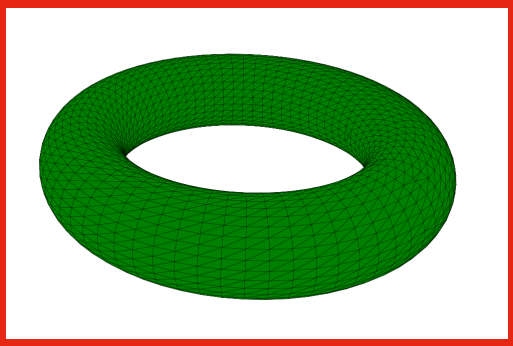
INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Projet scientifique

Moteur graphique 3D



INSA Toulouse
135, Avenue de Rangueil
31077 Toulouse Cedex 4 - France
www.insa-toulouse.fr

RODRIGUEZ Leandro
RÉBILLARD Baptiste

2MIC A - Promotion 59 - 2022/2023

Soutenance le 6 juin 2023

Projet scientifique

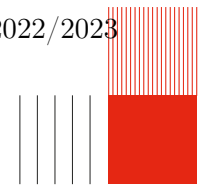
Moteur graphique 3D

INSA Toulouse
135, Avenue de Rangueil
31077 Toulouse Cedex 4 - France
www.insa-toulouse.fr

RODRIGUEZ Leandro
RÉBILLARD Baptiste

2MIC A - Promotion 59 - 2022/2023

Soutenance le 6 juin 2023



Remerciements

Avant de commencer ce rapport, nous tenions à remercier M. Sanchez, notre encadrant dans le cadre de ce projet.

Table des matières

Introduction

Un moteur graphique est un programme qui permet de convertir un ensemble de coordonnées tridimensionnel en un rendu bidimensionnel (c'est-à-dire d'un ensemble de points en relief affiché sur un écran par exemple).

Les moteur 3D ont divers domaines d'applications : l'affichage d'une simulation, d'une modélisation, les films d'animation, ou plus communément les jeux vidéo...

Dans les jeux vidéo, notamment, il existe deux principaux moteurs graphiques (unreal engine et Unity), qui sont assez complets et faciles à prendre en main. Ce qui permet aux développeurs d'avoir accès à un moteur graphique sans en comprendre l'aspect mathématique. Il est donc intéressant de recoder un moteur graphique depuis la base afin d'en comprendre tous les aspects.

Pour développer ce moteur graphique, nous avons dû nous informer : comprendre comment marche le moteur graphique et à notre niveau de recréer certaines fonctionnalités, comprendre l'aspect mathématique et enfin de le coder.

Nous sommes arrivés à un certains résultats malgré quelques difficultés que nous aborderons.

1 Début du projet : recherche et organisation

1.1 Choix du langage

Lorsque le projet nous a été présenté, on nous a conseillé de nous tourner vers le python, car facile à prendre en main et des librairies étaient déjà existantes pour faciliter le code. Nonobstant, connaissant déjà le langage python, nous voulions faire un choix différent, pour essayer de découvrir ce sujet d'un autre point de vue et d'apprendre un nouveau langage.

Dans un premier temps, nous nous sommes tournés vers le C++, mais on s'est vite rendu compte qu'il existait peu de librairies 2D. Les librairies existantes faisaient déjà office de librairies 3D, ce qui est assez regrettable.

Nous sommes donc partis sur le javascript (JS), un langage qui est assez courant, et que l'on peut intégrer à une page web.

1.2 Appréhensions et attentes

Les moteurs graphiques d'aujourd'hui étant assez complet, nous ne pouvions pas faire quelque chose d'aussi élaboré. C'est pourquoi nous avons dû faire des choix sur ce que nous souhaitions intégrer.

Notre objectif était d'avoir un moteur 3D qui tourne, avec des commandes simples pour se déplacer dans l'espace, et dans la possibilité d'importer un objet modélisé en 3D depuis un logiciel externe dans notre moteur graphique.

Ce projet était notre premier projet mêlant programmation et mathématiques, qui plus est, dans un nouveau langage. En sachant cela, nous avions peur de viser un peu trop haut et de ne pas être capable de fournir un travail abouti. Nous avions également peur d'être bloqué sur des erreurs de langage sans outils pour les corriger puisque notre moteur graphique est dans un langage différent de celui des autres groupes.

1.3 Connaissances à mobiliser

Codant dans un nouveau langage : le JavaScript, nous avons dû apprendre les bases afin d'être capable de le manipuler. Nous avons également dû nous familiariser avec l'aspect mathématique du fonctionnement du moteur graphique. Pour cela, nous nous sommes renseignés sur Internet et avons reçu l'aide de notre encadrant qui nous a également fourni un livre sur le sujet.

Enfin, par soucis d'organisation, il a fallu réfléchir à une manière de gérer le code, nous nous sommes donc tourné vers git afin d'avoir une trace de l'historique du code et pouvoir travailler sur plusieurs versions du code en parallèle.

2 Explication générique du moteur graphique (Aspect mathématique)

2.1 Processus global

Notre moteur graphique doit, à partir d'une caméra et de son inclinaison, afficher une liste de faces, chaque face étant caractérisée par les coordonnées de ses 3 sommets. Par assemblage de ces faces, il doit être capable d'afficher tout l'objet (plus il y a de faces, meilleure est la résolution).

Pour afficher cet objet, qui de base est modélisé en 3D, de manière fidèle sur un écran en 2D, il faut projeter une des coordonnées selon les deux autres. Ici, on voit qu'il faut faire une projection de la profondeur sur l'écran pour que même s'il est affiché plat, il y ait cette impression de relief. Cependant, il existe plusieurs types de projection, nous avons donc dû en choisir une, celle qui nous paraissait le plus fidèle : la projection en perspective.

