Rapport de soutenance 2

RAGE

6 mai 2021

Groupe: MJRD

Membres: Jules Cassegrain (Chef de projet)
Damien Didier
Reinier Bullain Escobar
Mike Li

Table des matières

Le projet

1.1 Le projet

RAGE (Rustic Ascii Graphic Engine) est un moteur de rendu en trois dimensions qui, donné un objet 3D, l'affiche en caractères ASCII ; et permet de manipuler la caméra de manière à pouvoir faire pivoter et zoomer l'objet, ainsi que de manipuler l'éclairage. Cela implique nombreux calculs algébriques sur les matrices et l'optimisation de la mémoire pour avoir une expérience agréable.

Le rendu ASCII nous a semblé intéressant car il nous permettrait de visualiser ces objets dans la même console ou dans une nouvelle console sans avoir besoin d'utiliser des bibliothèques graphiques. D'un point de vue mathématique, il propose également un défi consistant à choisir les bons caractères à afficher, car au lieu de montrer des pixels, nous afficherions de petites régions remplacées par des caractères ; ceux-ci pouvant varier en fonction de l'éclairage ou de la taille de la région à afficher.

Planning

2.1 Répartition des tâches

Pour un premier cahier de charges, les tâches que nous avons identifiées ont été réparties comme suit. Ces tâches, ainsi que leur répartition, sont susceptibles de changer.

1. Parseur : Transformer l'entrée fourni, pour le moment un fichier *Wavefront OBJ*, en une structure de données facile à manipuler lors du rendu.
2. API : Bibliothèque de fonctions d'algèbres linéaires utilisés pour le projet.
3. Rendu : Les fonctions et structures nécessaires pour afficher l'objet dans une image.
4. Application : Programme qui lancera l'affichage de l'objet sur la console sous la forme de caractères ASCII.
5. Site : Site web vitrine du projet à développer.

	Parseur	API	Rendu	Application	Site
Jules	assistant	responsable	assistant		
Damien		assistant	responsable	assistant	
Reinier	responsable		assistant	responsable	assistant
Mike			assistant	assistant	responsable

Table 2.1: Répartition des taches

2.2 Diagramme de Gantt

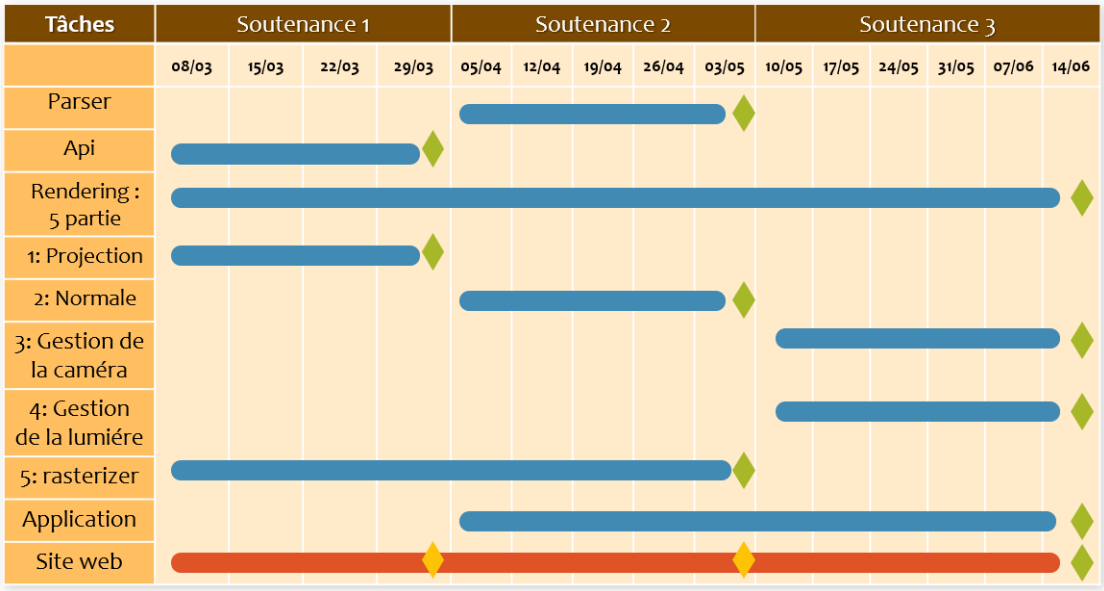

Figure 2.1: Diagramme de Gantt pour la répartition des tâches

Avancée du projet

Voici ce qui a été fait jusque là pour cette seconde soutenance.

