Projet : Conception et Optimisation d'un Moteur 3D en Logiciel

Jérémie Crenne

Avril 2019

1 Introduction

Ce module prends la forme d'un petit projet en binôme ayant pour but d'introduire les rudiments de l'affichage en 3D et les méthodes d'optimisations associées. Il s'agit à la fois d'un module de découverte et de consolidation de vos acquis en programmation. L'idée est de vous montrer comment implémenter un moteur 3D sans aucune bibliothèque tiers d'affichage. Même si nous aurions pu nous en passer, nous utiliserons tout de même la bibliothèque SDL (Simple Direct Layer) pour ouvrir une fenêtre et la manipuler, et une petite bibliothèque pour le chargement d'images. L'effort demandé dans ce projet peut paraître important car cela demande une réécriture quasi intégrale des

Figure 1: Un aperçu de l'objectif de ce projet. En fin d'exercice vous devriez être capable d'afficher un objet en 3D avec un rendu de cette qualité. Et surtout, vous serez capable de comprendre entièrement comment cela fonctionne !

fonctions de bas niveau, mais c'est également un exercice valorisant qui permet d'entrer dans les arcanes des bibliothèques traditionnellement utilisées dans le domaine (OpenGL, Direct3D...). Bien évidemment, nous ne tirerons pas parti de l'accélération matérielle proposée par les GPU dont s'est traditionnellement le rôle, puisque l'implémentation se fera purement sur CPU.

2 Déroulement du projet et évaluation

Vous serez guidé tout au long de votre progression par votre enseignant, il est là pour ça ! Pour vous aider à démarrer, un dossier contenant le squelette du programme du projet vous est proposé. Ce code peut être amélioré et vous pouvez le modifier à votre guise. Il contient des trous que vous devrez remplir au fur et à mesure de votre progression. L'évaluation de vos acquis se fera par le biais de la remise d'un rapport d'une dizaine de pages ayant la forme d'un journal de bord.

3 Structure du répertoire du projet

Le dossier du projet contient :

- Un fichier "makefile" pour la construction du programme
- Un dossier "bin" contenant :
  - le binaire du programme après sa construction avec le fichier "makefile"
  - un dossier "data" contenant trois fichiers .obj décrivant des objets 3D et trois fichiers .tga qui sont les fichiers images des textures associés à ces objets
- Un dossier "src" contenant l'ensemble du code source du projet avec comme fichiers (.c et .h) :
  - **"events.c" ("events.h")**
    Implémente la gestion des évènements liés à la fenêtre (ouverture, redimensionnement, fermeture...)
  - **"geometry.c" ("geometry.h")**
    Implémente les opérations mathématiques de base pour la manipulation d'objets 3D
  - **"model.c" ("model.h")**
    Implémente le chargement d'objets 3D

  - **“vector.c” (“vector.h”)**
    Implémente les fonctions pour manipuler des tableaux avec allocation dynamique

  - **“window.c” (“window.h”)**
    Implémente l’affichage et le rendu de la fenêtre

  - **“stb_image.h”**
    Implémente le chargement d’image au format .tga. C’est une fichier tiers (`https://github.com/nothings/stb`)

  - **“main.c”**
    Source principal du projet

- Un dossier “obj” contenant les fichiers objets générés lors de la compilation du projet

4 Description d’un objet en 3D

4.1 Principes

Un objet 3D peut être représenté très simplement. Pour ce faire, quelques notions doivent être définies :

- Les sommets : aussi appelés vertices (vertex au singulier). Les sommets sont les éléments de base qui représentent les points associés à l’objet. Ils n’indiquent cependant pas comment ils sont connectés entre eux. C’est le rôle des faces
- Les faces : les faces sont de simples polygones constitués d’un ensemble d’indices représentant des vertices. Une face n’est rien d’autre qu’une référence à un ensemble de trois sommets. Elle représente donc un triangle
- Les normales : les normales permettent de calculer la contribution lumineuse d’une source sur l’objet. Elle permettent de renforcer son aspect réaliste
- Les coordonnées de textures : un objet est souvent constitué d’images pour lui donner un aspect plus vivant. Les coordonnées de textures sont utilisées pour lier un sommet à une coordonnée (un pixel) dans une image

4.2 Le format de fichier .obj

Le format de fichier .obj est largement exploité dans le monde de la 3D pour sa simplicité puisqu’il permet de décrire un objet à l’aide d’un simple fichier texte ASCII. Ce fichier constitué de quelques listes de données :