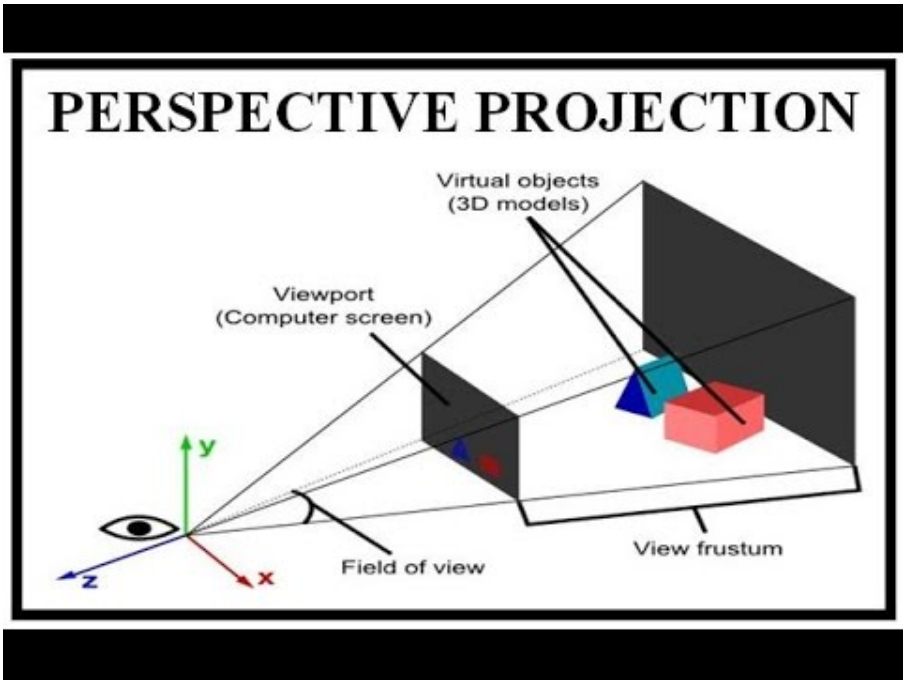


FIGURE 1 – Principe de la projection en perspective

Il existe d'autres types de moteurs graphiques, mais l'avantage de celui-là est qu'il est simple à comprendre et qu'on peut importer dedans des fichiers que nous avons modélisé en 3D avec un logiciel externe (car le format de ce genre de fichier est également une liste de faces). L'inconvénient est qu'il est difficile d'obtenir un rendu parfait avec peu de faces, ce qui multiplie les calculs et empêche d'avoir des formes "parfaitement" sphériques par exemple.

Nous allons donc commencer par aborder le fait d'arriver à déduire la position de l'écran par rapport à la position de la caméra et de son inclinaison. Nous verrons ensuite comment transformer et projeter chaque point sur l'écran. Dès lors qu'on sait projeter un point, il suffit d'en projeter trois pour obtenir une face, il suffit donc de projeter 3 points par faces.

Ensuite, nous aborderons le calcul de distance de chaque face à la caméra afin de réussir à ordonner les faces afin qu'elles s'affichent dans le bon ordre.

2.2 les variables libres et les variables liées

Pour commencer, on a la position de la caméra. C'est une variable qui va être modifiée par l'utilisateur lorsqu'il souhaitera se déplacer dans la scène. Ensuite, nous avons les angles de caméra, selon la

hauteur et de droite à gauche pour pouvoir voir dans toutes les directions. Avec ces 2 éléments, on va introduire la première variable liée : l'écran. Ce dernier est un plan dans l'espace sur lequel vont être projetés les différentes formes (l'utilisateur va voir ce qui est sur l'écran).

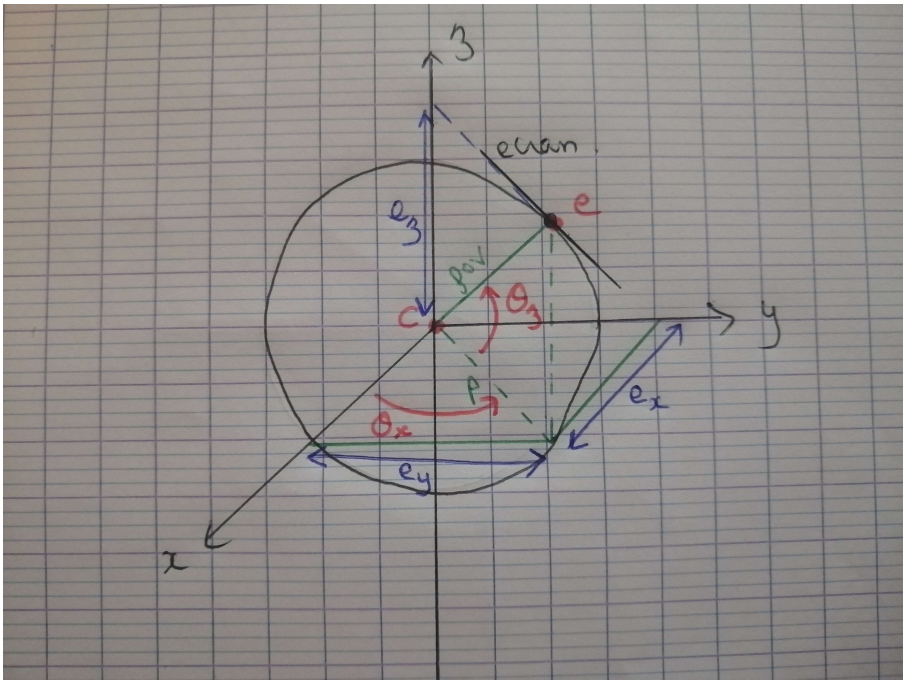


FIGURE 2 – Représentation de l'écran en fonction de la caméra

On note c la position de la caméra, e la position du centre de l'écran, et θ_x et θ_z les angles de la camera qui permettent d'orienter l'écran.

