Rendu « Do It Yourself »

Sylvie Chambon, Simone Gasparini et Géraldine Morin

Objectifs

Le but de ces 4 séances de TPs est d'**implémenter un moteur de rendu, en `Java`**. Plus précisément, un objet 3D et la pose d'une caméra étant définis, vous allez créer l'image vue par la caméra, comme dans la figure 1. La figure 7 vous donne l'architecture du programme (cf. dernière page du sujet).

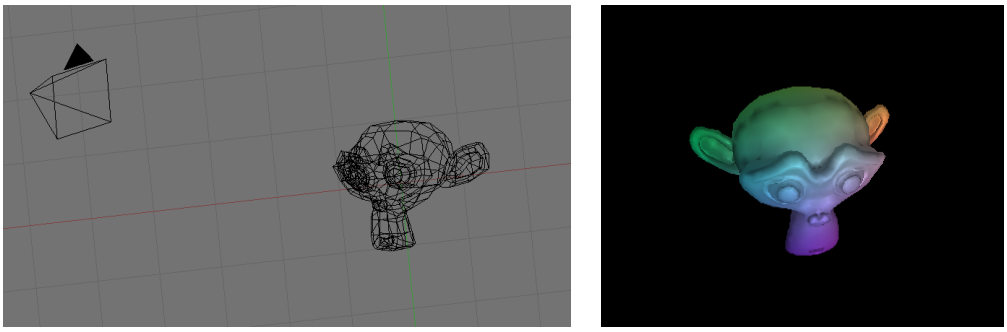


Figure 1 – À gauche, en entrée, nous connaissons la pose d'un appareil photographique et un objet défini en 3D. À droite, nous présentons l'image générée depuis le point de vue de la caméra. L'objectif de ce TP est de générer ce rendu.

Travail noté

Vous avez quatre séances pour réaliser ce sujet en binôme, **à renseigner sous moodle, dès la première séance**. Vous serez évalué·e·s **en groupe et individuellement** en réalisant une démonstration de codes et en répondant à des questions à l'oral **lors d'une séance dédiée indiquée dans votre emploi du temps** (le créneau est variable suivant les groupes, soyez attentifs et attentives).

Description des données fournies et manipulées

- La **caméra (position et orientation) ainsi que les informations nécessaires pour l'éclairage** définis dans un fichier `*.scene` où les lignes vides et les commentaires, commençant par #, ne sont pas pris en compte. Ce fichier contient :
  - le nom du fichier contenant l'objet : `nom_du_fichier_objet.OFF` ;
  - la position de la caméra : trois coordonnées $x_{cam}, y_{cam}, z_{cam}$, ;

- le point d'intérêt : point vers lequel « regarde » la caméra $x_{look}, y_{look}, z_{look}$;
- le *up vector* indiquant l'orientation de la caméra et qui ne correspond pas à la direction Y du repère caméra (mais permet de l'obtenir) ;
- la distance focale f en nombre de pixels : l'écran est placé, dans le repère caméra, dans le plan $Z = f$;
- la hauteur h et la largeur w de l'image en pixels (que nous supposons carrés) ;
- les données concernant la lumière (détaillées dans la suite de l'énoncé).

La figure 2 illustre tous les éléments contenus dans ce fichier.

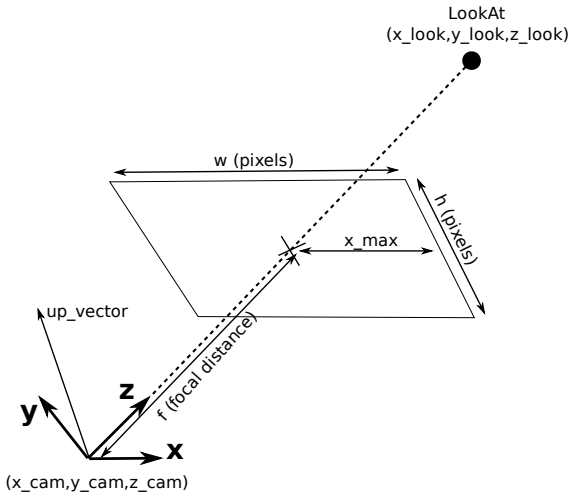


FIGURE 2 – Les paramètres donnés dans le fichier de configuration `*.scene`.