1. Une liste de sommets dont les lignes commencent par le caractère v. Chacune des lignes décrit les coordonnées en (x, y, z) d'un sommet

   ex : v 1.0 2.0 3.0

2. Une liste de normales dont les lignes commencent par les caractères vn. Chacune des lignes décrit la normale en (x, y, z) d'un sommet

   ex : vn 0.7 0.0 0.1

3. Une liste de coordonnées de texture dont les lignes commencent par les caractères vt. Chacune des lignes décrit les coordonnées en (u, v) d'une texture (valeurs entre 0.0 et 1.0)

   ex : vt 0.5 0.0

4. Une liste de faces dont les lignes commencent par le caractère f. Une face est composée d'un triangle représenté par trois couples de trois éléments chacun, séparés par le symbole / : l'indice d'un sommet v, l'indice d'une normale vn et l'indice d'une coordonnées de texture vt. Attention les indices commencent à 1 et non pas à zéro

   ex : f 4/1/3 5/2/4 6/3/5 (f v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3)

Il convient donc, pour exploiter un tel fichier, de constituer une liste pour les sommets, une pour les normales, une pour les coordonnées de textures et enfin une dernière pour les faces de l'objet.

1. Complétez la fonction **ModelLoad** du fichier "model.c" qui implémente le chargement d'un fichier au format .obj. Pour ce faire, vous pouvez vous aider des fonctions du fichier "vector.c" qui permettent de manipuler des listes. Souvenez vous également des fonctions utiles pour la manipulation des fichiers.

2. Vérifiez le bon fonctionnement de votre implémentation en parcourant et en affichant le contenu des différentes listes constituées après le chargement d'un des fichiers .obj

5 Les primitives d'affichage

Le dessin du plus élaboré des objets en 3D passe irrémédiablement par l'utilisation de primitives d'affichage très simples : le point, la ligne et enfin le triangle. Cela peut paraître évident et simple à implémenter, mais il n'en est rien. De telles fonctions ont la particularité d'être appelées des millions voire des milliards de fois par seconde pour des scènes complexes. Il convient donc d'en écrire des versions optimisées pour éviter une déperdition du nombre d'images affichables par secondes.

5.1 Le point

La première étape consiste à afficher un point dans une fenêtre. Cela sera notre primitive de base pour pouvoir dessiner n'importe quelle autre forme géométrique.

1. Complétez la fonction **WindowDrawPoint** du fichier "window.c" qui implémente l'affichage d'un point coloré en (x, y). Le format de pixel utilisé pour la couleur est le format 32-bit BRGA qui code chacune des composantes sur 8 bits. Le bleu (B) est codé dans les bits de poids faibles et la transparence dans les bits de poids forts. Il vous ai demandé de neutraliser la composante de transparence en positionnant sa valeur à 0xFF

2. Testez le bon fonctionnement de votre proposition en implémentant la fonction **WindowDrawClearColor** qui utilise **WindowDrawPoint** et doit effacer entièrement la fenêtre avec une couleur uniforme

5.2 La ligne

Deuxième étape pour dessiner un objet en 3D : l'affichage d'une ligne. Pour ce faire, nous utiliserons la fonction de tracé de point **WindowDrawPoint** écrite précédemment.

1. Proposez votre propre implémentation de la fonction **WindowDrawLine** du fichier "window.c" qui permet le tracé d'un segment de droite en couleur et entre les points ($x0$, $y0$) et ($x1$, $y1$)

2. Testez le bon fonctionnement de votre proposition en effectuant un tracé de lignes dans les cas usuels

3. Implémentez maintenant l'algorithme de tracé de Bresenham qui permet un tracé de ligne optimisé

Une fois le tracé cohérent, nous pouvons d'ores et déjà commencer à évaluer les performances de notre programme. L'idéal serait de pouvoir distinguer facilement quels sont les points chauds (ou hot spots), c'est à dire les régions exécutant un nombre important d'instructions. C'est exactement le rôle des outils de debug gdb (GNU DeBugger) et de profiling gprof (GNU PROFiling). Pour les utiliser, il faut modifier le fichier "makefile" fournit en ajoutant (ou en modifiant) les options de linker g++ suivantes :

- *-ggdb* pour indiquer au compilateur qu'il doit générer des informations de debug spécifiques à gdb

- *-g* pour indiquer au compilateur qu'il doit générer les informations de debug généralistes

- *-pg* pour indiquer au compilateur qu'il doit générer du code supplémentaire pour pouvoir instrumenter le code avec gprof