On souhaite définir la position de l'écran en fonction de l'inclinaison de la caméra et de la position de la caméra.

Projetons d'abord e dans le plan (xcy) : $\begin{cases} \sin\theta_x = \frac{e_y}{P} \implies P = \frac{e_y}{\sin\theta_x} \\ \cos\theta_x = \frac{e_x}{P} \implies P = \frac{e_x}{\cos\theta_x} \end{cases}$

D'autre part, on a : $\cos\theta_z = \frac{P}{fov} \implies P = \cos\theta_z \times fov$

On obtient naturellement : $\begin{cases} e_x = P \times \cos\theta_x = \cos\theta_z \times fov \times \cos\theta_x \\ e_y = P \times \sin\theta_x = \cos\theta_z \times fov \times \sin\theta_x \end{cases}$

e_z est simplement défini tel que : $e_z = \sin\theta_z \times fov$.

Nous avons donc les relations suivantes en rajoutant l'offset de la caméra (qui a été considéré comme l'origine dans notre schéma) : $\begin{cases} e_x = fov \times \cos\theta_x \times \cos\theta_z + c_x \\ e_y = fov \times \sin\theta_x \times \cos\theta_z + c_y \\ e_z = fov \times \sin\theta_z + c_z \end{cases}$

Nous devrons calculer à chaque itération la position de l'écran nécessaire aux calculs de projection grâce à la position de la caméra et de l'inclinaison de la caméra ainsi que du fov (field of view).

2.3 Rotation des points autour d'un axe (transformation)

On va appeler a_x,a_y,a_z la position du point à transformer, θ_x,θ_y,θ_z l'angle de la caméra (angle d'Euler même si on n'utilisera pas ici l'angle y) et enfin c_x,c_y,c_z est la position de la camera.

L'opération qui va suivre va permettre d'effectuer des rotations autour de l'angle x et y (de la caméra).

$$\begin{bmatrix} dx \\ dy \\ dz \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_x & \sin\theta_x \\ 0 & -\sin\theta_x & \cos\theta_x \end{bmatrix}}_{\text{rotation d'un angle } \theta_x} \times \underbrace{\begin{bmatrix} \cos\theta_z & \sin\theta_z & 0 \\ -\sin\theta_z & \cos\theta_z & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{rotation d'un angle } \theta_z} \times \left(\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} - \begin{bmatrix} c_x \\ c_y \\ c_z \end{bmatrix} \right)$$

Le point qui a pour coordonnée (d_x, d_y, d_z) est la rotation du point qui a pour coordonnée (a_x,a_y,a_z). Il va ensuite falloir projeter ce point sur l'écran qui est un plan.

2.4 Projection des points après la transformation

La projection est définie de la manière suivante : $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \frac{e_z \times d_x}{dz} + e_x \\ \frac{e_z \times d_y}{dz} + e_y \end{bmatrix}$

On rappelle que (e_x, e_y, e_z) représente la position de l'écran et que (d_x, d_y, d_z) représente le résultat de la rotation du point initial (que nous avons fait précédemment).

On sait dorénavant projeter 1 point, donc on peut en projeter 3 (ce qui correspond à une face). On peut donc projeter 3 points par faces afin de créer notre forme finale.

2.5 Affichage dans l'ordre des faces

On peut supposer à ce stade que l'on a terminé le moteur graphique, car on affiche toutes les faces, mais ce n'est malheureusement pas aussi simple, le moteur graphique va simplement afficher toutes les faces et il est possible que la dernière face qu'il affiche soit une face cachée par une autre à la base. Ainsi, il va afficher des faces qui sont censées être cachées par d'autres. Cela donne une impression de solide qui n'est pas fermé, car la face la plus éloignée cache la face la plus proche par exemple.

Pour remédier à ce problème, nous avons opté pour un affichage dans l'ordre : on affiche les faces les plus lointaines d'abord puis dans l'ordre décroissant de leur distance à l'écran. De cette manière, les faces les plus proches apparaissent devant les faces les plus éloignées. Pour cela, on calcule la position du centre de chaque face et on calcule la distance entre le centre de chaque face et la caméra. Il suffit ensuite d'afficher les faces dans l'ordre de leur distance décroissante afin d'ordonner les faces correctement.

Pour calculer le centre de la face, c'est une simple question de moyenne.

Prenons 3 sommets d'une face : $S1$, $S2$ et $S3$. Le centre de cette face est donc : $\begin{cases} enter_x = \frac{S1_x + S2_x + S3_x}{3} \\ center_y = \frac{S1_y + S2_y + S3_y}{3} \\ center_z = \frac{S1_z + S2_z + S3_z}{3} \end{cases}$

Pour ce qui est de la distance entre chaque face et l'observateur, on fait un simple calcul de distance : La distance entre F (le centre de la face) et C (la caméra) est donc :

$$distance_{F,C} = \sqrt{(F_x - C_x)^2 + (F_y - C_y)^2 + (F_z - C_z)^2}$$

3 Construction du moteur graphique

3.1 Bases du code

Pour commencer, on devait partir de rien et tout reconstruire, nous avons donc commencé par nous armer de quelques outils mathématiques comme le produit scalaire ou le produit vectoriel.

```
        //produit scalaire
        function dotProduct(u, v) {
          return u[0] * v[0] + u[1] * v[1] + u[2] * v[2];
        }

        // produit vectoriel en dimension 3
        function crossProduct(u, v) {
          const x = u[1] * v[2] - u[2] * v[1];
          const y = u[2] * v[0] - u[0] * v[2];
          const z = u[0] * v[1] - u[1] * v[0];
          return [x, y, z];
        }
```

