TP MSO Synthèse d'images: Rendu projectif
CPE

durée - 4h

Avril 2011

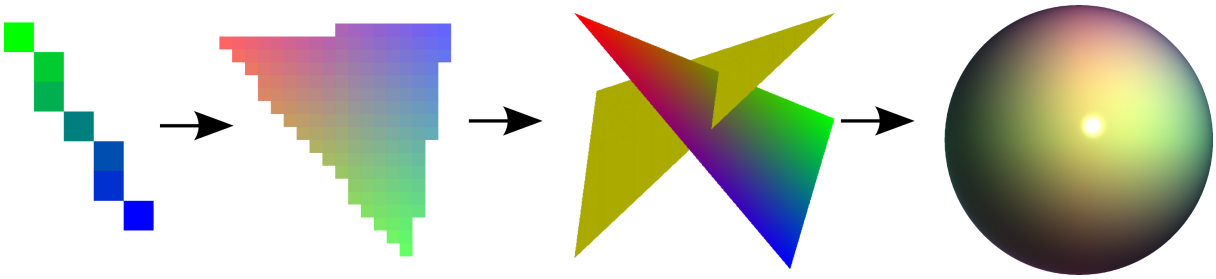


FIGURE 1 – Différentes étapes du rendu projectif : Tracé de segments discrets, tracé de triangles colorés, gestion de la profondeur, affichage d'un modèle 3D illuminé.

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. 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 barycentrique de couleurs.
- Enfin, nous procéderons à la projection et au calcul d'illumination dans le cas d'un triangle 3D de manière à modéliser sa visualisation sur une image 2D.

2 Prise en main de l'environnement

2.1 Programme main

La fonction *main* 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 500×500 pixels en mémoire.
- Remplissage de l'ensemble des pixels par la couleur blanche.
- Import d'un maillage triangulé complexe à partir d'un format d'échange asccii standard, calcul des normales et envoi de la géométrie dans le gestionnaire de rendu projectif (à compléter dans ce TP).

– 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.

