TP MSO Synthèse d'images: Rendu projectif
CPE

durée - 4h

2012/2013

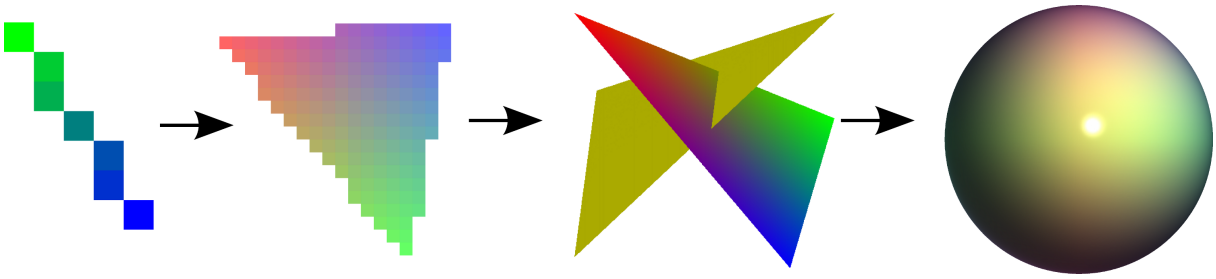

FIGURE 1 – Différentes étapes du rendu projectif : Tracé de segments discrets, tracé de triangles colorés, gestion de la profondeur, affichage d'un modèle 3D illuminé.

1 But

L'objectif de ce TP est de coder un outil de rendu de modèle 3D générique. Il consiste à implémenter un rendu projectif de triangles illuminés (shading). La méthode utilisée sera similaire à celle du pipe-line standard utilisé par les cartes graphiques (API type OpenGL/Direct3D), mais sera ici *émulé* par un programme CPU.

- Dans un premier temps, nous nous intéresserons au tracé de segments discrets, avec interpolation de couleurs.
- Dans un second temps, nous implémenterons le remplissage de triangles délimités par 3 segments discrets ainsi que l'interpolation de couleurs.
- Enfin, nous procéderons à la projection et au calcul d'illumination dans le cas d'un triangle 3D de manière à réaliser son rendu sur une image 2D.

2 Prise en main de l'environnement

2.1 Les différents répertoires

Les répertoires contiennent différentes librairies qui vous sont fournies soit complètes, soit à compléter au fur et à mesure du TP.

- `libimage/` contient les classes et fonctions de bases permettant de manipuler une image 2D, une ligne discrète, un triangle discret. On complètera différentes fonctions et classes de ce répertoire pendant le TP.
- `libprojectif/` contient les classes et fonctions liées à la mise en place du rendu projectif et du calcul d'illumination. Ces classes et fonctions font appels à la librairie de manipulation d'image de base et elles seront complétées en deuxième partie de TP.

- `lib3d/` est le répertoire contenant une librairies entièrement codé pour manipuler des points en 2D, 3D, et 4D, ainsi que des matrices 2x2, 3x3 et 4x4.
- `libcommon/` contient la définition des exceptions de bases qui sont émises par les différentes fonctions et classes en cas de problèmes.
- `local/` contient la fonction `main()` et ce qui sera nécessaire aux appels globales pour une scène spécifique.

2.2 Programme main

La fonction *main* (dans le répertoire `local`) réalise les différents appels d'affichage en récupérant les exceptions spécifiques aux classes de ce TP. Les appels sont dans l'ordre d'exécution :

- Initialisation d'une classe de gestion d'image de taille 20×20 pixels en mémoire.
- Remplissage de l'ensemble des pixels par la couleur blanche.
- Remplissage d'un pixel (15,14) par la couleur verte.
- Remplissage d'un ensemble de pixel par une couleur variant du noir au rouge.
- Ecriture de l'image dans un format ascii classique *ppm* (non compressé). L'image pouvant être manipulée par d'autres outils annexes : gimp, ...

Les différents appels sont présentés en fig. 2.

```
//une image
image im(20);

//remplissage de l'image en blanc
im.fill(color(1,1,1));


//pixel (5,4) en vert
im(pos2(15,14))=color(0,1,0);

for(int k=0;k<9;++k)
{
    pos2 u(k+3,3); //position variant suivant x
    color c(k/8.0,0,0); //couleur variant de noir à rouge
    im(u)=c; //ecriture de la couleur dans l'image
}


//sauve l'image dans un fichier
std::cout<<"Draw picture on disk ..."<<std::endl;
im.export_file("my_pic.ppm");
std::cout<<"[OK]"<<std::endl;
```

FIGURE 2 – Fonctions d'appels principales.

2.3 Principales classes utilisables

Un ensemble de classes de bases vous est fourni pour faciliter la mise en place de ce TP. L'ensemble des classes reste cependant bas-niveau et elles ne font pas appels à de librairies externes. Elles restent donc modifiables pour votre TP.

2.3.1 Classe pos2

Une classe de conteneur de base de 2 entiers. Elle sert principalement à indexer un pixel de coordonnées (kx,ky) dans une matrice.

```
pos2 u(5,4);       //u=(5,4)
u.x()=8;           //u=(8,4)
u=2*u-pos2(1,0)    //u=(15,8)
```

2.3.2 Classe vec3

Classe de conteneur de point 3D quelconque. En interne (x,y,z) étant stockés sur 3 flottants. Contiens de nombreuses fonctions vectorielles.

```
vec3 p(1.5,1.0,-2);              //p=(1.5,1.0,-2.0)
vec3 o=p.dot(vec3(1,1,0))*p;     //o=<p,(1,1,0)> p
o+=p;                            //o=o+p
vec3 e=o.normalized();           //e=o / ||o||
e.z()=4;                         //e=(e.x,e.y,4)
```

2.3.3 Classe color

Classe de conteneur d'une couleur (r,g,b) où chaque canal est encodé sur un flottant $\in [0,1]$.

```
color c(1.0,1.0,0.0) // c <- jaune
c.b()=1.0;           // c <- blanc
color bleu=(0,0,1.0);

//magenta = (1-0.4)*rouge + 0.4*bleu
color magenta=(1-0.4)*color(1,0,0)+0.4*color(0,0,1);
```