Code Source 1 – Exemple de fonctions que nous avons implémentées

Il a fallu aussi définir les variables, en utiliser le moins possible pour que tout soit lié et naturel, nous avons donc besoin : - de la position de la caméra et son angle pour pouvoir la faire bouger et donc bouger les objets - de la position de l'écran, qui est face à la camera et qui permet de faire les projections. De cela, nous pouvions utiliser ces trois jeux de variables pour créer tout ce dont nous allons avoir besoin : nos bases, la distance à l'écran, et les projections.

```
    var fov = 400
    var camera = {
        x: 0,
        y: 0,
        z: 0
    }

    var camera_angle = {
        x: 0, //phi sphérique (angle bas-haut)
        y: 0, // angle phi dans les angles d'Euler => laisser constant à 0
        z: 0  //theta sphérique (angle droite gauche)
    }

    var display = {
        x: 0,
        y: 0,
        z: 0
    }
```

Code Source 2 – Les variables que nous avons déclarées

3.2 Rendu et affichage

Maintenant que nous avons ces variables, il a simplement fallu à chaque image mettre à jour la position de l'écran en fonction de la position de la caméra et de la position angulaire de la caméra.

La partie mathématique a déjà été évoquée précédemment [1]. Nous l'avons implémentée de la manière suivante :

```
        function NewDisplay() {
            display.x = fov*Math.cos(camera_angle.x)*Math.cos(camera_angle.y)
            display.y = fov*Math.cos(camera_angle.x)*Math.sin(camera_angle.y)
            display.z = fov*Math.sin(camera_angle.x)
        }
```

Code Source 3 – Fonction NewDisplay qui permet de recalculer l'écran

Comme nous ne savions pas du tout comment faire nous avons voulu commencer par la base , afficher une forme en 2D. Jusque là c'etait simple puisque Javascript dispose de méthode native pour gérer du 2D. Ensuite on est passé sur du 3D avec beaucoup de travail. Il fallait créer la fonction de projection, et donc choisir la projection que l'on voulait. On a opté pour une projection en perspective, car la plus logique et réaliste comme mentionné dans la partie 2. On a donc cherché et vérifé la formule, pour avoir une fonction de projection qui est réaliste, toutes les formules ont été redémontrées précédemment [2]. Nous avons donc implémenté la transformation du point (rotation) de la manière qui suit :

```
    function TransformedPoint(ax, ay, az) {
        cx = Math.cos(camera_angle.x)
        cy = Math.cos(camera_angle.y)
        cz = Math.cos(camera_angle.z)
        sx = Math.sin(camera_angle.x)
        sy = Math.sin(camera_angle.y)
        sz = Math.sin(camera_angle.z)
        mat1 = [
                [1, 0, 0],
                [0, cx, sx],
                [0, (-1)*sx, cx]
        ]
        mat3 = [
                [cz, sz, 0],
                [(-1)*sz, cz, 0],
                [0, 0, 1]
        ]
        vect = [[ax-camera.x, 0, 0],
                [ay-camera.y, 0, 0],
                [az-camera.z, 0, 0]
        ]

        //multMatrix permet de multiplier des matrices
        d = multMatrix(multMatrix(mat1, mat3), vect)

        vectd = [d[0][0], d[1][0], d[2][0]]
        return vectd
    }
```

Code Source 4 – Fonction de transformation d'un point

1. Voir section 2.2 p.3
2. Voir section 2.3 p.4 ainsi que la section 2.4 p.5

Après la transformation du point, il faut le projeter sur l'écran, ce que l'on à fait de la manière suivante :

```
        function JustProjectPoint(dx, dy, dz) {
            ex = display.x
            ey = display.y
            ez = display.z
            return [ez*dx/dz+ex, ez*dy/dz+ey]
        }
```

Code Source 5 – Fonction de projection d'un point

Suite à cette fonction, on a les coordonnées x et y de chaque point sur notre écran, il ne reste plus qu'à tracer les faces et notre écran de moteur graphique est un canvas en javacript (une sorte de tableau blanc intégrant des fonctions pour tracer qui sont natives au language). Nous avons opté pour la méthode fill, qui a ses avantages et inconvenients. Les avantages sont que cette méthode est très simple à utiliser, visuelle dans le code, et grâce à elle, on pouvait modéliser des formes sur un autre logiciel et les importer (nous y reviendrons plus tard) ; cependant il empeche certaines corrections, comme la mise en place efficace d'ombre sur une face (car il crée un triangle de couleur uniforme).

```
    //permet d'afficher une face à l'écran
    function DrawTriangle(co1, co2, co3, color) {
            ctx.fillStyle = color
            ctx.stokeStyle = "black"
            ctx.lineWidth   = 1;
            point1 = Project(co1[0], co1[1], co1[2]) //premier sommet
            point2 = Project(co2[0], co2[1], co2[2]) //second sommet
            point3 = Project(co3[0], co3[1], co3[2]) //troisième sommet

            if(point1[2] && point2[2] && point3[2]) {
                    ctx.beginPath();
                    ctx.moveTo(point1[0], point1[1]);
                    ctx.lineTo(point2[0], point2[1]);
                    ctx.lineTo(point3[0], point3[1]);
                    ctx.lineTo(point1[0], point1[1]);
                    ctx.stroke()  // dessine le contour
                    ctx.fill()    // remplie la face avec la couleur souhaitée
                    ctx.closePath()
            }
    }
```

Code Source 6 – La fonction DrawTriangle qui permet de dessiner une face

La fonction "Project", utilisée dans le bout de programme précédent, ne permet pas de projeter. Il transforme le point, puis vérifie si la face doit être affichée (est-elle derrière la caméra ?) et si elle l'est, la projete et retourne le résultat de la projection ainsi que l'information si elle est visible ou non. Cette fonction est implémentée de la manière suivante :

```
function Project(x, y, z) {
    tp = TransformedPoint(x, y, z)
    visible_point = true
    if(tp[2]<0) {
            visible_point = false //la face n'est pas visible
    }
    projection = JustProjectPoint(tp[0], tp[1], tp[2])
    return [projection[0], projection[1], visible_point]
}
```

