnés qui fonctionne parfaitement. L'affichage d'un triangle devient très simple comme étant trois lignes reliant trois points.

Jusqu'à présent, compte tenu des toutes les fonctions réalisées, on sera capable d'afficher l'un des objets 3D proposés en fil de fer (wireframe), c'est-à-dire sans remplissage des triangles. Nous procédons donc à compléter la fonction **WindowDraw-**

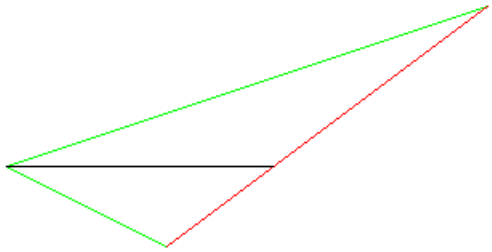


FIGURE 3 – Dessin d'un triangle avec ses limites gauche et droite

Triangle() qui nous permettra de tracer un triangle rempli de couleurs passés en argument. Ici, on appliquera la méthode de *line sweeping* qui consiste à :

- Trier les sommets selon leurs abscisses en utilisant la fonction **Vec3iSwap()** de tel manière que le premier point donné en argument de la fonction **WindowDrawTriangle()** est celui dont l'abscisse le plus grand.
- Balayer simultanément les côtés de droite et de gauche du triangle.
- Dessiner un segment horizontal entre les deux limites du triangle.

Ceci se fait en deux temps (deux boucles for), en traitant la partie entre t0 et t1 puis celle entre t1 et t2 (on coupe le triangle horizontalement).

Notre implémentation fonctionne. Ceci est vérifié en affichant l'objet 3D "head.obj" en passant en argument de la fonction **WindowDrawTriangle()** des couleurs aléatoires en utilisant la fonction **rand()**.

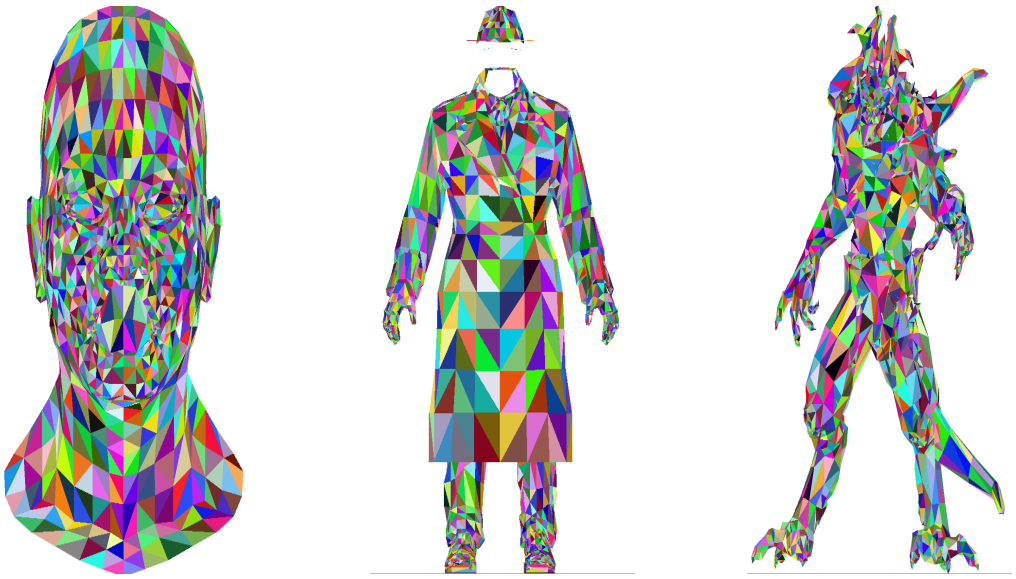


FIGURE 4 – Affichage des 3 modèles avec des triangles remplis des couleurs aléatoires

28/04/2019

Dans le but d'obtenir un rendu plus réaliste et donner la profondeur à nos objets 3D, nous ajouterons une source de lumière. On définit alors un vecteur représentant la position de notre source de lumière. Â partir du principe suivant : Un triangle reçois plus de lumière lorsqu'il est orienté orthogonalement au vecteur de lumière.

Pour coder cet effet, il faut calculer pour chaque triangle son vecteur normal à l'aide du produit vectoriel de ses deux côtés qui se calculent eux mêmes en soustrayant les coordonnées de ses sommets. La normale d'un triangle représente un vecteur de longueur 1 perpendiculaire à ce triangle, d'où l'intérêt de normaliser le résultat du produit vectoriel. Ensuite, le paramètre intensité de lumière est égale au produit scalaire entre le vecteur de direction de la lumière et la normale du triangle. Sachant que ce produit scalaire peut être négatif, cela signifie que la lumière provient de derrière le triangle. Alors, ce triangle peut être écarter de l'affichage (Intérêt de faire un test sur le signe du paramètre intensité) . Cette technique nous permet de supprimer les triangles non visibles.

