



RAPPORT DE SOUTENANCE 1

RAGE

9 avril 2021

Groupe: MJRD

Membres: Jules CASSEGRAIN (Chef de projet)

Damien DIDIER

Reinier BULLAIN ESCOBAR

Mike LI

Table des matières

1	Le projet	3
1.1	Le projet	3
2	Planning	4
2.1	Répartition des tâches	4
2.2	Diagramme de Gantt	5
3	Avancée du projet	6
3.1	Travail en équipe	6
3.2	Parseur	6
3.2.1	Structure du fichier	6
3.2.2	La mise en œuvre	7
3.3	API	7
3.3.1	Vertex et vecteur homogène	7
3.3.2	Matrice de transformation	8
3.3.3	Opérations d'algèbre linéaire	8
3.4	Rendu	8
3.4.1	Utilisation du parseur	8
3.4.2	Projection du repère de la caméra au repère homogène de clipping	9
3.4.3	Rasterization	9
3.5	Application	11
3.5.1	Traitement des arguments	11
3.5.2	La fenêtre	11
3.6	Site Web	11
3.6.1	Choix des technologies	11
3.6.2	Création du site	12
3.6.3	Mise en production	12
3.6.4	Résultat	13

4	La suite	14
4.1	L'application	14
4.2	Le site web	14
4.3	Le rendu graphique	14
5	Conclusion	15

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

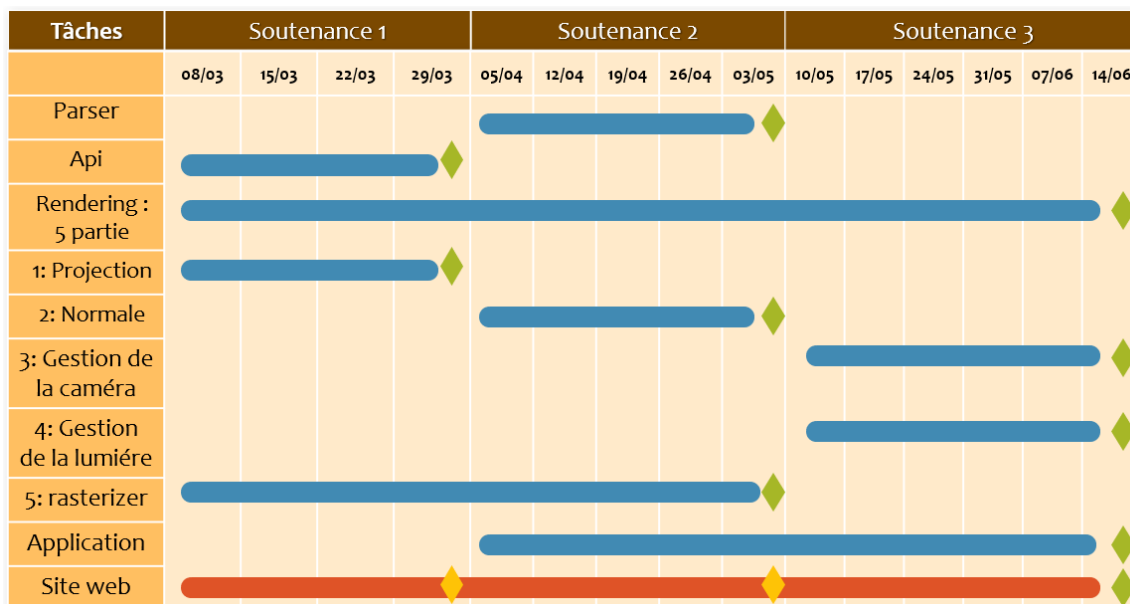


Figure 2.1: Diagramme de Gantt pour la répartition des tâches

Avancée du projet

Voici ce qui a été fait lors de cette première phase de développement.

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 créé 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 :

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

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.

3.4 Rendu

3.4.1 Utilisation du parseur

Une fois que Reinier a fini la partie parseur, j'ai utilisé son algorithme pour importer un cube.