Code Source 7 – La fonction Project qui permet de faire toutes les étapes de la projection

3.3 Gestion des faces

On sait dessiner une face, c'est super, mais maintenant il faudrait dessiner un objet ! Chaque objet est un emsemble de faces. Il faut donc ajouter des faces, puis les afficher en vidant la liste des faces, puis refaire cette opération à chaque image.

```
function AddTriangle(co1, co2, co3, color) {
    triangle_list.push([co1, co2, co3, color, 0])
    //on ajoute une face à la liste des faces à afficher
}
```

Code Source 8 – La fonction AddTriangle qui permet de rajouter une face à la liste des faces qui seras à afficher

Cette fonction va être appelée pour chaque faces. A la fin de chaque itération de la boucle du programme principale, on prend la liste des faces et on la vide en affichant chaque face retirée de cette liste de la manière suivante :

```
function DrawAllTriangle() {
    triangle_list = faceOrder(triangle_list)
    while((face = triangle_list.shift()) !== undefined) {
        DrawTriangle(face[0],face[1],face[2],face[3])//on draw la face retiré
    }
}
```

Code Source 9 – La fonction DrawAllTriangle qui permet d'afficher toutes les faces

On remarquera la présence de la fonction "faceOrder".

3.4 Affichage des faces dans l'ordre

On a une liste de faces et on les affiches, pourquoi cette fonction "faceOrder" ? La fonction "faceOrder" prend la liste des faces, les trie par leur distance à la caméra par ordre décroissant. De cette manière on affiche d'abord les faces les plus loin et ensuite les faces de plus en plus proche. Les faces cachées ne sont donc pas visibles et c'est la dernière touche qu'il fallait ajouter à notre moteur graphique pour qu'il fonctionne correctement.

```
function faceOrder (faces) {
    //partie du calcul de la distance entre la caméra et la faces
    faces.forEach(function(face){
      face[4] = distanceFromCamera(face[0],face[1],face[2])
    });

    //trie des faces dans l'ordre
    faces.sort(function(a, b) {
      return b[4] - a[4];
    });

    r = []

    //on met les faces dans l'ordre
    faces.forEach(function(face){
      r.push(face)
    });

    on retourne la liste des faces ordonnées
    return r
}
```

Code Source 10 – La fonction faceOrder qui permet de trier les faces

La fonction distanceFromCamera, elle, est faite de la manière suivante (d'après les calculs expliqués précédemment [3]) :

```
function centerOfTriangle(p1, p2, p3) {
    return [
            (p1[0]+p2[0]+p3[0])/3,
            (p1[1]+p2[1]+p3[1])/3,
            (p1[2]+p2[2]+p3[2])/3
    ]
}
```

Code Source 11 – La fonction faceOrder qui permet de trier les faces

Grâce à cela , nous avons pu faire un moteur graphique simple et efficace qui peut, sans être trop compliqué, afficher des objets 3D en 2D.

3. Voir section 2.5 p.5

4 Ou on en est arrivé

A la fin de notre projet, nous sommes arrivés à un moteur qui marche certes mais qui ne completait pas toutes nos attentes. Cependant, il a la particularité de pouvoir avoir une petite utilité. En effet, nous pouvons grâce à real intgine, afficher des formes 3D et les faire bouger, que ce soit par translation ou rotation . De plus nous pouvons importer une forme modélisée et l'afficher ce qui est pratique

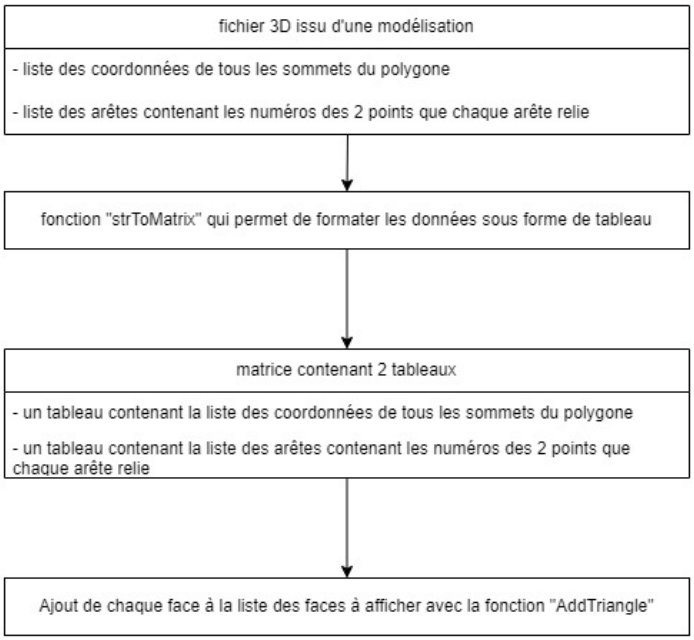


FIGURE 3 – Flux de données des objets importés

Cependant la version que nous avons fait contient beaucoup de défauts et est très simpliste. En effet, nous avons remarqué plusieurs défauts inhérents au choix que nous avons fait dans notre développement, comme le fait d'utiliser la méthode fill pour tracer nos formes. Ces défauts font qu'on ne peut gerer que les contours et le centre d'une face. Notre moteur graphique crée donc des situations non réalistes. Par exemple, on a le fait qu'il ne faut pas afficher les faces derrieres soit, donc avec fill nous avons caché les faces donc le centre est derriere nous. Néanmoins, il existe la situation où, lorsqu'il y a une face qui tourne et passe un peu derriere la camera, si on s'arrete à un angle où le centre de cette face est derriere nous mais il reste une partie devant nous, cette face n'est pas affichée. Cela est d'autant plus un défaut que lorsque l'on fait cela avec un solide, il y a donc une face qui disparait, et il s'ouvre devant nous.
Un autre problème est celui de l'ordre d'affichage, car on essaie en exportant chaque objet d'avoir le moins de faces possible (pour avoir moins de calculs donc plus de fluidité d'image). Cela implique que les faces générées peuvent être très grandes (sur les surfaces plane) et donc le centre de chaque face se trouve très loin des 3 sommets ce qui fait que l'ordre d'affichage va parfois placer ces très grandes faces devant des faces plus petite malgrès le fait que c'est le mauvais ordre. Par exemple :