- *-O0* pour indiquer au compilateur qu'il doit désactiver les optimisations de compilation

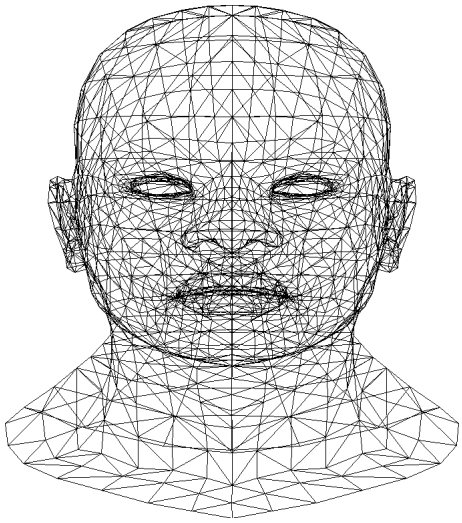

Figure 2: L'un des modèles ("head.obj") avec un affichage en "fil de fer" (wireframe)

A l'exécution du programme, un fichier "gmon.out" est créé. C'est ce fichier qui est utilisé par l'outil gprof comme entrée pour analyser les résultats de l'instrumentation du programme.

1. Effectuez les modifications du fichier "makefile" pour permettre l'instrumentation de votre programme et exécutez le
2. Exécutez la commande *gprof* pour analyser les résultats de l'instrumentation de votre programme
3. Comparez les temps d'exécution de votre approche de tracé de ligne vs. celle de Bresenham avec gprof. A partir de maintenant, il vous est demandé de profiler régulièrement votre code pour déterminer les points bloquants de votre moteur d'affichage
4. Faites l'affichage complet en "fil de fer" (comme montré en Figure 2) de l'un des modèles 3D proposés

5.3 Le triangle

Troisième et dernière étape (et pas des moindres) avant de pouvoir afficher un objet complètement : le dessin de triangles avec remplissage. Le faire correctement est une tâche qui n'est pas triviale. Pour cela, il faut :

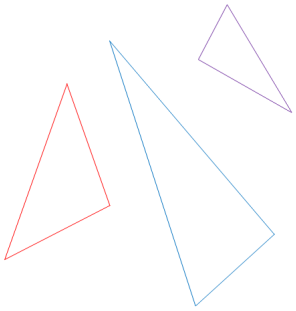

Figure 3: Le dessin de trois triangles sans remplissage

- Que la méthode soit simple et rapide (ce n'est pas une surprise)
- Que la méthode soit symétrique, c'est à dire que l'image ne doit pas dépendre de l'ordre des sommets passés en argument de la fonction de dessin
- Que si deux triangles ont deux sommets en commun, il n'y ai pas de trou entre eux lié à des arrondis de calculs

La méthode classique pour le faire est appelée le balayage de ligne (line sweeping). Elle consiste :

1. A trier les sommets selon leurs coordonnées en y
2. A balayer simultanément les côtés gauche et droit du triangle
3. A dessiner un segment de droite horizontal entre les points limites de gauche à droite

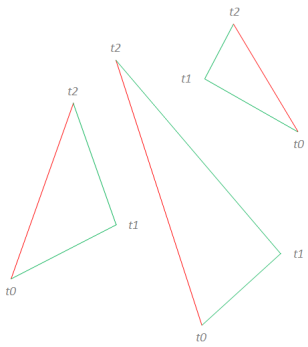

Figure 4: Le dessin des trois triangles avec leurs limites (représentation en rouge pour la limite A et en vert pour la limite B)

Admettons que les points d'un triangle soient $t0$, $t1$ et $t2$ et qu'il sont triés dans l'ordre ascendant selon leur coordonnées en y. Alors, la limite A est le segment entre les points $t0$ et $t2$, et la limite B est composée du segment entre les points $t0$ et $t1$ et de celui entre $t1$ et $t2$ (Figure 4). La limite B est constituée de deux segments. Il va donc falloir traiter le remplissage en deux parties, en coupant le triangle horizontalement comme montrée en Figure 5.

1. A l'aide des explications précédentes, complétez l'implémentation de la fonction **WindowDrawTriangle** du fichier "window.c" qui permet le tracé d'un triangle rempli en couleur et défini par trois points $t0$, $t1$ et $t2$.