- Les **objets 3D** : définis dans un fichier texte de format `*.OFF`. Il s'agit d'un **maillage 3D** dont les sommets ont chacun une couleur et éventuellement des coordonnées de texture. Plus précisément, si nous notons N_S, N_A et N_F, respectivement le nombre de sommets, le nombre d'arêtes et le nombre de faces, le fichier est formé ainsi :

NUMÉRO DE LIGNE	CONTENU DU FICHIER
1	`OFF`
2	$N_S \;\; N_F \;\; N_A$
3	$x_0 \;\; y_0 \;\; z_0 \;\; R_0 \;\; G_0 \;\; B_0 \;\; x_{t_0} \;\; y_{t_0}$
...	...
(3+nbSommets)	$x_{N_S-1} \;\; y_{N_S-1} \;\; z_{N_S-1} \;\; R_{N_S-1} \;\; G_{N_S-1} \;\; B_{N_S-1} \;\; x_{t_{N_S-1}} \;\; y_{t_{N_S-1}}$
(3+nbSommets+1)	`3` $i \;\; j \;\; k$
...	...

Pour simplifier, nous supposons que chaque face est un triangle, d'où le « 3 » en début de ligne, il correspond au nombre de sommets d'une face donnée. De plus, i, j et k correspondent aux indices des sommets de la face, l'indice étant défini par l'ordre d'apparition dans la liste des sommets qui normalement commence à 0.

Vous n'avez pas à programmer la lecture du fichier `*.OFF`.

Description des classes fournies

Les classes contenant les outils utiles sont :

- `Mesh` permet de générer le maillage 3D à partir d'un fichier d'extension `.OFF`. **Les coordonnées pour les sommets (*Vertices*), les composantes pour les couleurs et les indices de sommets pour les faces sont stockés séquentiellement**.
  Par exemple, le tableau de sommets est un tableau à une dimension de longueur $3 \times N_S$ et de forme $[x_0, y_0, z_0, x_1, y_1, z_1, \ldots, x_{N_S-1}, y_{N_S-1}, z_{N_S-1}]$.
- `Scene` récupère les paramètres de la scène, donnés dans le fichier d'extension `.scene`.
- `GraphicsWrapper` permet d'ouvrir une fenêtre graphique rectangulaire (origine en haut à gauche).
- `Fragment` représente un pixel, donc un sommet sur une grille 2D, avec ses attributs.

Les classes effectuant le rendu sont :

- `Renderer` programme principal qui contient les éléments de base `Mesh`, `Scene` et `GW` (GraphicsWrapper) et aussi une `Transformation`, un `Rasterizer`, un `Shader`.
- `Transformation` contient les matrices de transformation 3D-3D de changement de repère monde en repère caméra, et 3D-2D de projection en coordonnées homogènes.
- `Rasterizer` permet de discrétiser les primitives géométriques 2D (arêtes et faces) sur une grille de pixels et de générer des `Fragment`s.
- `Shader` gère l'affichage des `Fragment`s.

Partie préliminaire

Travail à faire: Tests et prise en main des codes fournis

1. Décompresser l'archive fournie sur moodle. Le dossier contient :
   - Un sous-dossier `src` contenant toutes les classes java nécessaires et/ou à compléter, y compris un sous-sous-dossier `algebra` comprenant toutes les opérations matricielles nécessaires, ainsi que les objets `Matrix`, `Vector` (vecteur de taille arbitraire) et `Vector3`
   - Un sous-dossier `test` contenant des programmes pour tester les différentes classes ;
   - Un sous-dossier `data` contenant toutes les données fournies.
2. Pour compiler utiliser la commande `make` depuis la racine du code fourni.
3. Pour lancer les tests utiliser la commande `make tests`.