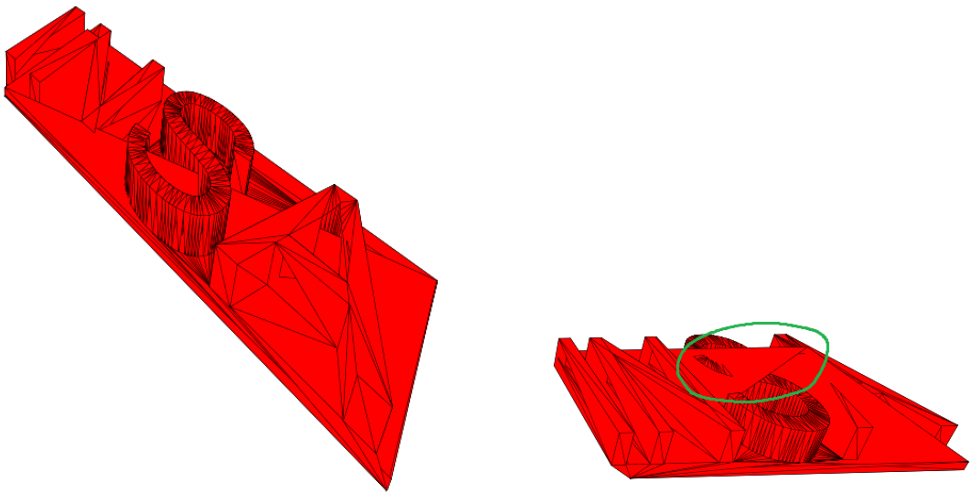


FIGURE 4 – Image de gauche sans bug, image de droite, on remarque le mauvais ordre d'affichage d'une grande face

Nous avons conscience de ce problème, qui n'est pas un bug puisque les faces sont censées être petites, mais il faut considérer cette limitation. Pour résoudre ce problème, soit on diviserait en plus de petites faces, soit on ferait un tri des faces beaucoup plus complexes, qui ne serait pas simplement basé sur la distance entre la caméra et le centre d'une face.

5 Difficultés

Comme dit précédemment, il y a eu beaucoup de difficultés lors de ce projet, dues aux contraintes que l'on avait et au manque d'expérience. Premièrement, vu que l'on codait en JavaScript, on affichait sur une page web, et donc on devait utiliser la console. À cause de cela, le débogage devenait très compliqué. De plus, il y a eu le problème d'ordre d'affichage des faces qui a créé des bugs, car comme il n'y avait pas d'ordre précisé, il y a des faces qui ne s'affichait pas dans le solide fermé.

Un autre problème rencontré est le fait que nous avons mal géré le temps, qui paraissait aussi un peu court pour l'ampleur du projet. Comme nous partions de zéro, il était compliqué d'avoir une bonne organisation du temps, car nous ne savions pas combien de temps une tâche allait durer, comme la fonction faceorder décrit au-dessus, où nous croyions que cela pouvait être fait en quelques jours, mais finalement cela a prit plusieurs semaines à cause de la non-connaissance du langage ou des problèmes de débogage, qui rend ce dernier impossible sur la fin.

À cause de tous ces problèmes, nous avons eu beaucoup de difficultés, mais au final nous avons créé un moteur graphique mathématique qui est interactif et fonctionnel, malgré quelques défauts.

6 Conclusion

En conclusion, notre projet de construction de moteur graphique a été une expérience passionnante et enrichissante. Bien que nous n'ayons pas atteint tous les objectifs initialement fixés. Nous avons tout de même réalisé des avancées significatives et avons réussi à créer quelque chose que nous aimons et dont nous sommes fiers.

Le processus de développement de ce moteur graphique nous a permis d'explorer de nouvelles idées, d'apprendre de nouvelles compétences et de repousser nos limites. Les obstacles et les défis rencontrés en cours de route nous ont permis d'acquérir une précieuse expérience et de mieux comprendre les complexités du développement d'un moteur graphique.

Il est important de reconnaître que la réalisation d'un projet aussi ambitieux que la construction d'un moteur graphique nécessite souvent plus de temps et de ressources que prévu initialement. Cependant, cela ne diminue en rien les accomplissements et les apprentissages que nous avons acquis tout au long de ce parcours.

En fin de compte, notre projet de construction de moteur graphique a été une expérience précieuse et stimulante. Nous avons développé des compétences techniques, renforcé notre esprit d'équipe à deux et découvert de nouvelles perspectives, de nouvelles applications de notre savoir-faire et un nouveau langage. Nous sommes fiers de notre travail et sommes contents d'avoir eu cette expérience pour l'avenir, notamment dans l'aspect gestion de projet, où nous gérerons mieux le temps et les objectifs pour mener à bien un travail d'équipe.

Table des Annexes

A1 Bibliographie

— Lengyel, Eric. Mathematics for 3D Game Programming and Computer Graphics Third Edition. Cengage Learning, 2011.