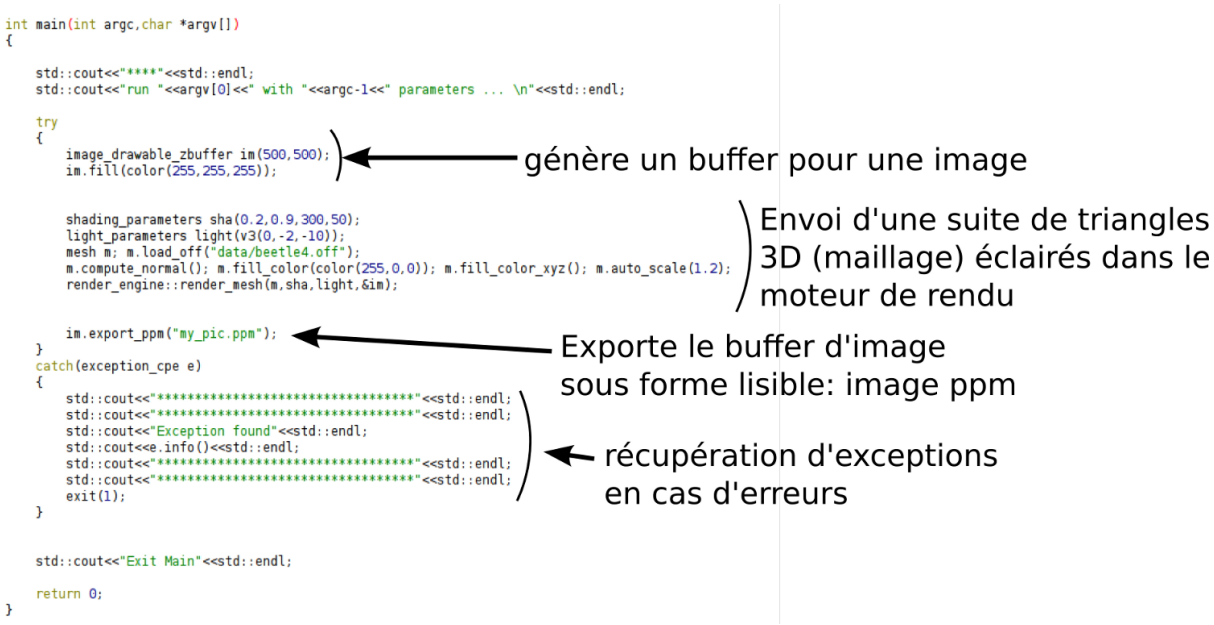


FIGURE 2 – Fonctions appelées depuis le main original.

Une fois que l'ensemble des fonctions sera complété, l'image de sortie doit représenter une vue du maillage 3D similaire à celui visualisé par une utilisation standard des API OpenGL/Direct3D (ex. fig. 11).

2.2 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.

2.2.1 Classe p2d

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

```
p2d u(5,4);     //u=(5,4)
u.x()=8;        //u=(8,4)
u=2*u-p2d(1,0) //u=(15,8)
```

2.2.2 Classe v3

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

```
v3 p(1.5,1.0,-2);          //p=(1.5,1.0,-2.0)
v3 o=p.dot(v3(1,1,0))*p; //o=<p,(1,1,0)> p
o+=p;                      //o=o+p
v3 e=o.normalized();       //e=o / ||o||
e.z()=4;                   //e=(e.x,e.y,4)
```

2.2.3 Classe v3shaded

Classe dérivée de v3. Elle permet de calculer l'illumination d'un sommet étant donné des paramètres d'éclairements.

2.2.4 Classe color

Classe de conteneur d'une couleur (r,g,b) où chaque canal est encodé sur un entier $\in [\![0, 255]\!]$. Notez que la méthode static *interpolate linear* implémente l'interpolation linéaire entre 2 couleurs sur des entiers.

```
color c(255,255,0) // c <- jaune
c.b()=255;         // c <- blanc
color bleu=(0,0,255);

//magenta = (1-0.4)*rouge + 0.4*bleu
color magenta=color::interpolate_linear(color(255,0,0),bleu,0.4);
```

2.2.5 Classe image

Classe de gestion d'une image (r,g,b). L'image est stockée en interne sous forme de vecteur concaténé de unsigned char $(r_0, g_0, b_0, r_1, \ldots, b_{N-1})$. La classe implémente l'initialisation, l'accès protégé aux données de couleurs et l'export d'une image dans un fichier *ppm*.

```
image pic(500); // créé une image 500x500
pic.fill(color(255,0,0)); // colore l'image en rouge

// colore pixel(10,15) en vert
pic.set_pixel(p2d(10,15),color(0,255,0));

//exporte l'image dans fichier mon_pimage.ppm
pic.export_ppm(``mon_image.ppm'');
```

2.2.6 Classe image drawable

Classe dérivée de image. Permet le tracé dans l'image de primitives lignes et de triangles (à compléter).

2.2.7 Classe image drawable zbuffer

Classe dérivée de image drawable. Les méthodes lignes et triangle sont surchargées pour tenir compte du Zbuffer qui doit être remis à jour lors de l'affichage de ces primitives.

2.2.8 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.

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.