   Remarque : Certains tests sont incomplets et fonctionneront totalement lorsque vous aurez complété les fichiers concernés. Toutefois, assurez vous qu'ils compilent tous et qu'il n'y a pas d'erreurs à l'exécution.
4. Pour lancer le moteur de rendu utiliser la commande (`make run SCENE=data/example1.scene`). Pour l'instant, il ne se passe pas grand chose (affichage d'une fenêtre noire qui se ferme automatiquement au bout de quelques secondes) puisque vous n'avez pas modifié les codes fournis mais assurez vous que cela compile et qu'il n'y a pas d'erreurs à l'exécution.

1 Formation de l'image : du 3D vers le 2D

Le but de cette partie est de définir les transformations permettant de passer d'un sommet 3D (`Vector`) à un point sur l'écran (`Fragment`).

Travail à faire: Transformation d'un sommet du maillage en point sur l'image

Nous souhaitons obtenir les coordonnées pixelliques du projeté sur l'image de chaque sommet 3D composant l'objet. Pour cela, transformez par étapes votre scène en implantant :

1. la fonction `setLookAt` (dans `Transformation.java`) qui définit la matrice `Transformation.worldToCamera`, comme vu en cours.
2. la matrice de projection dans `setProjection`, `Transformation.projection`.
3. la matrice de calibrage – changement de repère 2D/2D pour être dans le repère image – dans `setCalibration`, `Transformation.calibration`, également vue en cours.
4. la fonction `projectPoints` qui utilise les trois matrices précédentes pour projeter un sommet en 3D dans l'image.

Vérifiez que la profondeur du pixel projeté a été enregistrée dans ses attributs car elle sera nécessaire à l'étape de gestion de la profondeur. Cela signifie qu'**il faut penser à conserver z ($z \neq 1$)**.

Vous pouvez maintenant visualiser les sommets projetés sur l'image (la fonction appelée par le `main` de Renderer est `RasterizeEdge` (de la classe `Rasterizer`).

2 Rasterisation d'une arête, remplissage d'une face

La deuxième partie s'intéresse au tracé des segments et au rendu des faces. Les sommets sont maintenant projetés dans l'image. Pour faire un rendu filaire, chaque arête doit être dessinée.

2.1 Tracé d'un segment

Travail à faire

Pour tracer un segment, nous utilisons l'**algorithme de Bresenham**. Dans la fonction `RasterizeEdge` (`Rasterizer`), vous trouverez l'appel pour l'affichage d'un seul pixel (que nous dilatons un peu pour une meilleure visualisation). Vous remplacerez ce code par l'affichage des pixels du segment (code commenté). Les attributs sont interpolés linéairement par rapport aux extrémités du segment (en appelant la fonction `interpolate2`). Vous afficherez le rendu filaire du modèle, comme dans la figure 3.(a).

2.2 Remplissage d'un triangle en 2D

Travail à faire

Vous allez maintenant remplir les faces du modèle, cf. la figure 3.(b). Cette partie va proposer un premier rendu d'une face, en interpolant linéairement en 2D les attributs des pixels (*Fragment*) sommets de la face (ici toujours un triangle). Le but est de remplir un triangle à partir de ses trois sommets définis par trois fragments, i.e. de compléter la fonction `rasterizeFace` de `Rasterizer`. Le remplissage consiste à trouver l'ensemble des pixels à l'intérieur du triangle et à déterminer leur couleur via une interpolation : **on interpolera linéairement l'ensemble des attributs du fragment**. Dans un premier temps, il est nécessaire de déterminer les points p qui appartiennent au triangle. Pour cela, vous devez utiliser, d'une part, les coordonnées des sommets, pour déterminer les intervalles de variations en x et en y, et, d'autre part, conserver les points p pour lesquels **les signes des coordonnées barycentriques sont positifs**.
Les coordonnées barycentriques (α, β, γ) de $p = (x, y)$ dans la base (x_i, y_i), $_{i=1,2,3}$, peuvent être calculées à partir de la matrice $\mathbf{C}$ retournée par la fonction `Rasterizer.makeBarycentricCoordsMatrix`, cf. début du code de la fonction `rasterizeFace`. Vous calculerez les coordonnées barycentriques (α, β, γ) de $p = (x, y)$ et ferez l'interpolation des attributs (somme pondérée par les coordonnées barycentriques) :