3.1 Travail en équipe

Pour ce projet nous avons utilisé un dépôt git hébergé sur le GitLab du CRI. Sur celui-ci, nous avons crée plusieurs branches correspondants aux différents aspects du projet détaillés dans la répartition des tâches. Ce dépôt rassemble tout ce qui a été fait sur notre projet et nous permet de collaborer efficacement.

3.2 Parseur

L'idée du parseur est d'importer un fichier représentant une géométrie 3D dans une structure définie en C. Le format choisi pour stocker les objets que notre programme va afficher est **«Wavefront .obj»**.

OBJ est un format de fichier contenant la description d'une géométrie 3D ; ce format de fichier est *open source* et a été adopté par d'autres logiciels 3D pour des traitements d'importation / exportation de données.

3.2.1 Structure du fichier

Une surface polygonale est décrite par un ensemble de sommets (accompagné de coordonnées de texture et de normales en chaque sommet) et d'un ensemble de faces. Chaque élément est défini respectivement comme suit :

- Un sommet : *v 1.0 0.0 0.0*
- Une coordonnée de texture : *vt 1.0 0.0*
- Une normale : *vn 0.0 1.0 0.0*

- Une face : *f v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3*, d'où chaque élément fait référence aux coordonnées des points, de texture et des normales définies précédemment. Ces indices des sommets commencent par l'indice 1.
- Autres ...

Un commentaire peut être placé si la ligne commence par le caractère '#'.

3.2.2 La mise en œuvre

Le parseur implémenté utilise la fonction *fopen* pour lire le fichier donné et lit ligne par ligne en utilisant la fonction *getline*, identifiant et en traitant le type d'élément grâce aux fonctions de la bibliothèque **string.h** : *strtok_r*, *strcmp*, *strtof*, *strtoul*.

Pour l'instant il est possible d'interpréter un fichier .obj avec les éléments mentionnés et de sauvegarder ces données dans une structure appelée *obj_t* défini dans **obj.h** de la manière suivante :

```
typedef struct {
    vector3_t   **vertices;
    vector3_t   **texture_coords;
    vector3_t   **normals;
    face_t      **faces;

    size_t      v_count;
    size_t      vt_count;
    size_t      vn_count;
    size_t      face_count;
} obj_t;
```

Chaque erreur a été capturée et traitée comme une valeur entière spécifique en fonction du type d'erreur. Il ne devrait pas non plus y avoir de fuites de mémoire (vérifié avec **Valgrind**).

Plusieurs cas de test ont été mis en œuvre, en utilisant la bibliothèque **Unity**, dans le dossier *tests*, assez simple à utiliser.

3.3 API

3.3.1 Vertex et vecteur homogène

Un vertex est vecteur de position dans un espace en 3 dimension, et a donc 3 coordonnées x, y et z.

Un vecteur homogène est un vecteur à 4 coordonnées x, y , z et w, avec comme limitation que x = a.w, y = b.w et z = c.w : un vertex est donc un vecteur homogène dont la quatrième composante est 1 et en divisant chaque composante d'un vecteur homogène par w on obtient un vertex (normaliser le vecteur homogène)

3.3.2 Matrice de transformation

Une matrice de transformation est une matrice 4x4 telle que :

$$\begin{pmatrix} Rot_{3\times3} & 0_{1\times3} \\ Trans_3 \times 1 & 1_{1\times1} \end{pmatrix}$$

avec *Rot* une matrice de rotation 3x3 et *Trans* un vecteur de translation.

3.3.3 Opérations d'algèbre linéaire

Nous avons codé notre propre implémentation des opérations de base d'algèbre linéaire:

- produit scalaire
- produit vectoriel
- produit matriciel

En plus nous avons rajouté une fonction pour multiplier un vertex par une matrice 4x4 en utilisant le vecteur homogène correspondant au vertex et en normalisant le vecteur homogène résultant.