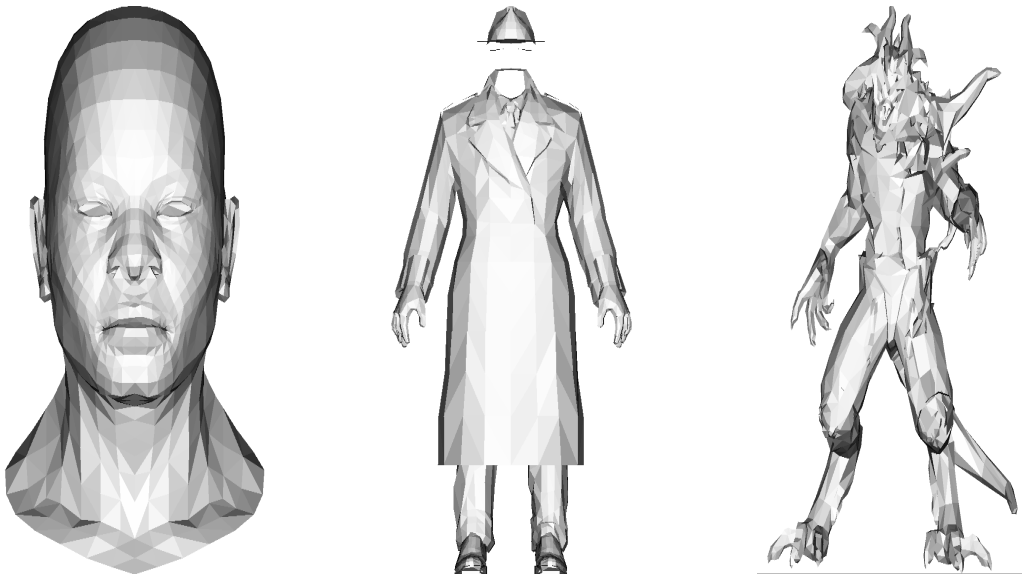


FIGURE 5 – Affichage des 3 modèles 3D en rajoutant de la lumière

02/05/2019

Dans cette partie, nous allons coder l'information de profondeur (z_buffer) pour résoudre le problème des faces dessinées en avant d'autres.

Avant de , on va appliquer ce principe sur 2D (y_buffer). On déclare un tableau flottant de dimension (Width,1) que l'on initialise à la valeur moins infinie. Ensuite, on parcourt toutes les coordonnées x entre les abscisses de mes deux points et calcule la coordonnée y correspondante du segment. Puis, on vérifie ce que nous avons dans notre tableau ybuffer avec indice x actuel : Si la valeur y actuelle est plus proche de la caméra que la valeur du ybuffer, alors je dessine le pixel à l'écran et met à jour le ybuffer.

En partant du même principe, on définit un tableau z_buffer de dimension (width,height) que l'on initialise à la valeur moins infinie. Pour faciliter la manipulation du tableau bidimensionnel, on le considère comme un tableau unidimensionnel avec la conversion suivante : $indice = x + y * width$.

La seule difficulté est de calculer la valeur z d'un pixel que nous voulons dessiner sachant qu'on a que les z des trois sommets du triangle. Ce problème peut être résolu à l'aide de calcul du barycentre. D'où, pour chaque pixel que nous voulons dessiner, il suffit de multiplier ses coordonnées barycentriques par les valeurs z des sommets du triangle.

Notre premier essai pour le z_buffer consiste à utiliser la méthode de rectangle (Box Method) qui est le plus petit rectangle pouvant contenir le triangle. On parcourt ce rectangle pixel par pixel et à l'aide des coordonnées barycentriques on pourra tester si le pixel courant appartient au triangle ou pas : Tous les points qui appartiennent à l'intérieur du triangle, ses coordonnées barycentriques sont positives. Du coup, on teste à chaque fois le signe de ces coordonnées, si la condition est satisfaite on dessine le pixel en prenant en considération sa profondeur dans le z_buffer. Si non, on cherche le prochain pixel.

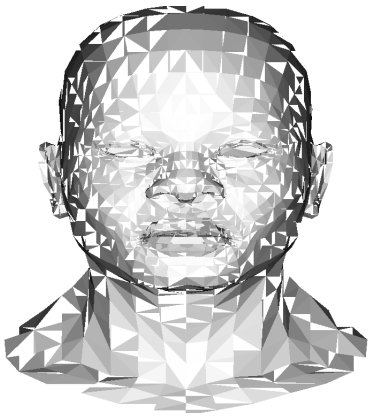


FIGURE 6 – Premier essai pour implémenter la profondeur

Cette approche nous a permis d'avoir à peu près le résultat prévu, mais il existe plusieurs triangles qui ne sont pas tracés. Ce problème peut venir de notre méthode de calcul du barycentre qui n'est pas adaptable pour tous les cas.

Un deuxième essai (**WindowDrawTriangle_prof**) consiste à utiliser la méthode de Line sweeping vu précédemment en ajoutant dans la fonction **WindawDrawPoint** un argument pour introduire l'information z_buffer. Dans ce cas, pour chaque pixel du triangle, sa valeur z reste constante calculée par la moyenne des valeurs z des sommets du triangle.