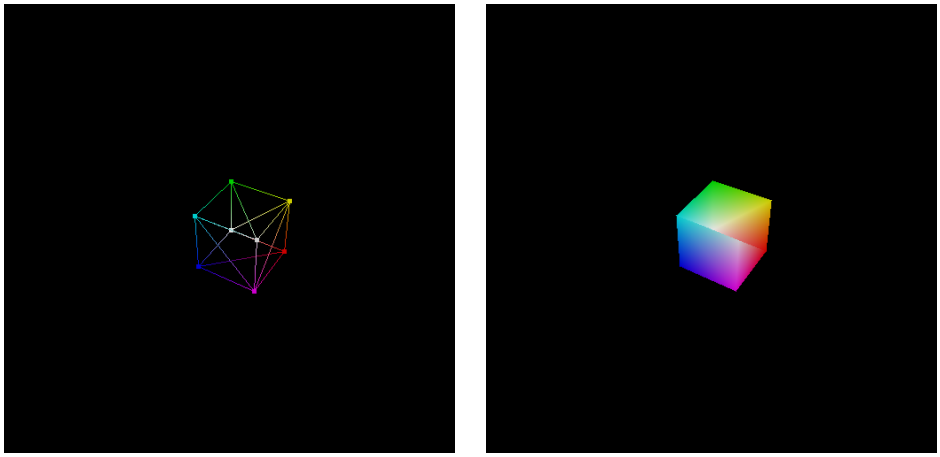
$$\begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix} = \mathbf{C} \begin{pmatrix} 1 \\ x \\ y \end{pmatrix} \tag{1}$$

avec

$$\mathbf{C} = \frac{1}{A} \begin{pmatrix} x_2y_3 - x_3y_2 & y_2 - y_3 & x_3 - x_2 \\ x_3y_1 - x_1y_3 & y_3 - y_1 & x_1 - x_3 \\ x_1y_2 - x_2y_1 & y_1 - y_2 & x_2 - x_1 \end{pmatrix}$$

où A est le double de l'aire du triangle : $A = x_2y_3 - x_3y_2 + x_3y_1 - x_1y_3 + x_1y_2 - x_2y_1$.

Après interpolation du fragment, vous l'affichez directement avec un appel au `shader` : `shader.shade(votre_fragment)`. Il vous faut également penser à décommenter dans `Renderer` la partie de code en dessous de ce commentaire : `// solid rendering, no lighting`.



(a) (b)

FIGURE 3 – Résultats obtenus avec le cube (`data/example0.scene`). En (a), il s'agit de l'affichage filaire alors qu'en (b), il s'agit de l'affichage plein sans gestion de la profondeur.

3 Gestion de la profondeur

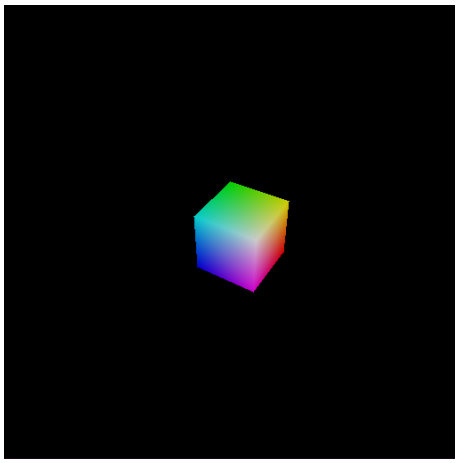


FIGURE 4 – Résultat avec le cube après gestion de la profondeur.