2.3.4 Classe image

Classe de gestion d'une image (r,g,b). L'image est stockée en interne sous forme de vecteur concaténé de `color`. La classe implémente l'initialisation, l'accès protégé aux données de couleurs et l'export d'une image dans un fichier *ppm*.

```
image im(500); // créé une image 500x500
im.fill(color(1,0,0)); // colore l'image en rouge

// colore pixel(10,15) en vert
im(pos2(10,15))=color(0,1,0);

//exporte l'image dans fichier mon_pimage.ppm
im.export_file(''mon_image.ppm'');
```

2.3.5 Classe image zbuffer

Classe dérivée de image permettant la gestion d'un buffer de profondeur (=une image à niveau de gris stockant la profondeur de chaque pixel).

2.3.6 Classe render engine

Classe d'aide permettant le rendu de triangle 3D et maillage 3D. Le rendu consiste à projeter les sommets dans l'espace 2D de la caméra, appeler le calcul d'illumination sur chaque sommet, et envoyer les informations de triangles colorés 2D dans le rendu d'image.

2.3.7 Diagramme des classes

Un diagramme des classes principales est proposé en fig. 4. Vous pourrez vous y reporter au fur et à mesure de l'avancement du TP pour vous aider à comprendre la communication entre les différents modules et fonctions proposées.

Notez que d'autres classes sont présentes (en dépassant le cadre de ce TP), la librairie étant réutilisée pour des TP ultérieurs ainsi qu'en 5ETI. L'ensemble peut être utile pour vos projets scolaire ou professionnels futurs.

2.4 Travail préliminaire

Question 1 *Etudiez le programme de la fonction* `main()` *pour comprendre les différents appels à la classe d'image. Compilez et executez le programme et observez l'image obtenue.*

Question 2 *Dans le répertoire de l'image, lancez en ligne de commande la commande suivante :*
`convert my_pic.ppm my_pic.jpg`
La commande `convert` *de la librairie* `ImageMagick`[1] *est un outil pratique de convertion et manipulation d'images. Le format jpg est plus concis et prendra moins d'espace mémoire que le ppm lors du stockage de vos résultats ou de l'inclusions d'images dans vos comptes-rendus.*

Question 3 *Faites en sorte d'ajouter un pixel jaune aux coordonnées (18,16), et affichez l'image résultante.*

Question 4 *Réalisez une image de taille* 20×20 *similaire à celle proposée en fig. 3.*

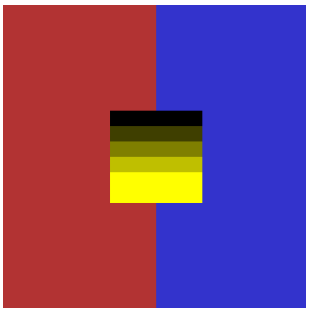

FIGURE 3 – Image à réaliser.