3.3.4 Racine carrée inverse rapide

Afin de calculer les normes des vecteur nécessaires pour transformer nos vecteurs en vecteurs unitaires, nous avions besoin de calculer la norme de ces vecteurs. Pour cela nous avons besoin de calculer la racine carré. Mais les méthodes normales de calcul de racine carré sont assez lentes et peu optimisées. Nous avons donc fait des recherches et nous avons découvert la méthode *FastInvSqrt* qui est une méthode pour calculer l'inverse de la racine carré d'un nombre. Cette méthode assez particulière a été inventé en 1999 durant le développement de *Quake III Arena*. Elle consiste à prendre en mémoire la moitié du nombre passé en argument, puis de traiter le

nombre comme un entier et de lui appliquer un *bitshift*. Le résultat est ensuite soustrait à une valeur dite "magique" afin d'obtenir une première approximation de l'inverse de la racine carré du nombre. On reconsidère maintenant ce nombre comme un nombre flottant et en appliquant la méthode de Newton, on obtient une meilleur approximation de cette racine carré. Cette méthode permet d'obtenir une approximation de l'inverse de la racine carré relativement précise et cela 4 fois plus vite que via les autres méthodes. Ici le code original:

```
float Q_rsqrt(float number)
{
long i;
float x2, y;
const float threehalfs = 1.5F;

x2 = number * 0.5F;
y = number;
i = * ( long * ) &y; // evil floating point bit level hacking
i = 0x5f3759df - ( i >> 1 ); // what the fuck?
y = * ( float * ) &i;
y = y * ( threehalfs - ( x2 * y * y ) ); // 1st iteration
// y = y * ( threehalfs - ( x2 * y * y ) ); // 2nd iteration, this can
be removed
    return y;
}
```

3.4 Rendu

3.4.1 Utilisation du parseur

On utilise le parseur pour importer un modèle.

3.4.2 Projection du repère de la caméra au repère homogène de clipping

Pour rendre une scène en trois dimensions dans une image en deux dimensions, on utilise une transformation appelée projection. La projection en perspective est la

plus utilisé parce qu'elle permet de rendre compte du phénomène de point de fuite : les objets lointains paraissent plus petits que les objets proches. Pour cela, il faut d'abord définir l'espace vu par la caméra, une pyramide tronqué ou frustum est le plus souvent utilisé. Enfin, on transforme les coordonnées du repère de la caméra dans un espace appelé espace homogène de clipping, qui est juste une déformation du frustum en un pavé droit ou un cube et dont les coordonnées en x et y sont entre -1 et 1. Les coordonnées en z sont soit entre -1 et 1 aussi (standard OpenGL) ou entre 0 et 1 (standard DirectX). Cet espace permet d'avoir les coordonnées des points indépendamment du champs de vision de la caméra et de sa résolution. En pratique, il a fallu déterminer la matrice de projection et multiplier chaque vertex de chaque face de l'objet par cette matrice ce qui nous donne un vecteur homogène dont la quatrième composante est l'ancien z. A partir de ses coordonnées, il est enfin aisé de retrouver la position du vertex a l'écran et de passer à la rastérisation.

$$\begin{pmatrix} aspectRatio \times fov & 0 & 0 & 0 \\ 0 & fov & 0 & 0 \\ 0 & 0 & \frac{z_Far}{z_Far - z_Near} & 1 \\ 0 & 0 & \frac{-z_Far \times z_Near}{z_Far - z_Near} & 0 \end{pmatrix}$$

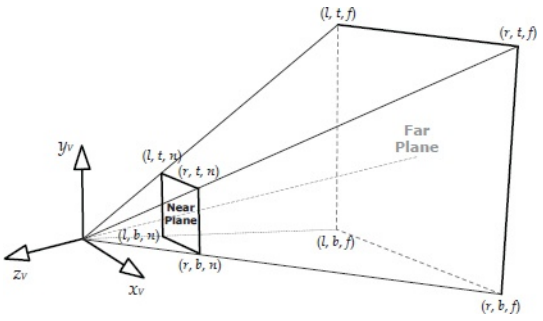

Figure 3.1: Projection en perspective dans un frustrum, *Game engine architecture*, Jason Gregory