2. Vérifiez le bon fonctionnement de votre implémentation en affichant l'un des objets 3D proposés. Utilisez la fonction rand() de la bibliothèque "stdlib.h" pour choisir une couleur aléatoire pour chacun des triangles (Figure 6)

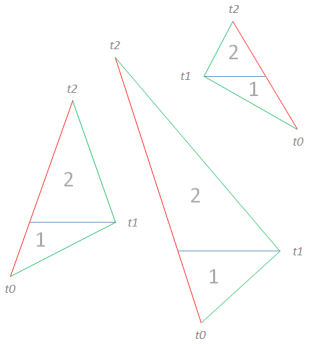

Figure 5: Dessiner et remplir un triangle revient à traiter le triangle en deux parties. Une partie basse (1) et une partie haute (2)

6 Un peu de lumière

Dans le but d'obtenir un rendu plus réaliste et donner de la profondeur à nos objets 3D, nous allons ajouter une source de lumière comme montré en Figure 7. Commençons par un rappel : à une même intensité, un triangle reçois plus de lumière lorsqu'il est orienté orthogonalement au vecteur représentant la direction de la source lumineuse. Il n'y aura pas d'illumination du triangle si celui-ci est parallèle à ce même vecteur. Pour paraphraser, l'intensité de la lumière est égale au produit scalaire entre le vecteur direction de la lumière et la normale du triangle. La normale du triangle peut être calculée simplement en effectuant le calcul du produit vectoriel de ses deux côtés. Notons que le produit scalaire peut être négatif. Dans ce cas, cela signifie que la lumière provient de derrière

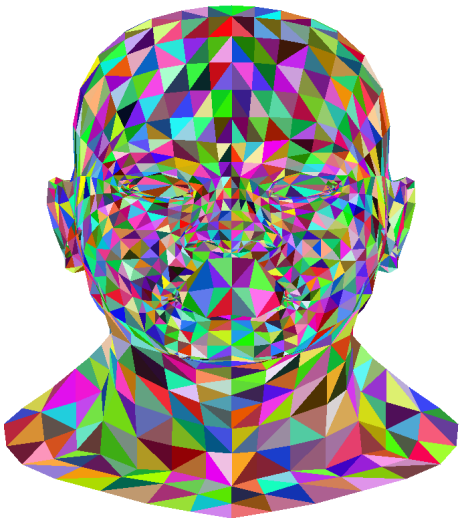

Figure 6: Le modèle “head.obj” affiché avec des triangles remplis et ayant des couleurs choisies aléatoire

le triangle. Si l'objet est bien modélisé (nous admettons que c'est le cas), nous pouvons simplement écarter ce triangle de l'affichage. Cette technique nous permet de supprimer rapidement les triangles non visibles par la caméra et s'appelle le back-face culling.

1. Complétez votre implémentation pour ajouter une source de lumière
2. Vérifiez le bon fonctionnement de votre implémentation en altérant la direction du vecteur lumière et en vérifiant la cohérence de l'affichage.

N.B: Vous remarquerez probablement des faces dessinées en avant d'autres alors que cela ne devrait par être le cas (Figure 7). C'est parfaitement normal car notre technique de suppression des triangles non visibles ne fonctionne que pour les formes parfaitement convexes ce qui est rarement le cas dans la réalité. Pour éviter ce problème, nous devons coder l'information de profondeur (z-buffer).

7 Suppression des faces cachées

En théorie, nous pourrions afficher tous les triangles sans en supprimer un seul, si nous commencions le rendu de l'arrière vers l'avant. Les faces les plus en avant effacerais celles qui se trouvent les plus en arrière. C'est ce que l'on appelle l'algorithme du peintre. Malheureusement, le coût calculatoire serait trop important (d'autant plus si le scène est en mouvement) et il serait même impossible de déterminer l'ordre d'affichage des faces dans certains cas.

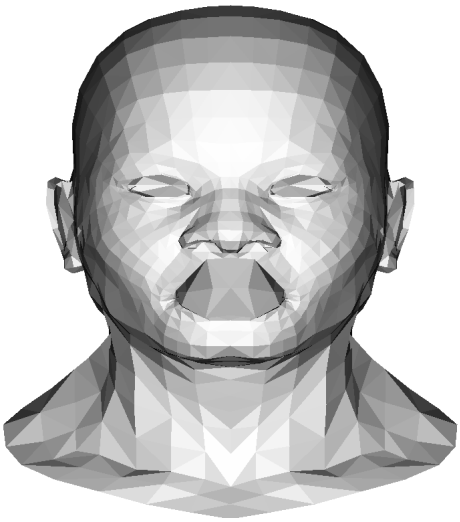

Figure 7: Le modèle “head.obj” affiché avec une source de lumière. Notons que la cavité de la bouche est, pour le moment, dessinée devant les lèvres