1. http ://www.imagemagick.org/script/index.php

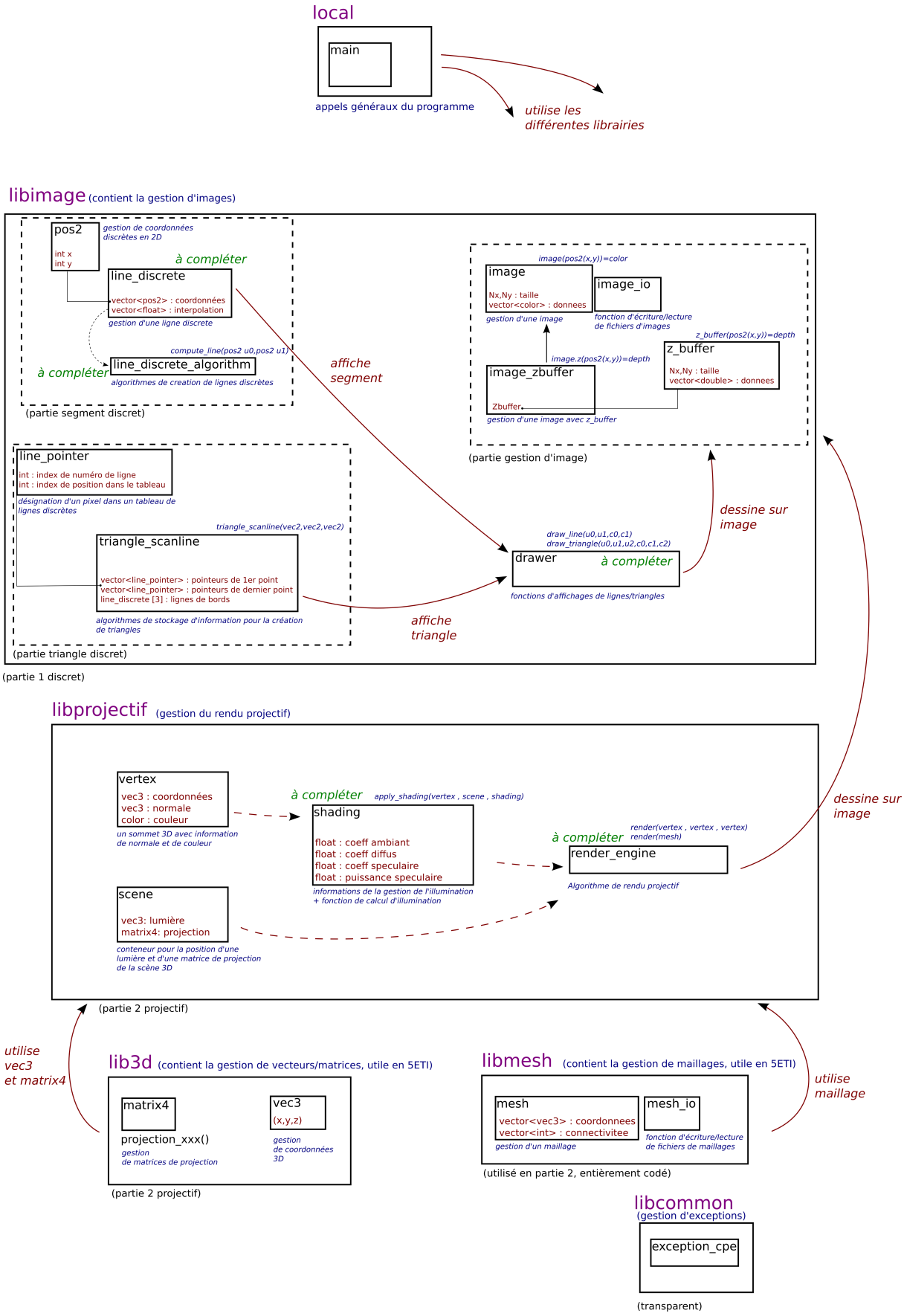

FIGURE 4 – Diagramme des classes et fonctions proposées.

3 Partie I : Tracé de segment

Dans un premier temps, on va s'intéresser au cas du tracé d'un segment discret de couleur uniforme.

3.1 Stockage d'un segment discret

Le fichier `line_discrete_algorithm` contient différents algorithmes permettant de générer un segment discret. Le segment n'est pas directement affiché sur l'image, mais l'ensemble des coordonnées des pixels appartenant à ce segment sont stockés dans la structure `line_discrete` (dans le fichier du même nom).

La structure `line_discrete` est la suivante :

```
struct line_discrete
{
  std::vector<pos2> coordinate;
};
```

Il s'agit d'une structure stockant sous forme de vecteur l'ensemble des coordonnées d'un segment.
En plus du stockage, la structure possède une méthode `add()` permettant d'ajouter une coordonnée au vecteur (réalise un `push_back()` sur le `std::vector`). Une méthode `size()` retourne la taille du vecteur.

Différentes fonctions permettent également d'effectuer quelques opérations sur un segment pré-existant.

- `swap_xy()` permet d'échanger les coordonnées x et y.
- `symetry_x()` permet de réaliser la symétrie du segment par rapport à l'axe x.
- `symetry_y()` permet de réaliser la symétrie du segment par rapport à l'axe y.
- `reverse()` permet d'inverser l'ordre des coordonnées dans le vecteur.

Ces fonctions seront utilisées pour gérer les différents cas de symétries lors de la création d'un segment.

3.2 Calcul d'un segment

L'appel à la création d'un segment discret quelconque entre 2 positions (u_0, u_1) est obtenue par l'appel à la fonction `compute_line` du fichier `line_discrete_algorithm`.

Cette fonction calcule les coordonnées d'un segment discret et retourne une structure de type `line_discrete`. La figure 5 illustre la modélisation d'un tel stockage.

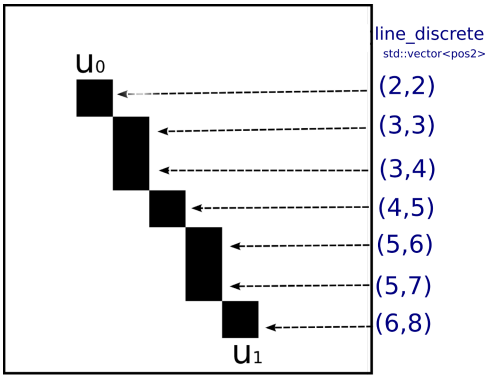

FIGURE 5 – Segment discret et vecteur d'information associé.

Cette fonction vient se décomposer en sous-fonctions suivant les différents cas possibles :

1. Ligne confondue avec un seul point.
2. Ligne horizontale.
3. Ligne verticale.
4. Ligne diagonale.
5. Lignes obliques (construit à l'aide de l'algorithme de Bresenham).

Dans le code fourni, le cas de la ligne confondue en un seul point, et le cas de la ligne vertical sont déjà traités.

Question 5 *Observez la fonction* `compute_line_vertical` *déjà implémentée. Notez que la fonction pré-suppose que l'on fournisse un* $y_{\min}$ *et* $y_{\max}$*. Comment est traité le cas où* $y_1 < y_0$ *dans la fonction* `compute_line()` *?*

3.3 Affichage d'un segment

Une fois une ligne calculée et stockée dans la structure `line_discrete`, celle-ci peut être affichée à l'aide des fonctions du fichier `drawer`.

Question 6 *Observez la fonction* `draw_line()`*, que fait-elle jusqu'à présent ?*

Faites appel à cette fonction dans le `main()`. Par exemple, on pourra appeler :

```
draw_line(im,pos2(2,2),pos2(2,8),color(1,0,0))
```

Question 7 *Qu'observe t-on sur l'image résultat ?*

Question 8 *Modifiez la fonction* `draw_line()` *pour satisfaire à l'algorithme voulu. Pour s'aider, on rappel que :*

- *Le calcul d'un segment discret entre une position* $[u_0, u_1]$ *se réalise par l'appel*
  `compute_line(u0,u1);`
- *La vérification si une coordonnée* `u` *est dans l'image se fait par l'appel*
  `if(im.check_position(u)==true)`
- *L'affectation d'une couleur* `c` *aux coordonnées* `u` *se réalise par*
  `im(u)=c;`

Question 9 *Vérifiez que l'on est bien capable d'afficher une ligne verticale par cet appel.*

3.4 Ligne horizontale

Question 10 *En vous inspirant de la fonction* `compute_line_vertical()`*, complétez désormais la fonction* `compute_line_horizontal()`*.*

Dans l'ensemble des tracés, pensez qu'il est possible de visualiser le contenu du vecteur d'une ligne discrète à l'aide de la méthode *debug()*. Par exemple, l'appel suivant dans la fonction `main()`

```
line_discrete line=compute_line(pos2(2,2),pos2(8,2));
line.debug();
```

permet d'afficher la ligne retournée par la fonction.