On rappelle pour cela l'algorithme de tracé de ligne de Bresenham :

```
//source Wikipedia
function line(x0, y0, x1, y1)
  dx := abs(x1-x0)
  dy := abs(y1-y0)
  if x0 < x1 then sx := 1 else sx := -1
  if y0 < y1 then sy := 1 else sy := -1
  err := dx-dy

  loop
    setPixel(x0,y0)
    if x0 = x1 and y0 = y1 exit loop
    e2 := 2*err
    if e2 > -dy then
      err := err - dy
      x0 := x0 + sx
    end if
    if e2 <  dx then
      err := err + dx
      y0 := y0 + sy
    end if
  end loop
```

3.1 Segment de couleur constante

On va chercher à implémenter l'appel suivant dans la fonction *main*

```
image_drawable im(50);
im.line(p2d(10,10),p2d(45,32),color(255,255,255));
im.export_ppm(``mon_image.ppm'');
```

La fonction d'affiche de ligne est le suivant :

```
void image_drawable::line(const p2d& u0,const p2d& u1,const color& c)
{
  std::vector<pixel_interp> vec=algorithm::bresenham(u0,u1);
  for(unsigned int k=0,N=vec.size();k<N;++k)
  {
    const p2d& u=vec[k].u;
    if(check_position(u)==true)
      set_pixel(u,c);
  }
}
```

L'algorithme de parcours de pixels de Bresenham est appelé par l'appel de fonction statique *algorithm : :bresenham*. Comme montré en fig. 3 cet appel renvoi un vecteur contenant les positions des pixels du segment en *p2d*, ainsi que la position relative de ce pixel dans le segment (variable α variant entre 0 et 1). Ces deux variables sont stockées dans une structure spécialisée *pixel interp*.

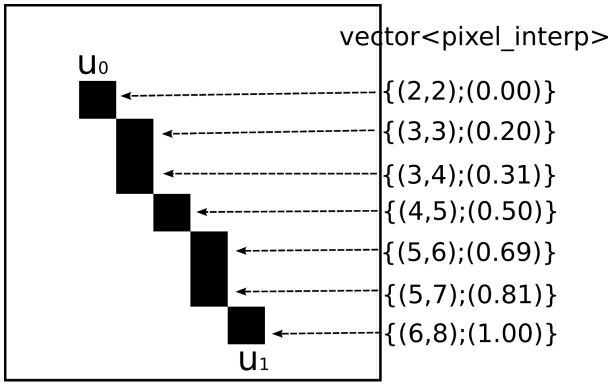


FIGURE 3 – Tracé de segment, et vecteur d'information associé rempli par l'appel à la méthode *bresenham*.

L'algorithme de Bresenham doit ainsi remplir ce vecteur.

Question 1 *Complétez la fonction* algorithm : :bresenham *suivant l'algorithme du même nom afin qu'il remplisse le vecteur convenablement pour un segment.*

Note : Cette manière de procéder n'est pas optimale en terme d'efficacité. Pour chaque ligne, on stocke les positions dans un vecteur dynamique retaillé au cours de l'exécution, et l'on réalise un calcul en précision flottante pour le calcul de la position relative.
Cette implémentation permet cependant d'être plus générique pour la suite du TP. Dans le cas d'une implémentation optimisée, les positions des pixels ne sont pas stockées dans un vecteur, mais la couleur est directement modifiée dans l'algorithme lui-même. De plus, la position relative en nombre flottant est ici inutile.

3.2 Interpolation linéaire de couleurs

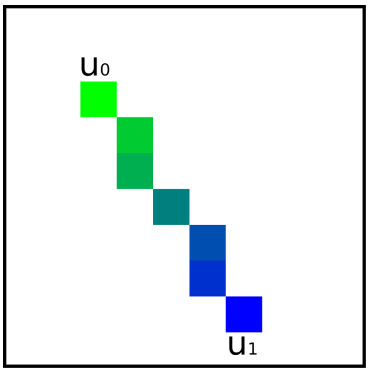


FIGURE 4 – Segment discret avec couleurs interpolées linéairement entre le vert (255,0,0) et le bleu (0,0,255).