3.4.3 Rasterization

"La rastérisation est un procédé qui consiste à convertir une image vectorielle en une image matricielle destinée à être affichée sur un écran ou imprimée par un matériel d'impression.". Dans notre cas, cela consiste à prendre un triangle qui est définie en 3D et l'afficher sur notre écrans en 2D. Ce procédé se déroule en 2 étapes:

3.4.3.1 Étape 1

La première étape consiste à déterminer les pixels sur lesquels notre triangle va être projeté. Pour cela on va tester pour chaque cases de la matrice si elle appartient au triangle: si oui alors on la marque sinon on passe à la suivante. Pour optimiser ce procéder on commence par obtenir les coordonnées maximum et minimum sur lequel le triangle pourra être projeté puis on parcoure la matrice entre ces coordonnées. Cela permet d'éviter de parcourir l'entièreté de la matrice pour afficher un seul objet et donc de rendre le procédé beaucoup plus rapide.

3.4.3.2 Étape 2

La deuxième étape consiste à calculer la profondeur de notre triangle afin d'éviter d'afficher par exemple un triangle qui se trouverait derrière un autre : on ne doit afficher que le triangle le plus proche de nous et pas celui derrière. Si cette étape n'était pas présente, en affichant un cube en 3D par exemple, les faces du cube qui ne sont pas visibles de notre point de vue seraient visibles et donc l'image en 2D n'aurait rien à voir avec la réalité.

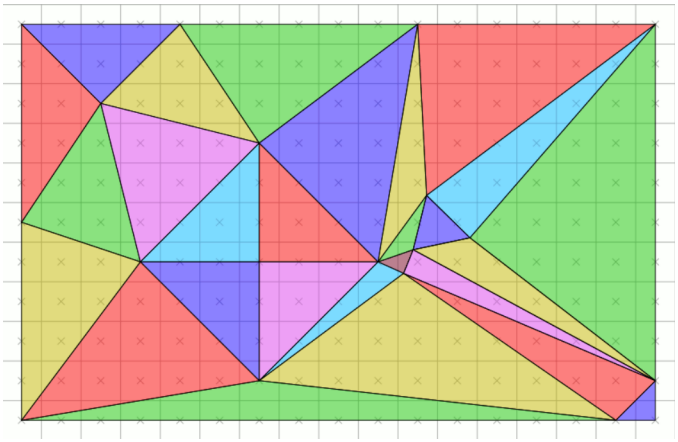

Figure 3.2: Avant rasterisation

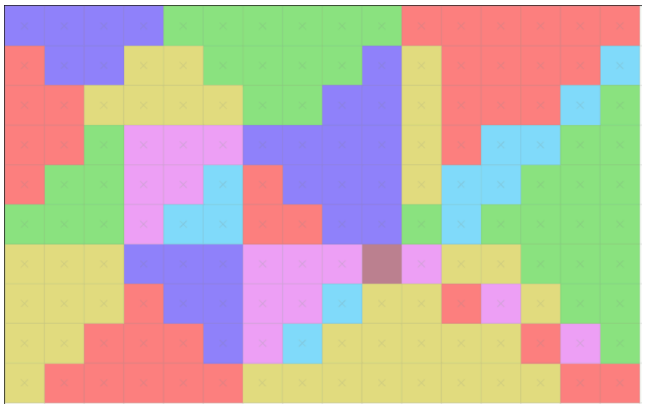

Figure 3.3: Après rasterisation

3.4.4 Utilisation des normales

3.4.4.1 Calcul des normales

On peut déterminer une normale de chaque face en faisant le produit vectoriel de deux des cotés du triangle. Cependant, celle ci n'est pas nécessairement normé ce qui peut être embêtant, d'autant plus que normaliser un vecteur peut être coûteux: en effet, le calcul de la racine carré et de l'inversion de celle ci sont des opérations complexes. Heureusement, nous pouvons utiliser la méthode *FastInvSqrt* vu plus tôt.