Question 11 *Mettez à jour la fonction* `compute_line()` *pour faire appel à la création d'une ligne horizontale lorsque cela est nécessaire. Traitez également le cas de symétrie.*

Question 12 *Vérifiez que vous puissiez tracer une ligne horizontale avec différents appels dans la fonction* `main()`*. Vérifiez en particulier que vous puisiez tracer une ligne allant de gauche à droite et de droite à gauche afin de bien satisfaire à la symétrie.*

3.5 Ligne diagonale

Question 13 *Complétez la fonction* `compute_line_diagonal()` *qui calcule les coordonnées d'une ligne positionnée sur une diagonale.*
Décommentez la partie correspondante de la fonction `compute_line()` *qui est écrite pour gérer les 4 cas symétriques.*
Vérifiez que vous puissiez tracer des segments dans les 4 directions diagonales.

3.6 Ligne oblique

Dans le cas où une ligne ne correspond à aucun des cas précédent, on utilise l'algorithme de Bresenham. L'algorithme présenté en cours pour le *1er quadrant* correspond au cas où $dx > 0$, $dy > 0$ et $dx > dy$.

Question 14 *Implémentez l'algorithme présenté en cours dans la fonction* `compute_bresenham`.

Question 15 *Décommentez dans la fonction* `compute_line()` *la partie correspondante au premier quadrant. Vérifiez que vous puissiez tracer une ligne oblique dans ce quadrant.*

Question 16 *Décommentez dans la fonction* `compute_line()` *les parties correspondantes aux cas des autres quandrants obtenues par symétries. Quelle formule est utilisée pour calculer le symétrique des coordonnées par rapport à l'axe x ? par rapport à l'axe y ?*

Question 17 *Vérifiez que vous puissiez tracer un segment correspondant aux différents quadrants.*

3.7 Ligne avec couleur interpolée

On va chercher à implémenter la fonction suivante :

```
void draw_line(image& im,
               const pos2& u0,
               const pos2& u1,
               const color& c0,
               const color& c1);
```

Cette fois, le segment tracé doit avoir une couleur variant linéairement entre la couleur `c0` et la couleur `c1` comme montré par exemple en fig. 6.

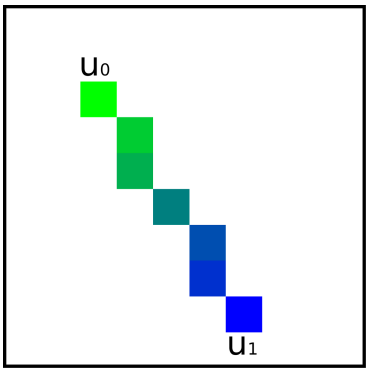

FIGURE 6 – Segment discret avec couleurs interpolées linéairement entre le vert (0,1,0) et le bleu (0,0,1).

Pour cela, on va calculer la position relative α d'un pixel dans le segment afin d'interpoler les couleurs linéairement.

Soit $\mathbf{u}$, une coordonnée quelconque d'un segment discret entre $\mathbf{u}_0$ et $\mathbf{u}_1$. La coordonnée relative de $\mathbf{u}$ par rapport à $\mathbf{u}_0$ et $\mathbf{u}_1$ peut être obtenue par

$$\alpha = \frac{\|\mathbf{u} - \mathbf{u}_0\|}{\|\mathbf{u}_1 - \mathbf{u}_0\|} .$$

Ensuite, la couleur $\mathbf{c}$ associée au pixel de coordonnées $\mathbf{u}$ est donnée par l'interpolation linéaire :

$$\mathbf{c} = (1 - \alpha)\mathbf{c}_0 + \alpha\mathbf{c}_1 .$$

On notera que cela assure bien d'avoir la couleur $\mathbf{c}_0$ au pixel de coordonnées $\mathbf{u}_0$, et la couleur $\mathbf{c}_1$ au pixel de coordonnées $\mathbf{u}_1$.

3.7.1 Mise en place d'une coordonnée de position relative

Nous allons désormais stocker l'information de coordonnée relative dans la structure `line_discrete`. Pour cela, ajoutez un champs permettant de stocker un ensemble de valeurs flottantes. On aura donc :

```
struct line_discrete
{
  std::vector<pos2> coordinate; //coordonnee du pixel (x,y)
  std::vector<float> interpolation; //position relative
};
```

Question 18 *Modifiez la structure* `line_discrete` *pour ajouter ce champs.*

Question 19 *Complétez la fonction* `compute_interpolation()` *qui va venir calculer ce vecteur de position relative pour toutes les coordonnées contenues dans le vecteur* `coordinate`*.*

On fera attention à ce que le calcul soit bien celui d'une norme, et que celui-ci soit réalisé dans l'espace des nombres flottants et non des entiers (pour rappel : la division de deux entiers renvoie un nombre entier.)

Question 20 *Vérifiez votre calcul de coordonnées relatives en générant différentes lignes discrètes et en affichant le résultat sur la ligne de commande. (On pourra modifier la fonction* `debug()` *afin qu'elle affiche également les coordonnées relatives.)*
Notez que vos coordonnées relatives doivent être des nombres qui varient continuement entre 0 *et* 1*. La première valeur doit toujours être* 0*, et la dernière sera toujours* 1*. Quelle règle appliquez vous lorsqu'un segment est confondu en un seul point ? Vérifiez ce cas particulier et assurez-vous que vous n'avez pas de division par zéro.*

3.7.2 Affichage de l'interpolation de couleur

Question 21 *Mettez en place la fonction d'affichage proposée d'une ligne avec une couleur interpolée (vous pouvez soit créer une nouvelle fonction, soit écraser votre fonction précédente avec couleur uniforme).*

Notez que l'algorithme d'affichage est le suivant :

```
Calculer coordonnees entre (u0,u1)
Mettre a jour les coordonnees relatives
  Pour toutes coordoonnees u=(x,y)
      Si u est dans l'image
         alpha <- coordonnee relative courante
         couleur = (1-alpha)c0 + alpha c1
         affecter couleur a l'image en position u
```

Question 22 *Vérifiez que vous arriviez à afficher une ligne dont les couleurs sont interpolées.*

4 Partie II : Affichage de triangle

4.1 Triangle de couleur uniforme

Soit un triangle défini par les 3 positions (u_0, u_1, u_2).
On calcul dans un premier temps les 3 segments frontières par l'algorithme de Bresenham $[u_0, u_1], [u_1, u_2], [u_2, u_0]$.