On cherche maintenant à tracer un segment dont la couleur est linéairement interpolée entre les deux positions extrêmes. On rappel que de manière générale l'interpolation linéaire d'une

valeur à la position $\mathbf{x}$ entre deux positions extrêmes $\mathbf{x}_a$ et $\mathbf{x}_b$ contenant les valeurs respectives f_A et f_B est donné par

$$\begin{cases} f = (1-\alpha)f_A + \alpha f_B \\ \alpha = \frac{\|\mathbf{x}-\mathbf{x}_A\|}{\|\mathbf{x}_B-\mathbf{x}_A\|} \end{cases}.$$

Ayant stocké le paramètre α lors de l'appel à *algorithm : :bresenham*. Il est possible de réaliser le tracé d'un segment coloré où chaque pixel possède la couleur obtenue par interpolation comme le montre la fig. 4.

Question 2 *Complétez la méthode*

```
void image_drawable::line(const p2d& u1,
                          const p2d& u2,
                          const color& c1,
                          const color& c2)
```

Note : On pourra s'aider de l'appel à la méthode *color : :interpolate linear(c1,c2,alpha)*.

4 Partie II : Tracé de triangles

On s'intéresse désormais au tracé de triangles pleins. Dans un premier temps, on s'intéressera uniquement au remplissage des pixels intérieurs au triangle, puis dans un second temps à l'interpolation de la couleur.

4.1 Remplissage de triangle par couleur uniforme

L'appel à l'affichage d'un triangle se réalise par la méthode

```
void image_drawable::triangle(const p2d& u1,
                              const p2d& u2,
                              const p2d& u3,
                              const color& c);
```

Dans un premier temps, cette méthode va faire appel à l'algorithme de Bresenham sur chacun des trois côtés du triangle. Une fois les données des côtés discrets obtenues, le remplissage se réalise par l'algorithme dit *scanline*. La figure 5 résume les différentes étapes

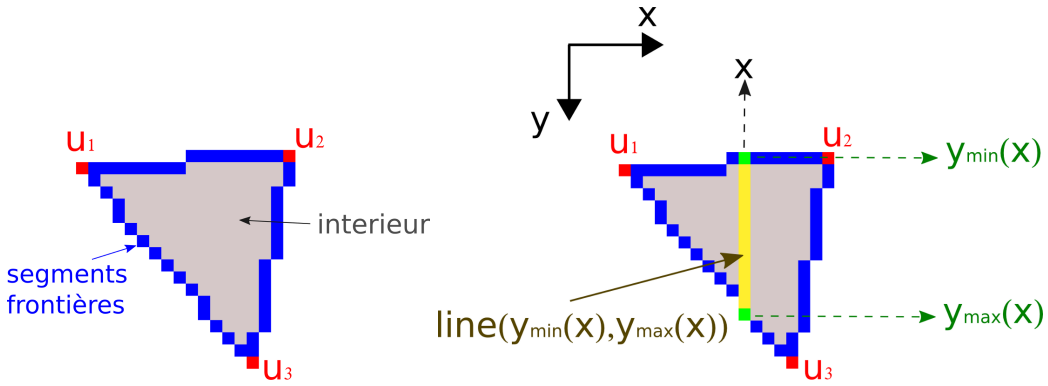


FIGURE 5 – Méthode de remplissage de triangles dit *scanline*. À x fixé, on stocke les valeurs de $y_{\min}$ et $y_{\max}$ correspondantes. Avant de tracer un segment vertical joignant ces deux positions.

L'algorithme *scanline* consiste à trouver pour chaque x (entre $x_{\min}$ et $x_{\max}$ des sommets du triangle) la coordonnée $y_{\min}(x)$ et $y_{\max}(x)$ correspondante. Le remplissage consiste alors à afficher à plusieurs reprises un tracé de lignes verticales suivant l'algorithme

```
Pour tout x entre [x_min,x_max]
  [y_min,y_max] <- extrema(x)
  line(x,y_min,y_max)
Fin pour
```

Il est donc nécessaire de remplir un buffer de taille $x_{\max} - x_{\min} + 1$ que l'on pourra stocker sous forme d'un *std : :vector* et contenant une position min et max. Afin d'être suffisamment générique pour permettre l'interpolation dans la suite du TP, on stockera pour x-donné également la position relative des pixels extrémaux dans leur segment respectif, ainsi que le numéro de leurs segments.