3.4.4.2 Backface culling

La première utilisation des normales est pour déterminer quelles faces seront potentiellement vu par la caméra et d'éliminer des faces dont les normales ne pointent pas vers la caméra, ce qu'on appelle le *backface culling*. Cette étape est importante car la projection est plutôt complexe, mettant en jeu des produits de matrices. Il est donc impératif d'éviter de projeter des triangles qui ne seront de toute manière pas visibles. Ce calcul est relativement simple puisse qu'il suffit de faire un produit scalaire entre la normale et un vecteur qui part de la caméra et va jusqu'à l'un des sommet du triangle. en fonction du signe du résultat, soit on continue la projection soit on passe au triangle suivant.

3.4.4.3 Éclairage

Nous pouvons aussi utiliser les normales pour éclairer les faces. Une première méthode, mise en place pour cette soutenance, est de calculer le produit scalaire entre

la normale de la face et un vecteur qui correspond à une lumière directionelle. Cette méthode est approximative mais permet de faire une différence entre les différentes faces d'un objet.

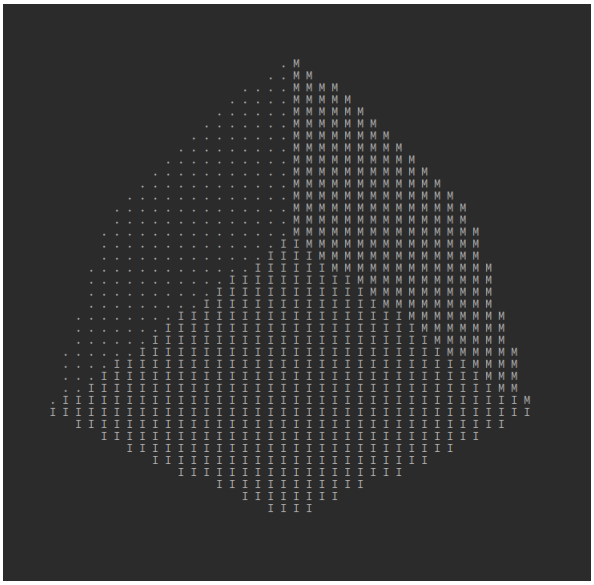

Figure 3.4: Cube avec éclairage

3.5 Application

3.5.1 Traitement des arguments

La première étape de l'application consiste à obtenir les arguments ; le fichier .obj et éventuellement quelques paramètres facultatifs. Pour cela, aucune bibliothèque n'est nécessaire, mais pour faciliter le traitement de chaque argument la bibliothèque **argp** est utilisée. Cette bibliothèque permet, à partir d'un tableau d'éléments, d'ajouter des arguments à l'application qui sont traités dans une fonction spécifiée, et génère une documentation de type *man*.

3.5.2 La fenêtre

Après que le fichier contenant la géométrie 3D soit traité, elle doit être affiché dans le terminal, où elle peut être manipulé à l'aide du clavier. Pour cela, après avoir fait quelques recherches, nous allons utiliser la bibliothèque **ncurses**. Cette bibliothèque propose une API pour le développement d'interfaces utilisateur (GUI), en utilisant les caractères et couleurs d'un mode semi-graphique. Non seulement ce

type d'interface utilisateur se conçoit de manière indépendante du terminal, mais il accélère le rafraîchissement d'écran, diminuant par là le temps de latence que subissent d'ordinaire les utilisateurs de shells à distance.

Pour l'instant, il n'a été programmé que jusqu'au point de créer une nouvelle fenêtre et de reconnaître les touches pressées, soit la touche «Q» pour quitter la fenêtre.

3.6 Site Web

Le site est accessible à l'adresse suivante :
https://damien.didier.pages.epita.fr/rage-rustic-ascii-graphic-engine/

3.6.1 Choix des technologies

La première étape a été de choisir les technologies à utiliser, cette étape a pris largement plus de temps que ce que j'avais prévu. Dès le départ je savais que je voulais utiliser **GitLab Pages** pour le déploiement et l'hébergement ce qui nous permet de réunir tous les aspects du projet dans un unique endroit.