Une fois ces 3 segments calculés, on calcule pour x donné, la position $y_{\min}$ et $y_{\max}$ correspondante. Il est alors possible de compléter l'intérieur du triangle en affichant une succession de segment verticaux compris entre $[y_{\min}, y_{\max}]$ pour la coordonnée x donnée. Cet algorithme est appelée `scanline`. Une illustration est proposée en fig. 7.

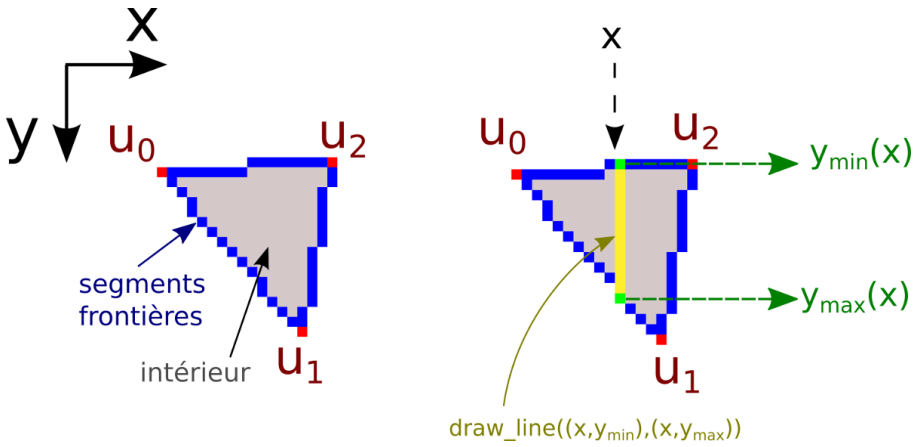

FIGURE 7 – Méthode de remplissage de triangle dite *scanline*. À x fixé, on calcule la valeur $y_{\min}$ et $y_{\max}$ correspondante. Puis un segment vertical est tracé entre ces deux positions extrêmes.

La classe `triangle_scanline` (située dans le fichier du même nom) met en place cet algorithme pour le tracé de triangle. L'algorithme est directement calculé dans le constructeur de la classe.

Le principe est le suivant :

1. On calcul les 3 segments discrets que l'on stocke dans la variable `line[3]`.
2. Pour toutes les valeurs de x, on met à jour des pointeurs qui vont indiquer la position la plus haute ($y_{\min}$) et la plus basse ($y_{\max}$) dans l'image.

Ces pointeurs utilisent la structure intermédiaire `line_pointer` qui stocke le numéro de ligne (0,1, ou 2), et l'indice dans la ligne désignée.

Question 23 *Appelez le code montré en fig. 8 dans le fonction* `main()` *pour comprendre la manipulation de la structure* `triangle_scanline`.

Notez que la récupération des informations de positions relatives peut être réalisée par l'appel suivant :

```
line_pointer p0=scanline.first_point[k];
float alpha=scanline.line[p0.line_number].interpolation[p0.index];
```

(On pourra éventuellement récupérer cette information de position relative plus simplement en créant une fonction similaire à `get_coordinate()`).

```
triangle_scanline scanline(pos2(2,2),pos2(2,16),pos2(18,4));
for(int k=0;k<scanline.size();++k)
{
    line_pointer p0=scanline.first_point[k];
    line_pointer p1=scanline.last_point[k];

    int x=scanline.get_coordinate(p0).x();
    int y_min=scanline.get_coordinate(p0).y();
    int y_max=scanline.get_coordinate(p1).y();

    std::cout<<"Pour x="<<x<<", je varie entre y=["<<y_min<<","<<y_max<<"]"<<std::endl;
}
```

FIGURE 8 – Exemple d'utilisation de triangle scanline.

Question 24 *Complétez la fonction* `draw_triangle()` *pour afficher un triangle à l'aide de la classe triangle scanline.*

4.2 Triangle de couleur interpolée

On cherche désormais à définir une couleur différente pour chaque sommet du triangle. L'algorithme de remplissage doit ainsi interpoler ces couleurs à l'intérieur de celui-ci.

Dans notre cas, on implémentera l'interpolation barycentrique classique correspondant à une interpolation linéaire des couleurs.

Il existe 2 méthodes possibles de calcul. L'interpolation barycentrique directe : Soit un triangle défini par 3 sommets A, B, C de valeur (ex. couleurs) respectives f_A, f_B et f_C. La valeur f d'une position P situé à l'intérieur du triangle est donné par

$$f = uA + vB + wC\ ,$$

avec (u, v, w) coordonnées barycentriques à l'intérieur du triangle (tel que $(u, v, w) \in [0, 1]$ et $u + v + w = 1$). Les coordonnées sont calculables sous la forme suivante (pouvant s'exprimer géométriquement par rapport d'aires) :

$$\begin{cases} u = \frac{\det(X-C,B-C)}{\det(A-C,B-C)} \\ v = \frac{\det(X-C,C-A)}{\det(A-C,B-C)} \\ w = \frac{\det(X-B,A-B)}{\det(A-C,B-C)} \end{cases}$$

Cette approche directe illustrée en fig. 9 gauche ne prend cependant pas avantage de l'algorithme incrémental suivi pour le remplissage du triangle.