— Toutes les photos/images/illustations présentes dans ce rapport sont les nôtes à l'exception de la figure 1 à la page 3 qui est la miniature de la vidéo youtube de Unacademy Computer Science, "PERSPECTIVE PROJECTION (ONE POINT, TWO POINT, THREE POINT)", disponible à l'adresse suivante : https ://www.youtube.com/watch ?v=qWA1V-KZipc

A2 Code : Outils mathématiques

```
    //multiplication de 2 matrices
    function multMatrix(m1, m2) {
        var result = [];
        for (var i = 0; i < m1.length; i++) {
            result[i] = [];
            for (var j = 0; j < m2[0].length; j++) {
                var sum = 0;
                for (var k = 0; k < m1[0].length; k++) {
                    sum += m1[i][k] * m2[k][j];
                }
                result[i][j] = sum;
            }
        }
        return result;
    }

    function dotProduct(u, v) {    //produit scalaire
      return u[0] * v[0] + u[1] * v[1] + u[2] * v[2];
    }

    function vectorLength(v) {     //Norme
      return Math.sqrt(v[0] * v[0] + v[1] * v[1] + v[2] * v[2]);
    }

    function crossProduct(u, v) {    // produit vectoriel en dimension 3
      const x = u[1] * v[2] - u[2] * v[1];
      const y = u[2] * v[0] - u[0] * v[2];
      const z = u[0] * v[1] - u[1] * v[0];
      return [x, y, z];
    }

    // retourne le vecteur normal au triangle
    function TriangleToNormal(som1,som2,som3) {
      const u = [
        som2[0] - som1[0],
        som2[1] - som1[1],
        som2[2] - som1[2],
      ];
      const v = [
        som3[6] - som1[0],
        som3[7] - som1[1],
        som3[8] - som1[2],
      ];
      const normal = crossProduct(u, v);
      return normal;
    }

    function Normalise(u){    //normalise un vecteur
      return [u[0]/vectorLength(u), u[1]/vectorLength(u), u[2]/vectorLength(u)]
    }
```

A3 Code : Projection

```
1
2  //a est le point à transformer
3  function TransformedPoint(ax, ay, az) {
4          cx = Math.cos(camera_angle.x)
5          cy = Math.cos(camera_angle.y)
6          cz = Math.cos(camera_angle.z)
7          sx = Math.sin(camera_angle.x)
8          sy = Math.sin(camera_angle.y)
9          sz = Math.sin(camera_angle.z)
10         mat1 = [
11                 [1, 0, 0],
12                 [0, cx, sx],
13                 [0, (-1)*sx, cx]
14         ]
15         mat3 = [
16                 [cz, sz, 0],
17                 [(-1)*sz, cz, 0],
18                 [0, 0, 1]
19         ]
20         vect = [[ax-camera.x, 0, 0],
21                 [ay-camera.y, 0, 0],
22                 [az-camera.z, 0, 0]
23         ]
24
25         d = multMatrix(multMatrix(mat1, mat3), vect)
26
27         vectd = [d[0][0], d[1][0], d[2][0]]
28
29         return vectd
30 }
31
32 function JustProjectPoint(dx, dy, dz) {
33         ex = display.x
34         ey = display.y
35         ez = display.z
36         return [ez*dx/dz+ex, ez*dy/dz+ey]
37 }
38
39 function Project(x, y, z) {
40         tp = TransformedPoint(x, y, z)
41         visible_point = true
42         if(tp[2]<0) {  //on affiche la face que si elle est devant nous
43                 visible_point = false
44         }
45         projection = JustProjectPoint(tp[0], tp[1], tp[2])
46         return [projection[0], projection[1], visible_point]
47 }
```

A4 Code : Gestion des faces

```
    //permet d'afficher une face à l'écran
    function DrawTriangle(co1, co2, co3, color) {
            ctx.fillStyle = color
            ctx.stokeStyle = "black"
            ctx.lineWidth   = 1;
            point1 = Project(co1[0], co1[1], co1[2])
            point2 = Project(co2[0], co2[1], co2[2])
            point3 = Project(co3[0], co3[1], co3[2])

            if(point1[2] && point2[2] && point3[2]) {
                    ctx.beginPath();
                    ctx.moveTo(point1[0], point1[1]);
                    ctx.lineTo(point2[0], point2[1]);
                    ctx.lineTo(point3[0], point3[1]);
                    ctx.lineTo(point1[0], point1[1]);
                    ctx.stroke()  // dessine le contour
                    ctx.fill()    //remplit la face avec la couleur souhaité
                    ctx.closePath()
            }
    }
```

```
    //on ajoute une face à la liste des faces à afficher
    function AddTriangle(co1, co2, co3, color) {
            triangle_list.push([co1, co2, co3, color, 0])
    }

    //On rend toutes les faces (on affiche toutes les faces)
    function DrawAllTriangle() {
        triangle_list = faceOrder(triangle_list)
        while((face = triangle_list.shift()) !== undefined) {
            //On affiche la face qu'on vient de retirer
            DrawTriangle(face[0],face[1],face[2],face[3])
        }
    }
```

A5 Code : Gestion des variables

```
var fov = 400

var triangle_list = []

var camera = {
    x: 0,
    y: 0,
    z: 0
}

var camera_angle = {
    x: 0, //phi sphérique (angle bas-haut)
    y: 0, // angle phi dans les angles d'Euler => laisser constant à 0
    z: 0  //theta sphérique (angle droite gauche)
}

var display = {
    x: 0,
    y: 0,
    z: 0
}

function SetCamera(x, y, z) {
    camera.x = x
    camera.y = y
    camera.z = z
}

function SetCameraAngle(x, y, z) {
    camera_angle.x = x
    camera_angle.y = y
    camera_angle.z = z
}

// Fonction qui permet de calculer l'écran (qui est lié
// à la caméra et l'angle de la caméra).
// Cette fonction est appelé à chaque itération
function NewDisplay() {
    display.x = fov*Math.cos(camera_angle.x)*Math.cos(camera_angle.y)
    display.y = fov*Math.cos(camera_angle.x)*Math.sin(camera_angle.y)
    display.z = fov*Math.sin(camera_angle.x)
}