Ensuite je me suis intéressé aux très (trop ?) nombreuses solutions pour la création d'un site statique. Pour cette étape je ne voulais pas utiliser un simple template en HTML, CSS et Bootstrap comme lors du projet de S2. Cependant je suis beaucoup plus familier avec les technologies utilisées pour créer des sites dynamiques, je me suis donc pas mal perdu en explorant toutes les solutions disponibles.

Au final j'ai décidé d'utiliser le très populaire **Jekyll** surtout pour sa faciliter d'utilisation tout en me fournissant toutes les fonctionnalités qui me sont nécessaires. Et enfin puisque le but était de créer un site plutôt minimal, j'ai choisi le thème minimal pour GitHub Pages, cependant ce choix posera quelques problèmes plus tard quand j'essaierai de le déployer en utilisant GitLab Pages.
J'ai du faire ce choix car GitHub Pages est une solution beaucoup plus populaire pour ce type de projet ce qui fait que le choix de thèmes pour GitLab Pages est très limité et aucun ne me convenait pour atteindre le rendu que je désirais.

3.6.2 Création du site

Le thème que j'ai choisi étant optimisé pour GitHub, j'ai téléchargé le dépôt GitHub du thème minimal pour y apporter toutes les modifications nécessaires au

déploiement sur GitLab Pages tout en me familiarisant avec Jekyll. Le framework était à première vue assez différent de ce que j'ai pu utiliser jusque là, cependant passé cette première étape d'adaptation je me suis rendu compte de sa facilité d'utilisation et j'ai pu retrouver des concepts familier comme les "includes" qui servent de composants ou encore l'interpolation de variables.

3.6.3 Mise en production

Pour cette partie j'ai utilisé l'outil de **CI/CD** de GitLab pour déployer le site sur GitLab Pages. Pour cette étape j'ai aussi rencontré quelques problèmes. Tout d'abord je n'avais pas la possibilité d'utiliser les *shared runners* de GitLab pour le déploiement car le GitLab du CRI n'en possède pas. J'ai donc configuré un **Docker** sur ma machine et l'ai enregistré en tant que *runner* GitLab pour le projet afin de pouvoir générer et déployer le site en continue. Ensuite j'ai modifié le fichier YAML du projet afin de pouvoir déployer le site en production directement depuis la branche web de notre projet et non depuis la branche master.

Enfin, quelques problèmes de dépendances plus tard, la pipeline était configuré et prête à l'emploi. Bien que ces problèmes n'était au final pas bien difficiles à résoudre, le débogage était plutôt long puisqu'après chaque modifications je devais attendre plusieurs minutes avant de voir si le *runner* parvenait à générer le site ou non.

3.6.4 Résultat

Au final le résultat est très concluant, nous avons un site sobre et opérationnel et surtout extrêmement simple à maintenir. En effet maintenant que toute la structure est faite et que la pipeline est configurée, ajouter du contenu au site se fait simplement en modifiant un fichier **Markdown**, ce qui est faisable directement depuis le site GitLab. Et après chaque nouveaux commits la nouvelle version du site est automatiquement mis en production.

3.7 Conversion en ASCII

Dans cette partie nous réalisons un programme qui nous permet de convertir une image 2D en ASCII Art, celui-ci nous sera utile plus tard afin d'obtenir le résultat final une fois que le rendu 3D sera opérationnel.

3.7.1 Manipulation de l'image

Pour importer et manipuler des images nous avons d'abord pensé à utiliser la bibliothèque **SDL** que nous avions utilisé lors d'un TP de S3, mais finalement nous avons décidé d'utiliser la bibliothèque **stb_image** qui est l'option la plus simple et la plus répandu pour manipuler des images en C. Cette bibliothèque a aussi l'avantage d'importer les images sous formes de tableau d'entiers, ce qui correspond aussi au format que nous utilisons lors du rendu graphique. Ainsi nous limitons le nombre d'ajustements qu'il faudra réaliser lors de l'intégration au reste du projet.

3.7.2 Redimensionnement de l'image

La bibliothèque **stb_image** contient des fonctions qui nous permettent de redimensionner facilement une image. Pour cela nous utilisons la fonction **stbir_resize_uint8** de la bibliothèque **stb_image_resize**, afin d'obtenir une image ne dépassant pas les limites de la fenêtre de la console. De plus nous prenons soin de ne pas déformer l'image en conservant son ratio, pour cela nous prenons une largeur fixe en fonction de la taille de la fenêtre et nous la multiplions par le ratio pour obtenir les dimensions finales de l'image. Nous avons choisi de fixer la largeur plutôt que la hauteur car c'est souvent la dimension la plus problématique dans notre cas.