7.1 Affichage d’une scène simple

Imaginons une scène composée de trois triangles (Figure 8) avec une camera orientée pour la regarder de haut en bas (Figure 9). La face bleue est-elle derrière ou devant la face rouge ? Il est possible de décomposer en deux la face (une derrière la face rouge et l’autre devant), puis de décomposer de nouveau en deux la face devant la face rouge (une devant la face verte et l’autre derrière), etc... Le problème est que si la scène est composées de millions de faces, cela devient trop coûteux en temps de calcul. Il faut donc trouver une autre méthode.

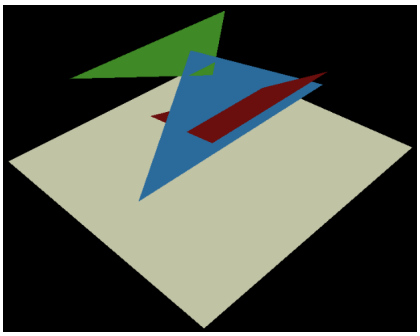

Figure 8: Une scène simple pour comprendre comment traiter la profondeur

7.2 Le y-buffer

Reprenons la même scène que précédemment mais en simplifiant les choses en la coupant par un plan (Figure 10). Si nous regardions la scène de côté, nous verrions la Figure 11. Nous allons déclarer et utiliser un tableau (y-buffer) de dimension ($width$, 1). L'ensemble des éléments de ce tableau sera initialisé avec la valeur moins l'infini. Notre scène est maintenant composée de trois segments de droite (intersection du plan jaune avec chaque triangle) et notre affichage final peut être effectué avec un buffer d'affichage de hauteur 1 comme montré en Figure 12.

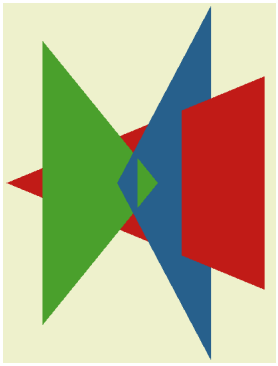

Figure 9: La scène d'exemple vue de dessus

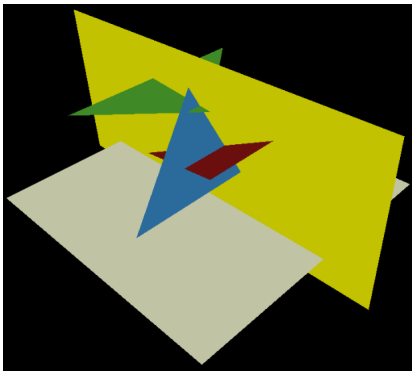

Figure 10: La scène d'exemple coupée par un plan

Pour constituer le y-buffer, il faut itérer sur l'ensemble des coordonnées en x entre les deux limites des triangles et calculer les coordonnées en y du segment. Il faut ensuite vérifier le contenu du buffer à l'indice courant de x. Si la valeur en y est plus proche de la camera que la valeur présente dans le buffer (c'est à dire une valeur numérique plus grande) alors nous dessinons le pixel à l'écran et nous mettons à jour le contenu du y-buffer. La Figure 13 montre le buffer

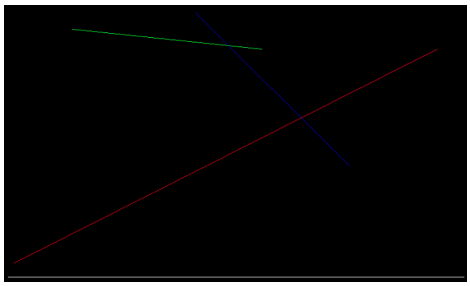

Figure 11: La scène d'exemple vu de côté

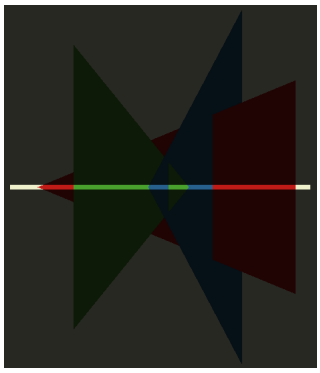

Figure 12: Le buffer d'affichage

d'affichage en haut et le y-buffer en bas pour chacune des trois étapes du rendu de la scène d'exemple. Pour le y-buffer, la couleur magenta représente la valeur moins l'infini. Le reste est représenté en niveaux de gris : gris clair lorsque le pixel est proche de la caméra et gris foncé lorsqu'il est plus éloigné.