Pour gérer correctement la profondeur, c'est-à-dire, n'afficher que ce qui est visible (et non caché par une face plus proche de la caméra), l'information de profondeur doit être gardée. Lorsque vous affichez un pixel, il faut donc vérifier si vous ne l'avez pas déjà affiché : dans ce cas, il faut comparer les profondeurs des différents points 3D auxquels correspondent ce même pixel.

Travail à faire: le DepthBuffer

La gestion de la profondeur se fait en gardant dans un tampon de profondeur, la profondeur courante de chaque pixel, qui sera ici la profondeur du point 3D actuellement affiché en ce pixel. Pour cela, dans le module `Depthbuffer`, complétez les méthodes `testFragment` et `writeFragment` et créez une instance de `PainterShader` qui remplacera `SimpleShader` dans la méthode `init` du `Renderer` (il s'agit de permuter les commentaires dans le code de `Renderer`).
Vous devez à présent visualiser un résultat qui prend en compte la profondeur correctement, cf. la figure 4.

Partie 4: Prise en compte de l'éclairage – Gouraud

Comme vous l'avez vu en cours, deux algorithmes d'éclairage – hormis l'éclairage constant par morceaux – peuvent être implémentés : celui de Gouraud ou celui de Phong. Ici, nous vous proposons d'implémenter d'abord l'algorithme de Gouraud qui est plus simple et moins coûteux.

Les paramètres dont vous aurez besoin pour l'éclairage sont dans le fichier **.scene*:

- l'intensité de la lumière ambiante: I_a ;
- une lumière ponctuelle définie par ses coordonnées homogènes $x\ y\ z\ w$ (pour pouvoir mettre la lumière à l'infini) et son intensité: I_d ;
- les constantes du matériau de l'objet : K_a, K_d, K_s, s.

Travail à faire: Calcul de l'intensité en chaque sommet

Pour prendre en compte l'éclairage, il est nécessaire de calculer la normale pour dériver l'intensité. Conformément à l'algorithme de Gouraud, la normale en chaque sommet du maillage doit être calculée dans la fonction `computeNormals` du module `Mesh`. La normale en un sommet est obtenue comme la moyenne des normales des faces auxquelles appartient le sommet (attention il faut un vecteur normé). Ensuite, l'intensité est calculée dans `Lighting`, avec la méthode `applyLights` à chaque sommet, pour chaque composante de couleur, en utilisant la formule vue en cours:

$$I = I_{ambient} + I_{diffus} + I_{specular}$$

1. $I_{ambient} = I_a * K_a$,
2. $I_{diffus} = I_d * K_d * \cos(n, l)$ où n est la normale, l la direction de la lumière,
3. $I_{specular} = I_d * K_s * \cos^s(h, n)$ où h est la bissectrice de l et v, la direction de la caméra.

Cette fonction est appelée dans *Renderer* via la méthode `projectVertices` appelée dans `rendersolid`. Ensuite, pour chaque point appartenant à une face, son intensité est déduite comme une combinaison affine des intensités des sommets de la face (en utilisant simplement l'interpolation des *Fragments* que vous avez programmé en Partie 2). Enfin, il vous faut également penser à mettre à nouveau en commentaire, dans `Renderer`, la partie de code en dessous de ce commentaire : `// solid rendering, no lighting` et à décommenter la partie en dessous de ce commentaire : `// solid rendering, with lighting`. L'exemple fourni dans la figure 5 vous permet de voir l'effet de l'ajout de l'éclairage sur l'exemple du singe.