Cette deuxième méthode, même qu'elle reste un peu moins efficace au terme de précision, nous a permis d'avoir un très bon résultat.

06/05/2015 & 13/05/2019

On souhaite ajouter des textures à nos objets 3D pour avoir une représentation plus réaliste. Ceci se fait en récupérant les coordonnées des textures et en les interpolant dans le triangle. Ainsi nous procédons par la méthode du calcul du barycentre de la coordonnée de la texture qui se trouve dans un triangle quelconque.

On multiplie alors chaque coordonnée des textures par le barycentre correspondant et on stock le résultat dans la variable `tab_text_coord`, puis on multiplie par la taille de l'image utilisée comme texture de la manière suivante :
$pixelOffset = data\_txt + (tga\_width - tab\_text\_coord.x + (tga\_hight - tab\_text\_coord.y) * tga\_width) * 4$

On multiplie par 4 à la fin comme chaque pixel est codé sur 4 octets R, G, B et α.

Un premier essai avec la méthode de rectangle (Box method) du z-buffer donne une représentation incomplète de l'objet 3D, où on remarque l'absence de plusieurs triangles qui n'ont pas été dessiné. Pourtant, les textures sont très bien interpolées.

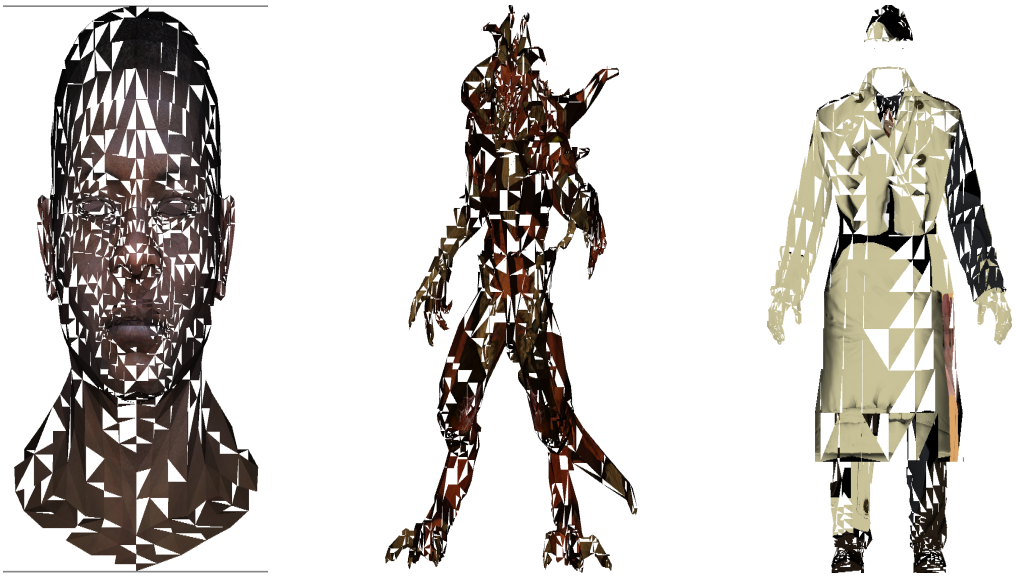


FIGURE 7 – Affichage des 3 modèles 3D avec gestion de profondeur et texture en utilisant une première implémentation du barycentre

Dans un deuxième essai, en partant toujours de la méthode du line sweeping avec z_buffer, on a modifié la manière dont on calcule le barycentre, car la première n'était pas assez précise vu qu'il fallait passer par une conversion $float \Rightarrow int$.

On remarque un très bon rendu d'image, après avoir changer la façon dont on calcule les coordonnées barycentriques. Pourtant, lorsqu'on essaye d'afficher les deux autres objets, on trouve un bogue qui résulte d'un indice plus grand que la taille du tableau des couleurs de textures. Ce problème vient sûrement d'un cas particulier dans le calcul des coordonnées barycentriques.

FIGURE 8 – Modèle "head" affiché avec une source de lumière, texture et gestion de profondeur

Conclusion

Dans ce journal de bord, nous avons décrit toutes les étapes dont on a passé pour achever la conception du moteur 3D sans aucune bibliothèque tiers d'affichage. Le projet a commencé par l'implémentation des bases de l'affichage 3D : le point, la ligne et le triangle. Ces derniers nous ont permis un premier affichage en fil de fer (wireframe) des objets 3D. Il est aussi indispensable de pouvoir remplir (colorier) le triangle pour pouvoir afficher un objet d'une manière complète, en utilisant la méthode du *line sweeping*. Ensuite, pour aboutir à un affichage réaliste de l'objet, il faut rajouter une source de lumière ainsi que supprimer les faces cachés. Finalement, on rajoute des textures à l'objet.

Malheureusement, nous n'avons pas pu aborder la partie d'optimisation du calcul en utilisant des méthodes de SIMD (Single Instruction Multiple Data). Ça aurait pu être très intéressant de voir l'effet du parallélisme sur la rapidité dans le calcul effectué par le CPU, durant l'affichage des objets 3D, vu que ceci utilise une multitude d'opérations calculatoires pour produire le rendu final de l'image.