```

A6 Code : Gestion des "inputs" (souris et clavier)

```
    // Variables pour stocker les coordonnées précédentes de la souris
    var previousX = null;
    var previousY = null;

    var mouseeventon = false;

    document.addEventListener("keydown",keyPush);

    document.addEventListener("mousemove", handleMouseMove);
    document.addEventListener("wheel", handleMouseZoom);

    function handleMouseMove(event) {
        if(mouseeventon) {
            // Vérifier si les coordonnées précédentes existent
            if (previousX !== null && previousY !== null) {
                // Calculer le delta horizontal et vertical
                var deltaX = event.clientX - previousX;
                var deltaY = event.clientY - previousY;

                // Utiliser les valeurs de deltaX et deltaY à des fins quelconques
                camera_angle.x = camera_angle.x - deltaY*0.002
                camera_angle.z = camera_angle.z + deltaX*0.002
            }
        }

        // Mettre à jour les coordonnées précédentes avec les coordonnées actuelles
        previousX = event.clientX;
        previousY = event.clientY;
    }

    function handleMouseZoom(event) {
        if(mouseeventon) {
            // Vérifier si le déplacement de la souris est un zoom
            if (event.deltaY < 0) {
                // Zoom in (approche)
                fov = fov - event.deltaY*0.1;
            } else {
                // Zoom out (éloignement)
                fov = fov - event.deltaY*0.1;
            }
        }
    }

    function keyPush(evt) {
        switch(evt.keyCode) {
            case 40: // bas
                    camera_angle.x = camera_angle.x - 0.05
                break;
            case 38: // haut
                    camera_angle.x = camera_angle.x + 0.05
                break;
            case 39: // droite
```

```
                    camera_angle.z = camera_angle.z + 0.05
                break;
            case 37: // gauche
                    camera_angle.z = camera_angle.z - 0.05
                break;

            case 81 : // q
                    camera.x = camera.x - 1
                break;
            case 68: // d
                    camera.x = camera.x + 1
                break;
            case 69: // e
                    camera.y = camera.y - 1
                break;
            case 65: // a
                    camera.y = camera.y + 1
                break;
            case 83: // s
                    camera.z = camera.z - 1
                break;
            case 90: // z
                    camera.z = camera.z + 1
                break;
        }
    }
```

A7 Code : Parsing et Chargement d'une map

```
function strToMatrix(str) {
    str = str.replace(/[^0-9.\s\r\nvf]/g, "");
    matrix = str.split("\n").map(line => line.split(" "))
    v = []  //liste des coordonnées des sommets
    f = []  //liste des faces contenant les identifiants des sommets

    //les 6 première lignes sont des métadonnées
    for(let i=5; i<matrix.length; i++) {
        if(matrix[i][0] == 'f' || matrix[i][0] == 'v') {
            matrix[i][3] = matrix[i][3]//.slice(0, -1)
            if(matrix[i][0] == 'v') {
                v.push(matrix[i])
            } else {
                f.push(matrix[i])
            }
        }
    }

    //ici on convertie le tableau de string en tableau de flottant
    var v = v.map(function(row) {
      return row.map(function(value) {
        return parseFloat(value, 10);
      });
    });

    //ici on convertie le tableau de string en tableau d'entier
    var f = f.map(function(row) {
      return row.map(function(value) {
        return parseInt(value, 10);
      });
    });

    //symétrie car sinon la projection inverse certaines mesures
    v.forEach(function(face){
      face[1] = (-1)*face[1]
      face[3] = (-1)*face[3]
    });

    return [v, f]
}

document.getElementById('inputFile').addEventListener('change', function() {
    var file = new FileReader();
    file.onload = () => {
        //usermap est la variable qui stocke les faces importés
        usermap = strToMatrix(file.result)
      console.log(usermap)
    }
    file.readAsText(this.files[0])
});
```

A8 Code : Main Code

```
var canvas = document.getElementById("Canvas");
var ctx = canvas.getContext("2d");

SetCamera(-27, -18, -24)
SetCameraAngle(1.594, 0, 1.4879999999999)

setInterval(mainloop);  //boucle infini de la fonction mainloop

//fonction appelé à chaque itération (à chaque rendu d'une image)
function mainloop() {
    NewDisplay()
    ReloadInformation()
    ctx.fillStyle="white";
    ctx.fillRect(0,0,canvas.width,canvas.height); //on efface tout

    //si la map a été importé, la liste des faces est dans usermap
    //on ajoute alors toutes les faces via la fonction AddTriangle
    if(typeof usermap != "undefined") {
        for(let i = 0; i < usermap[1].length; i++) {
            offsetX = 0
            offsetY = 0
            offsetZ = 0
            AddTriangle(
                [offsetX+usermap[0][usermap[1][i][1]-1][1],
                offsetY+usermap[0][usermap[1][i][1]-1][2],
                offsetZ+usermap[0][usermap[1][i][1]-1][3]
                ],
                [offsetX+usermap[0][usermap[1][i][2]-1][1],
                offsetY+usermap[0][usermap[1][i][2]-1][2],
                offsetZ+usermap[0][usermap[1][i][2]-1][3]
                ],
                [offsetX+usermap[0][usermap[1][i][3]-1][1],
                offsetY+usermap[0][usermap[1][i][3]-1][2],
                offsetZ+usermap[0][usermap[1][i][3]-1][3]
                ],
                "green"
            )
        }
    }

    DrawAllTriangle() // on Draw toutes les faces qu'on a ajouté à la liste

    //ici on bloque la caméra si on regarde trop haut
    if (camera_angle.x > 3.14) {
        camera_angle.x = 3.14
    }
    //ici on bloque la caméra si on regarde trop bas
    if (camera_angle.x<0) {
        camera_angle.x=0
    }
}
```