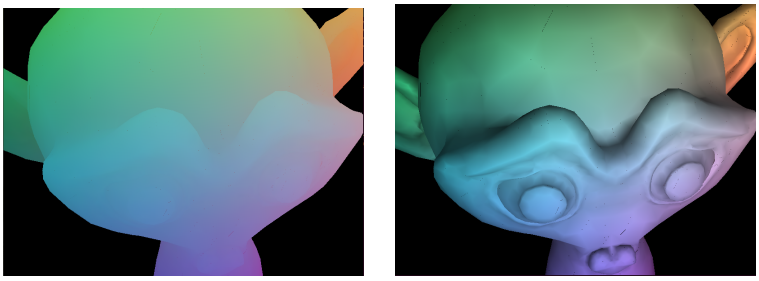


FIGURE 5 – Résultat avec le singe, avant gestion de l'éclairage (à gauche), et après gestion (à droite).

Partie 5: Plaquage de textures

Dans cette dernière partie vous allez plaquer des textures sur un objet 3D. Pour cela, des coordonnées de texture sont ajoutées à chaque sommet dans le fichier OFF. Vous utiliserez donc l'exemple `example_textured.scene`.

Travail à faire: Calcul de la couleur à partir de la texture

Les attributs correspondants au coordonnées de texture sont stockés dans les attributs du Fragment. Vous pouvez les retrouver en utilisant la méthode `getAttribute (int, int)` de `Fragment.java` (cf. commentaire de la méthode). Compléter `Texture.java` et `TextureShader.java`.
Attention, le `textureShader` ne fonctionne pas si le `depthBuffer` n'est pas implémenté. Il vous faut également modifier `Renderer` en suivant les instructions données dans le code source.

Le résultat comporte « la mauvaise distorsion perspective ». Pour cela, nous vons donnons la classe `PerspectiveCorrectRasterizer.java` qui corrige ce problème. Vous pouvez lire la correction de cette classe, remplacer la ligne

`rasterizer = new Rasterizer (shader)` par

`rasterizer = new PerspectiveCorrectRasterizer (shader)` dans le programme principal pour observer le résultat après la correction perspective.

L'exemple fourni dans la figure 6 vous permet de voir l'effet de cette correction sur l'exemple texturée du mur.

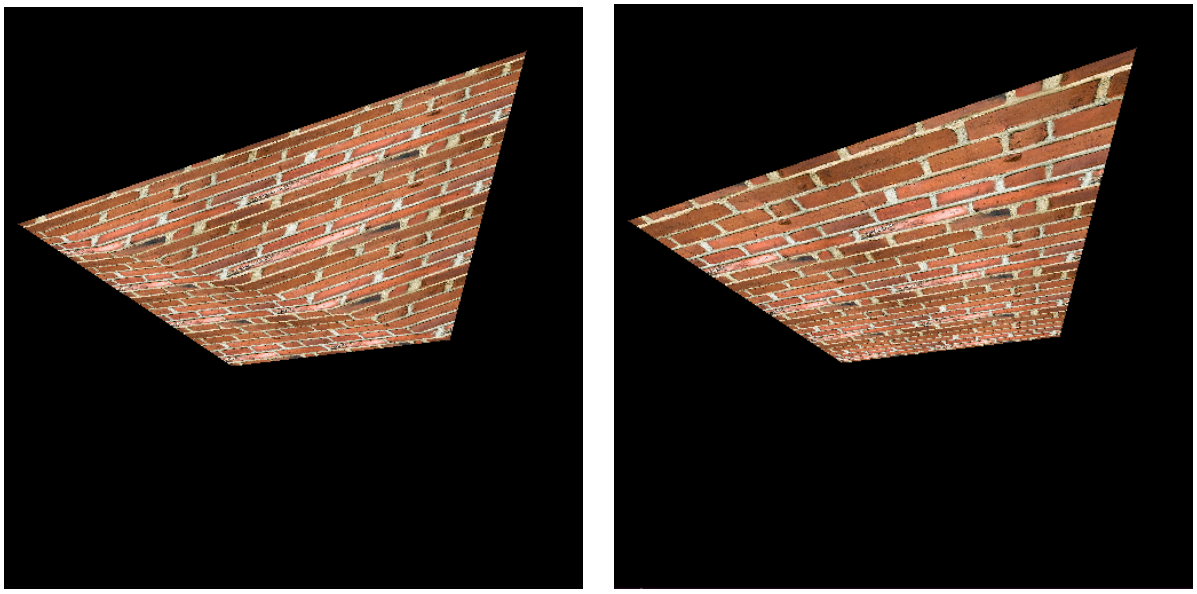


FIGURE 6 – Résultat avec la texture, avant correction de la perspective (à gauche), et après correction (à droite).