Dans ce TP, le buffer de données est implémenté sous forme de
std : :vector<std : :pair<scanline data,scanline data> >, avec

```
struct scanline_data
{
    p2d u;
    double alpha;
    int line_index;
};
```

La première occurrence du *std : :pair< >* contiendra les données correspondantes à $y_{\min}$, et la seconde à $y_{\max}$.

Un exemple de valeurs stockées dans un tel buffer est présenté en fig.6

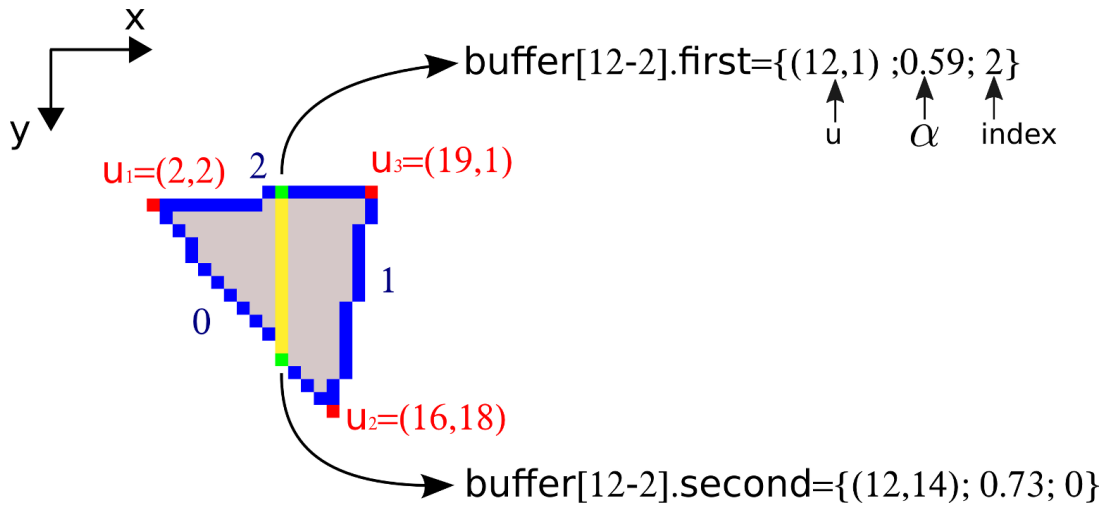


FIGURE 6 – Exemple de structure de données permettant de stocker les informations de position absolue (pixel) et relative (interpolation de couleurs, ...) pour chaque position extrême pour une coordonnée x-fixé.

Question 3 *Complétez la méthode* triangle *avec couleur uniforme de la classe* image drawable *de manière à suivre l'algorithme scanline présenté.*

Afin d'éviter de la duplication de code pour les 3 côtés du triangle, vous pourrez également compléter et appeler la méthode statique remplissant le buffer de données :

```
void algorithm::scanline_buffer(
        const std::vector<pixel_interp>& vec,
        const int& line_number,
        const int& x_min,
        std::vector<std::pair<scanline_data,scanline_data> >* buffer);
```

Note : L'algorithme *scanline* présenté ici fonctionne à x-donné (par colonne). On pourrait raisonner de manière symétrique suivant les lignes. Pour gagner en efficacité, on pourrait appeler un code d'affichage de ligne verticale optimisé en remplissant des blocs de mémoires contigües.

4.2 Remplissage de triangle avec interpolation linéaire des couleurs

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. 7 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. 7 droite.