Une seconde approche plus efficace consiste à réaliser le calcul d'interpolation suivant 2 interpolations linéaires consécutives. Soit $c_{u_{min}}$ et $c_{u_{max}}$ les deux couleurs associées aux positions correspondantes à $y_{\min}$ et $y_{\max}$ pour x-donné dans l'algorithme *scanline*. La couleur finale associée à la position (x, y), avec $y \in [y_{\min}, y_{\max}]$ peut être obtenu par interpolation linéaire de $c_{u_{min}}$ et $c_{u_{max}}$. L'approche est schématisée en fig. 9 droite.

Question 25 *Réalisez la fonction*

```
void draw_triangle(image& im,
                   const pos2& u0,
                   const pos2& u1,
```

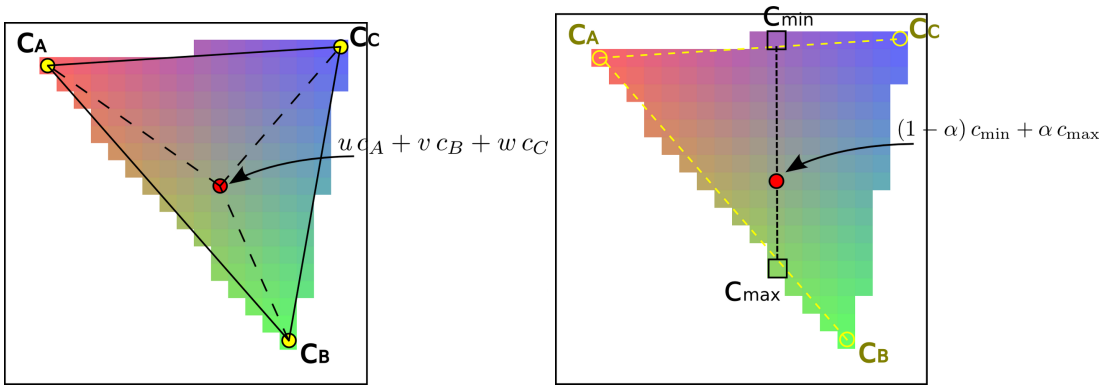

FIGURE 9 – Interpolation de couleurs dans un triangle. Gauche : interpolation par calculs de coordonnées barycentriques. Droite : Interpolation suivant deux calculs d'interpolation linéaires.

```
const pos2& u2,
const color& c0,
const color& c1,
const color& c2);
```

permettant d'afficher un triangle dont les couleurs sont interpolées linéairement entre c_0, c_1 et c_2 d'après la méthode illustrée en fig. 9 droite.

5 Partie III. Rendu projectif

Dans la troisième partie, nous nous intéressons au cas de rendu de triangle 3D par projection. L'aspect tridimensionnel du rendu 2D provient du calcul d'illumination pour chaque sommet. On implémentera l'illumination (shading) par la méthode de **Gouraud**.

Dans un premier temps, on va s'intéresser à gérer une image + un buffer de coordonnées de profondeur (ZBuffer). Dans un second temps, on s'intéressera au calcul d'illumination afin de rendre compte de l'apparence 3D.

5.1 Gestion du ZBuffer

Lors de l'affichage de plusieurs triangles à des profondeurs variables, il n'est pas possible de connaitre aisément les pixels situés en avant de ceux situés en arrière. Pour cela, on utilise un algorithme dit de *Zbuffer* consistant à associer à chaque pixel une profondeur. On n'affiche alors que les pixels situés en avant et donc potentiellement visibles à la caméra.

L'algorithme de MAJ du Zbuffer est le suivant.

```
Zbuffer <- initialise a +infini

Pour tout pixel u de couleur c de profondeur z
  Si(z>0 && z<Zbuffer(u))
    Zbuffer(u)=z
    pixel(u)=c
```

La classe `image_zbuffer` (dans le fichier du même nom) est une extension de la classe d'image simple mais gérant en plus un Zbuffer. L'accès à la coordonnée de profondeur pour un pixel donné est obtenu à l'aide de la méthode `im.z(pos2 )`.

Pour réaliser le dessin d'un segment avec gestion du Zbuffer, on va désormais implémenter la fonction suivante :

```
void draw_line(image_zbuffer& im,
               const pos2& u0,const pos2& u1,
               const color& c0,const color& c1,
               double z0,double z1);
```

Cette fonction devra afficher un segment discret dont les couleurs sont interpolées linéairement et dont les profondeurs vont également être interpolées linéairement entre z_0 et z_1. On affichera le pixel en question que si il n'existe pas d'autre pixel ayant déjà été écrit dans l'image avec une profondeur plus faible.

Question 26 *Implémentez la fonction proposée.*

Question 27 *Vérifiez que les pixels les plus proche en valeur de z sont affichés* devant *les pixels plus éloignés en valeur de z.*

Question 28 *Affichez la carte de profondeur dans une image séparée.*

De manière similaire, on réalise la fonction d'affichage de triangle avec interpolation des valeurs de profondeurs :

```
void draw_triangle(image_zbuffer& im,
                   const pos2& u0,const pos2& u1,const pos2& u2
                   const color& c0,const color& c1,const color& c2
                   double z0,double z1,double z2);
```

Question 29 *Implémentez la fonction de tracé de triangle avec gestion de profondeur. Vérifiez votre résultat sur différents cas, et affichez le buffer de profondeur dans un fichier séparé.*

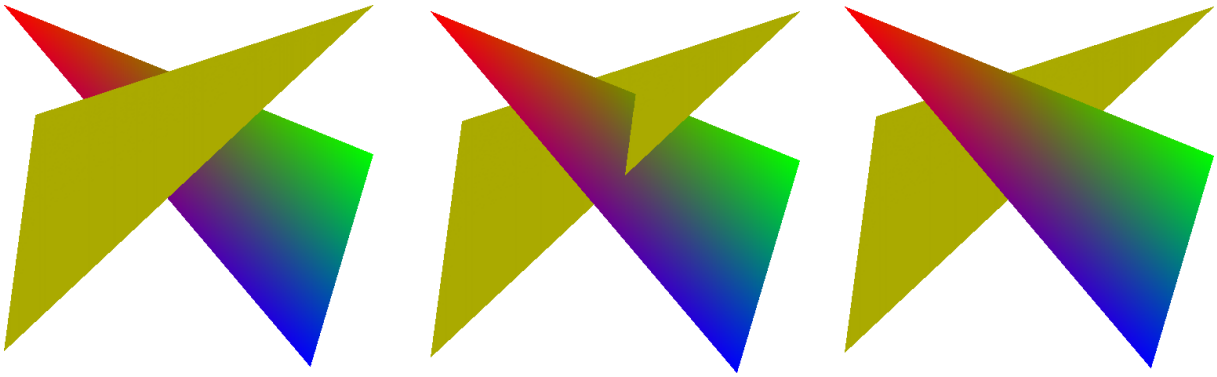

FIGURE 10 – Trois triangles affichés avec différentes valeurs de profondeur. La gestion du Zbuffer permet de n'afficher que les pixels les plus proches.