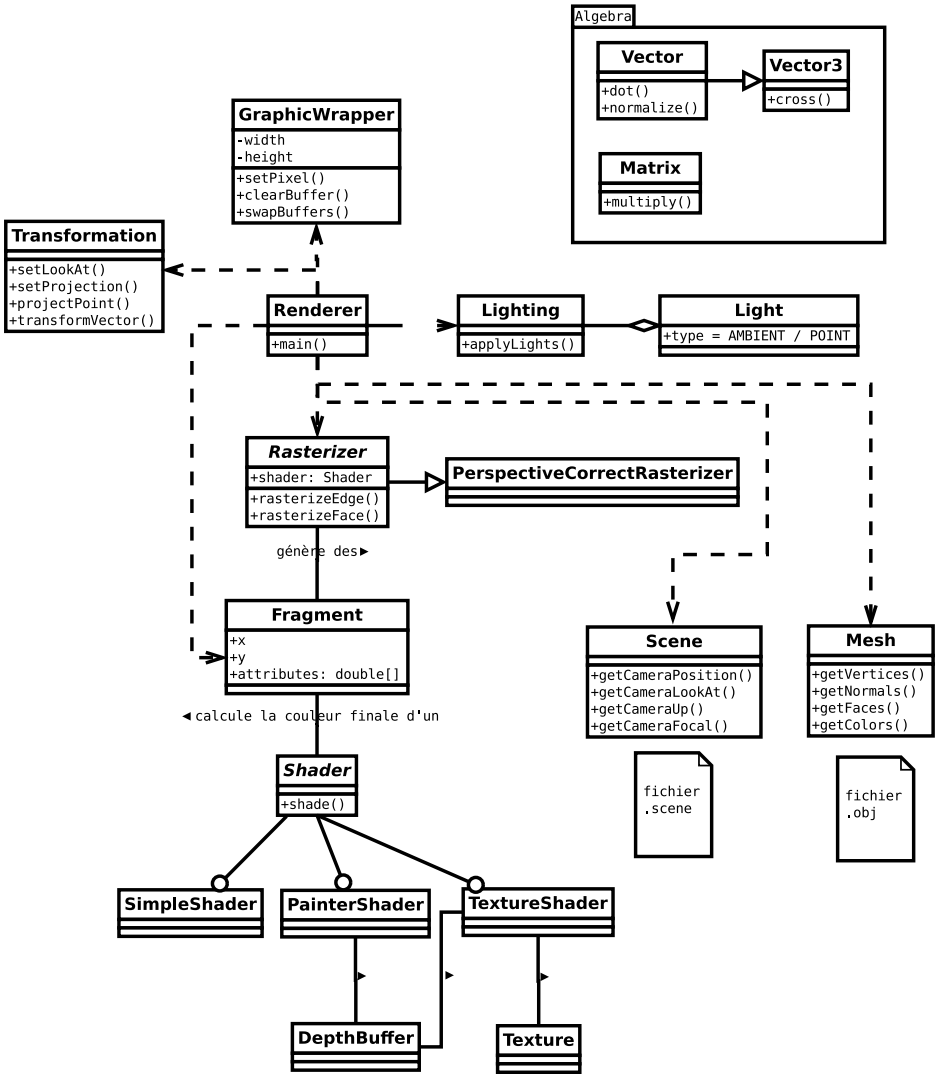


FIGURE 7 – Organisation du programme.