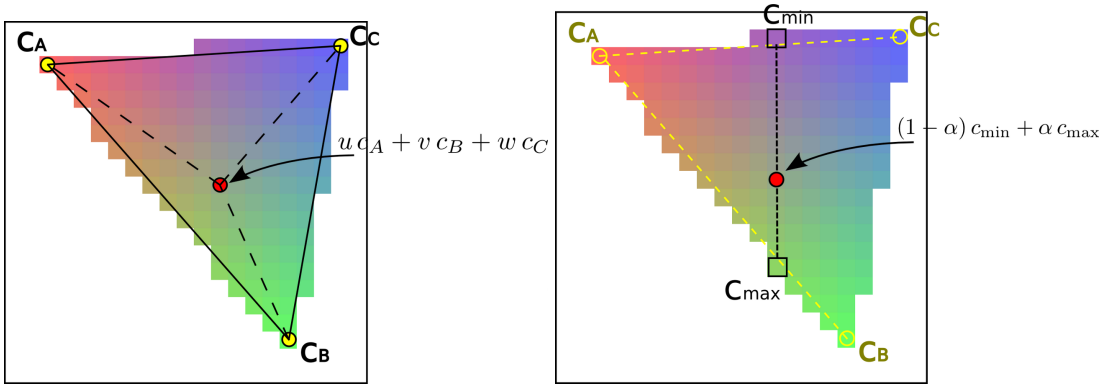


FIGURE 7 – Interpolation de couleurs dans un triangle. Gauche : interpolation par calculs de coordonnées barycentriques. Droite : Interpolation suivant deux calculs d'interpolation linéaires.

Question 4 *Complétez la méthode*

```
void image_drawable::triangle(
        const p2d& u1,
        const p2d& u2,
        const p2d& u3,
        const color& c1,
        const color& c2,
        const color& c3);
```

afin de réaliser une interpolation linéaire (de type barycentrique) des couleurs.

L'algorithme est similaire au cas du remplissage de triangle précédent, mais un traitement supplémentaire doit être réalisé pour l'interpolation des couleurs lors du tracé des lignes à x fixé.

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 pour cela l'illumination (shading) par la méthode de Gouraud. L'algorithme général du rendu projectif est le suivant :

```
T: triangle 3D avec normale + couleur

// partie "vertex"
- Calcul d'illumination pour chacun des 3 sommets.
- Projection de T dans le plan 2D

// partie "fragment"
- Remplissage du triangle dans le plan 2D
  - La couleur obtenue par l'illumination est interpolée linéairement
     entre les sommets
  - Les pixels ne sont affichés que si ils sont placés
     en avant de tout les autres (Zbuffer).
```

L'algorithme global de rendu de triangle 3D est géré par l'appel *render engine : :render triangle*.

Question 5 *Observez cette méthode, retrouvez les parties mentionnées et comprenez les différents appels de fonctions.*