3.7.3 Conversion en nuances de gris

La première étape est de convertir l'image en nuance de gris. Nous partons d'une image en couleur composée de pixels de 3 canaux de couleurs et 1 canal de transparence optionnel en fonction du format de l'image. Pour réaliser la conversion en nuances de gris nous parcourons l'image et pour chaque pixels nous réalisons une moyenne des valeurs des canaux de couleurs, nous ajouterons des poids aux différents canaux afin d'obtenir un résultat plus satisfaisant. Ainsi nous obtenons une nouvelle image en nuances de gris d'un canal ou deux s'il existait un canal de transparence dans l'image de départ.

3.7.4 Conversion en ASCII

Ensuite, pour convertir l'image en ASCII Art nous découpons l'image en groupes de pixels rectangulaires, la taille du rectangle sera donnée par l'utilisateur dans

l'application mais nous utiliserons des groupes de 5 x 10 pixels pour notre démonstration. Pour chaque rectangles nous réalisons une moyenne des valeurs qui sera comprise entre 0 et 255, nous divisons cette valeur par 255 afin d'obtenir une valeur comprise entre 0 et 1 et nous y associons le caractère correspondant de la manière suivante :

$0.0 <= x <= 0.1 == '@'$

$0.1 < x <= 0.2 == '\%'$

$0.2 < x <= 0.3 == '\#'$

$0.3 < x <= 0.4 == '*'$

$0.4 < x <= 0.5 == '+'$

$0.5 < x <= 0.6 == '='$

$0.6 < x <= 0.7 == '-'$

$0.7 < x <= 0.8 == ':'$

$0.8 < x <= 0.9 == '.'$

$0.9 < x <= 1.0 == ' '$

3.7.5 Gestion des bords

Lorsque le rectangle chevauche les bords de l'image, nous faisons la moyenne sur un rectangle plus petit et nous leurs associons un caractère de la même manière. Toutefois le caractère associé est toujours de même taille, l'image finale peut donc être légèrement plus grande que l'image de départ.

Figure 3.5: Logo en ASCII Art

La suite

4.1 Conversion en ASCII

Pour ce qui est de cette partie nous avons plusieurs pistes d'améliorations possibles. Pour commencer nous pouvons obtenir un rendu plus proche en utilisant plus de caractères différent. Ensuite nous pouvons encore améliorer la précision du rendu en prenant en compte la forme contenu dans le rectangle pour y associer le caractère le plus similaire, pour cela il faudra calculer la distance de nos rectangles aux matrices représentant les caractères.

Pour le moment notre programme affiche l'image finale en réalisant des *printf* dans la console, cette solution est tout à fait satisfaisante pour une image statique cependant elle ne sera pas très efficace lorsqu'il s'agira de bouger la caméra autour de l'objet. Pour cela nous pouvons utiliser la bibliothèque **ncurses** afin d'avoir plus de contrôle sur la fenêtre de la console.

4.2 Le rendu graphique

Pour la prochaine soutenance, il ne nous reste plus qu'à mettre en place la gestion de la caméra, ainsi que la gestion de la lumière. Nous avons d'ores et déjà commencé à mettre en place une gestion de la caméra en intégrant dans la gestion du backface culling un lien avec un vecteur caméra afin de pouvoir modifier l'angle de vue de la caméra. Nous avons atteint nos objectifs et sommes totalement prêt pour commencer le travail jusqu'à la prochaine soutenance.

4.3 Nettoyer le code

Pour la soutenance finale, il nous faudra "nettoyer" le code de tous les commentaires, codes inutilisés et autre restes afin de le rendre présentable. Nous devrons donc nous atteler à cette tâche en vue de la dernière soutenance.

Conclusion

Le projet est toujours sur de bons rails, nous avons réussi à tenir les délais fixés dans le planning de départ et nous sommes confiants de réussir à mener ce projet à terme.