5.2 Projection

Les sommets 3D de coordonnées $\mathbf{p} = (p_x, p_y, p_z)$ sont affichés sur un écran 2D.

Dans le cas de ce TP, on va supposer que la caméra est orientée suivant l'axe z. La classe `matrix4` (dans le fichier du même nom) permet d'utiliser différentes matrices de projection (perspectives ou orthographiques).

Question 30 *Observez la méthode* `matrix4::projection_perspective()` *et la matrice associée.*

La classe `scene` permet de stocker l'une de ces matrice de projection. Elle est déjà proposée avec des paramètres standard.

L'appel au code suivant permet de réaliser la projection d'un sommet 3D quelconque.

```
scene ma_scene;
vec3 p(0.2,0.5,-0.7);
vec3 p2=ma_scene.matrix_projection*p;

std::cout<<"Position initiale="<<p<<
        ", position apres projection="<<p2<<std::endl;
```

La classe `render_engine` permet de mettre en place l'algorithme de rendu projectif pour un triangle, ou un ensemble de triangles (maillage).

La méthode `render_engine::render()` prend en entrée 3 structures de types `vertex`. Cette structure est un conteneur pour une position, une nomale et une couleur.

Question 31 *Complétez la méthode* `render_engin::render()` *permettant de calculer la projection de 3 sommets de type* `vertex`*. (Dans un premier temps, on sautera l'étape de calcul d'illumination de l'algorithme proposé, et on affectera directement la couleur originale au pixels).*

Question 32 *Assurez vous que l'appel suivant de la fonction* `main()` *permette d'afficher un triangle de couleur.*

```
image_zbuffer im(200);
im.fill(color(1,1,1));
scene scene_parameter;
shading shading_parameter;
render_engine engine(im,scene_parameter,shading_parameter);

vertex v0(vec3(-1,-1,-3),vec3(0,0,1),color(1,0,0));
vertex v1(vec3( 1,-1,-3),vec3(0,0,1),color(0,1,0));
vertex v2(vec3( 1, 1,-3),vec3(0,0,1),color(0,0,1));
engine.render(v0,v1,v2);
```

Il est également possible d'appeler l'affichage d'un maillage complet (classe `mesh` entièrement codé d'avance). Par exemple, le code proposé en fig. 11, permet d'obtenir les deux images montrées en fig. 12.

```
image_zbuffer im(1000);
im.fill(color(1,1,1));
scene scene_parameter;
shading shading_parameter;
render_engine engine(im,scene_parameter,shading_parameter);

mesh maillage;
maillage.load_file("data/bunny.off");
maillage.compute_normal();
maillage.fill_color_normal();
engine.render(maillage);
```

FIGURE 11 – Code permettant de demander l'affichage d'un maillage complet.

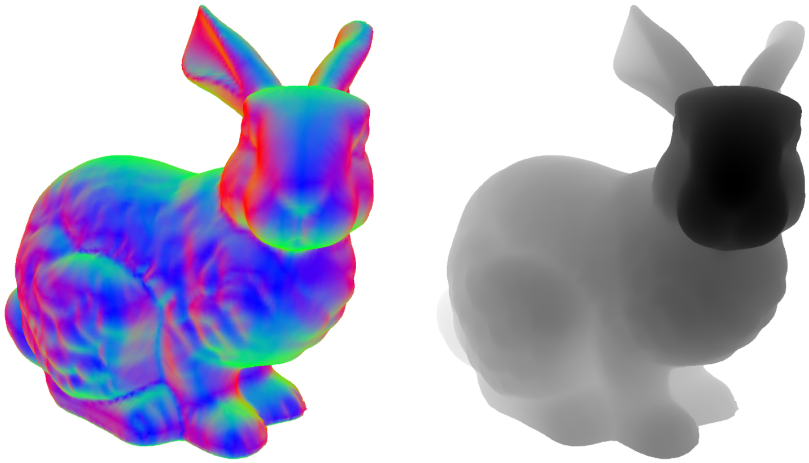

FIGURE 12 – Affichage d'un maillage (couleur donnée par les normales sans calcul d'illumnation) à gauche. Image de la carte de profondeur (ZBuffer) à droite.

5.3 Calculs d'illumination

L'illumination (shading) est calculée pour chaque sommet 3D en lui associant une couleur. Pour cela, le sommet doit être associé à des attributs qui lui sont propres :

- position spatiale
- couleur du matériaux
- normale de la surface en cette position

Ainsi que de paramètres extérieurs :

- position de la caméra
- position et paramètres de la source lumineuse

En sortie du calcul d'illumination, une nouvelle couleur est associée au sommet 3D. La coloration du triangle est finalement obtenue par interpolation des couleurs des sommets. On nomme ce procédé : *Gouraud Shading*.

Soit $\mathbf{p}$ la position du sommet actuel de couleur c, $\mathbf{n}$ la normale unitaire. $\mathbf{u}_L$ est le vecteur unitaire pointant de $\mathbf{p}$ vers la source de lumière et $\mathbf{s}$ son symétrique par rapport à la normale. $\mathbf{t}$ est le vecteur unitaire pointant de $\mathbf{p}$ vers la caméra. La couleur de la source lumineuse est donnée par c_L.