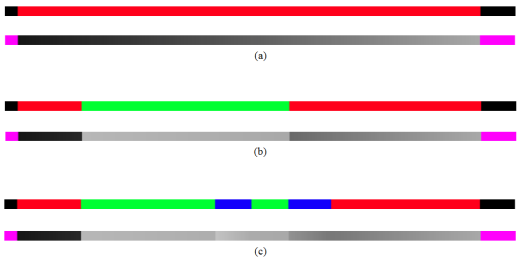

Figure 13: Contenu des buffers pour (a) : le dessin du segment rouge, (b) : le dessin du segment vert et (c) : le dessin du segment bleu

7.3 Retour à la 3D

Pour coder correctement l'information de profondeur et dessiner en 2D, le z-buffer doit avoir deux dimensions (*width*, *height*). Pour faciliter sa manipulation, il est préférable d'utiliser un buffer simple dimension.

1. Complétez votre implémentation pour ajouter le codage de l'information de profondeur par l'intermédiaire du z-buffer

2. Vérifiez le bon fonctionnement de votre implémentation en testant l'ensemble des objets 3D proposés. Pour référence, le modèle "head.obj" correctement affiché est donné en Figure 14

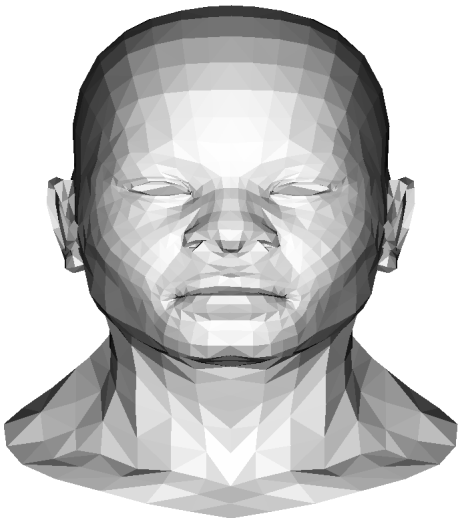

Figure 14: Le modèle "head.obj" affiché avec une source de lumière et avec gestion de la profondeur par le Z-buffer

8 Ajout des textures

Dans le fichier ".obj", les ligne"vt u v" permettent de constituer un vecteur de coordonnées de texture. Le nombre au milieu (entre les slashes) contenu dans les lignes des faces "f v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3" sont les indices des coordonnées de texture du sommet du triangle correspondant. Pour connaître la couleur du pixel à afficher, il faut interpoler ces coordonnées dans le triangle, et multiplier par la longueur et la hauteur de l'image utilisée comme texture. Le résultat d'une bonne implémentation est montré en Figure 15.

1. Complétez votre implémentation pour ajouter la texture de votre objet 3D. Pour vous faciliter le travail, utilisez la fonction :

stbi_load(filename, &imgwidth, &imgheight, &comp, STBI_rgb_alpha)

qui permet de charger une image à partir d'un fichier ".tga" et de la manipuler comme un tableau de pixels

2. Vérifiez le bon fonctionnement de votre implémentation

Figure 15: Le modèle "head.obj" affiché avec une source de lumière, gestion de la profondeur par le Z-buffer et texture

9 Optimisation

Dernier point de votre projet: l'optimisation.

1. Mettez en place une méthode simple du calcul de FPS (frames per second) qui permet de comptabiliser le nombre d'images affichées en une seconde par votre programme. On pourra s'aider de la fonction **gettimeofday** en incluant le fichier d'en tête "sys/time.h"

2. Utilisez la fonction **WindowSetTitle** pour mettre à jour le titre de le fenêtre avec le nombre de FPS effectifs

3. Consultez la documentation de GCC pour trouver les flags d'optimisations utiles et ainsi améliorer les performances de votre moteur

4. Dressez un tableau comparatif des résultats sans et avec les optimisations sélectionnées

5. Avec l'aide des documents [1] et [2], proposez une réécriture de votre code qui utilise les intrinsèques SIMD (Single Instruction Multiple Data). Attention, selon la façon dont vous avez écrit votre programme et les options de compilation que vous avez choisies précédemment, il se peut que votre compilateur fasse mieux que vous !

[1] `http://www.cs.uu.nl/docs/vakken/magr/2017-2018/files/SIMD%20Tutorial.pdf`
[2] `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`