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 utilisée 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 tronquée 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 champ 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 à l'écran et de passer à la rasterisation.

$$\begin{pmatrix} aspectRatio \times fov & 0 & 0 & 0 \\ 0 & fov & 0 & 0 \\ 0 & 0 & \frac{z_{Far}}{z_{Far} - z_{Near}} & 1 \\ 0 & 0 & \frac{-z_{Far} \times z_{Near}}{z_{Far} - z_{Near}} & 0 \end{pmatrix}$$

3.4.3 Rasterization

"La rasterisation 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 que 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 procédé on commence par obtenir les coordonnées maximum et minimum sur lequel le triangle pourra être projeté puis on parcourt 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 face du cube qui ne sont pas visible de notre point de vue serait visible et donc l'image en 2D n'aurait rien a voir avec la réalité.

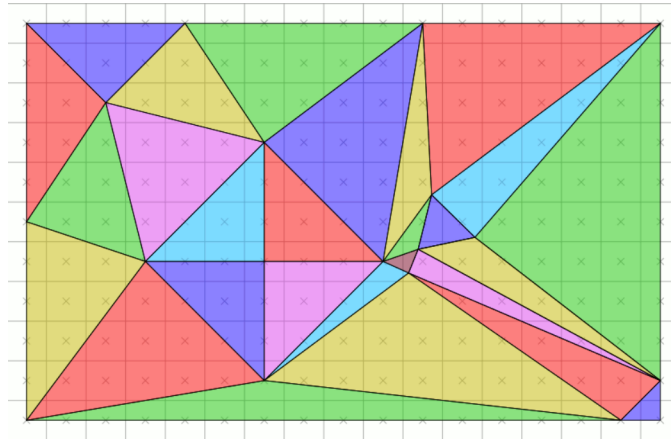


Figure 3.1: avant rasterisation

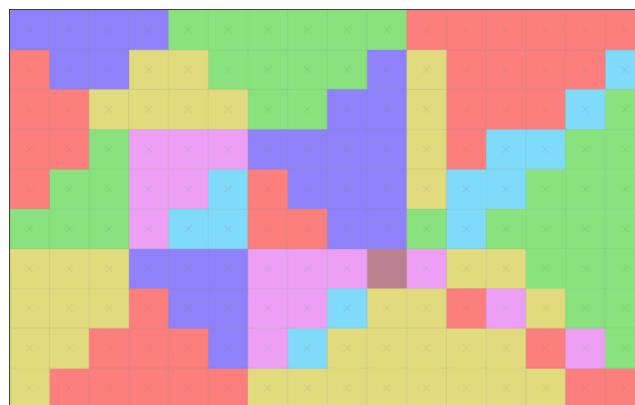


Figure 3.2: après rasterisation

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 affichée dans le terminal, où elle peut être manipulée 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 facilité 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.

La suite

4.1 L'application

Afin d'avancer le statut de l'application, il est nécessaire que toute la partie rendu 3D soit terminée.

4.2 Le site web

Pour le site web il n'y a plus rien à faire à part continuer à mettre à jour le contenu tout au long du projet. Cependant une amélioration possible serait de pouvoir utiliser un serveur pour le déploiement du site, en effet actuellement le déploiement est réalisé par une machine personnelle qui ne tourne pas 24 heures sur 24. L'utilisation d'un serveur permettrait un réel déploiement en continu du site.

4.3 Le rendu graphique

Pour la prochaine soutenance nous devons mettre en place la gestion de la normal et finir la rasterisation qui est presque finis. Nous devons également pour la 3ème soutenance nous devons finir la gestion de la caméra et mettre en place la gestion de la lumière. nous avons atteint nos objectif fixé pour cette soutenance et avons même pris un peu d'avance sur la soutenance 2

Conclusion

Comme vous pouvez le voir, le projet avance bien. Nous sommes parvenu à suivre le planning prévu et même à prendre un peu d'avance. Nous sommes donc très confiant pour la suite de ce projet.

Maintenant que toutes les fondations sont posées, il est temps de s'attaquer à la partie principale du projet.