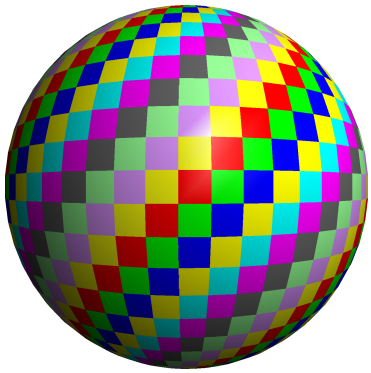

FIGURE 13 – Exemple de sphere. La triangulation utilisée est rendue apparente en modifiant la couleur associée à chaque patch de la sphère.

On distingue généralement 3 catégories d'illuminations
- L'illumination ambiante (éclairage homogène) : $I_a = a_a\, c$.
- L'illumination diffuse (effet de profondeur) : $I_d = a_d\ < \mathbf{n}, \mathbf{u}_L >^{k_d}\ c$.
- L'illumination spéculaire (effet de brillance) : $I_s = a_s\ < \mathbf{s}, \mathbf{t} >^{k_s}\ c_L$.

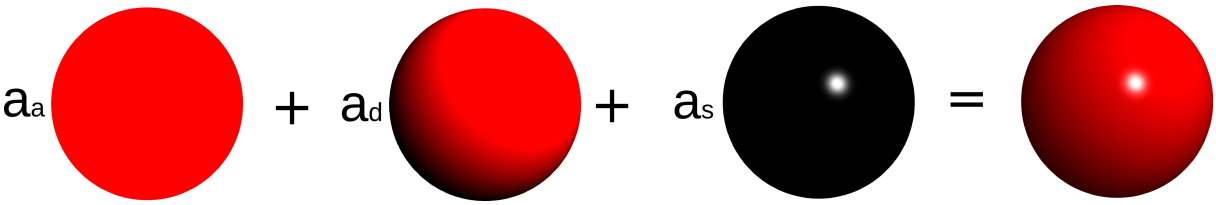

FIGURE 14 – Exemple d'illumination dans le cas d'une sphère. Les 3 illuminations : ambiante, diffuse et spéculaire sont montrés séparément. L'image finale étant obtenue comme somme pondérée de ces 3 illuminations.

Ce calcul d'illumination est généré par l'appel à la fonction

```
color apply_shading(const vec3& position,
                    const vec3& normale,
                    const color& couleur,
                    const scene& scene_courante,
                    const shading& shading_courant);
```

avec

```
struct shading
{
   double ambiant;
   double diffuse;
   double specular;
   double specular_exponent;
};
```

Question 33 *Implémentez le calcul d'illumination dans la fonction* `apply_shading`.

Question 34 *Complétez la méthode* `render` *de la classe* `render_engine` *de manière à réaliser le calcul d'illumination en chaque sommet.*

Question 35 *Vérifiez vos résultats sur différents maillages. Vérifiez l'influence de la position de la lumière et des différents paramètres d'illuminations.*

5.4 Notion de shaders

La gestion de projection des sommets un à un et du calcul d'illumination est contrôlable sur la carte graphique et possède le nom de *vertex shader*. La gestion de l'interpolation des couleurs au niveau des pixels est également contrôlable sur la carte graphique et possède le nom de *pixel shader*.

Ces calculs pour chaque sommet/pixels sont effectués en parallèles sur les différentes unitées de calculs des cartes graphiques.

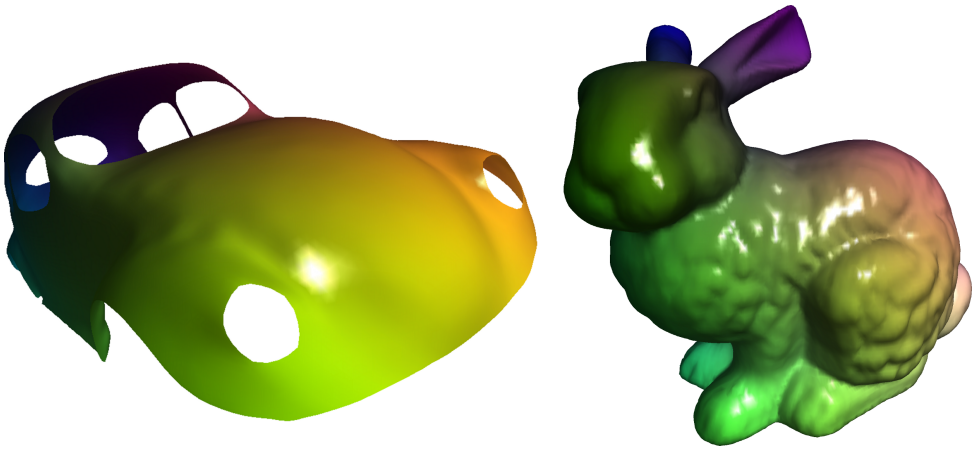

FIGURE 15 – Exemple de rendu projectif des maillages triangulés illuminés suivant la méthode de Gouraud.

6 Extensions possibles

6.1 Gestion des textures

Les sommets actuels contiennent des informations de positions, de couleur et de profondeur. Il est possible d'ajouter des coordonnées de textures. Leur gestion est similaire aux autres paramètres, ce sont des coordonnées bidimensionnelles (u, v) associés au sommet du maillage et interpolés linéairement.

Lors de l'affichage des fragments, la couleur de celui-ci est alors donnée par la couleur de l'image aux coordonnées de textures (u, v) interpolées.

Question 36 *Modifiez votre programme afin de gérer des textures.*

6.2 Librairie OpenGL

OpenGL est une librairie standard multiplateforme permettant de faire appel aux méthodes de rendu projectives qui sont exécutées sur votre carte graphique en temps réel.

Question 37 *Regardez et suivez des exemples de programmes simples d'affichages en OpenGL. Comparez à vos résultats.*