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
  Fin si
Fin pour
```

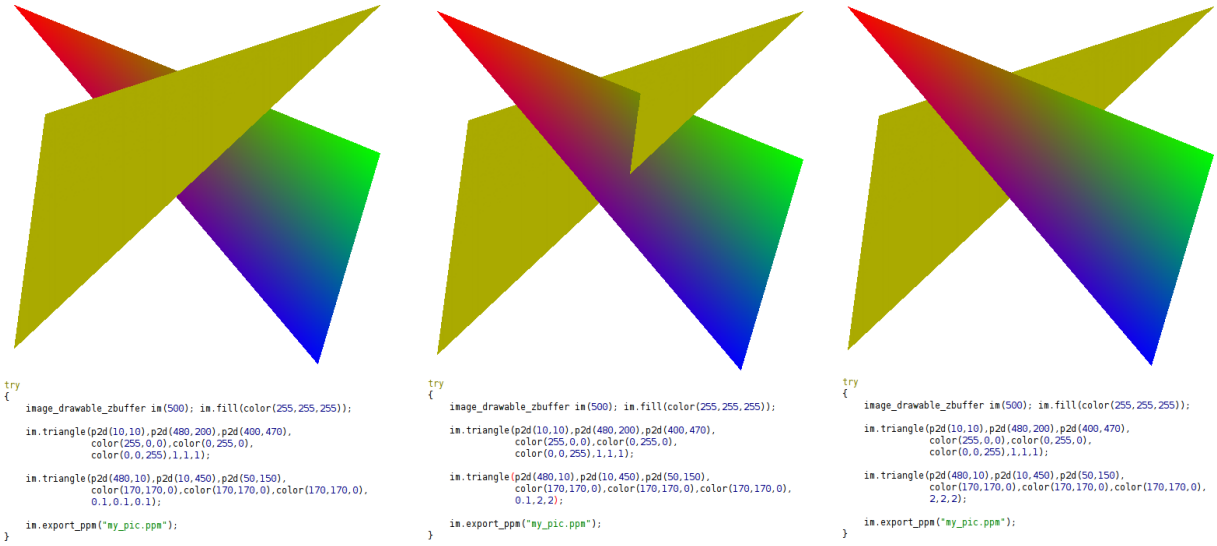
Notons que contrairement au buffer d'image, le ZBuffer est stocké en virgule flottante. Il n'est pas toujours possible de traiter avec une grande précision des objets proches, et lointains de manière unique.

La classe *image drawable zbuffer* reprend le même principe que la classe *image drawable* pour l'affichage de lignes et de triangles, mais ajoute la gestion du ZBuffer.

Question 6 *Complétez les appels à* image drawable zbuffer : :line *et* image drawable zbuffer : :triangle *afin que l'affichage des primitives ne soit plus dépendante de l'ordre des appels, mais de leur profondeur comme illustrée en fig. 8.*

On notera que la coordonnée z sera interpolée linéairement et barycentriquement tout comme la couleur.

Note : Lors du remplissage du triangle, chaque pixel se voit doté d'attributs supplémentaires (couleur, profondeur, ...) qui sont interpolés. On désigne cet élément de base par *fragment*. La suite d'instruction servant à la gestion du remplissage d'un triangle par ces *fragments* sur un GPU se désigne sous le terme *fragment program*. La gestion des *fragments* se réalisant en parallèle.



```
try
{
    image_drawable_zbuffer im(500); im.fill(color(255,255,255));

    im.triangle(p2d(10,10),p2d(480,200),p2d(400,470),
                color(255,0,0),color(0,255,0),
                color(0,0,255),1,1,1);

    im.triangle(p2d(480,10),p2d(10,450),p2d(50,150),
                color(170,170,0),color(170,170,0),color(170,170,0),
                0.1,0.1,0.1);

    im.export_ppm("my_pic.ppm");
}
```

```
try
{
    image_drawable_zbuffer im(500); im.fill(color(255,255,255));

    im.triangle(p2d(10,10),p2d(480,200),p2d(400,470),
                color(255,0,0),color(0,255,0),
                color(0,0,255),1,1,1);

    im.triangle(p2d(480,10),p2d(10,450),p2d(50,150),
                color(170,170,0),color(170,170,0),color(170,170,0),
                0.1,2,2);

    im.export_ppm("my_pic.ppm");
}
```

```
try
{
    image_drawable_zbuffer im(500); im.fill(color(255,255,255));

    im.triangle(p2d(10,10),p2d(480,200),p2d(400,470),
                color(255,0,0),color(0,255,0),
                color(0,0,255),1,1,1);

    im.triangle(p2d(480,10),p2d(10,450),p2d(50,150),
                color(170,170,0),color(170,170,0),color(170,170,0),
                2,2,2);

    im.export_ppm("my_pic.ppm");
}
```

FIGURE 8 – 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 positionnée à l'origine et orientée suivant l'axe z. L'approche la plus simple de la projection est celle du rendu orthographique. Dans ce cas, la perspective est absente, et la position sur la caméra est donnée par $u = (p_x, p_y)$.

Dans le cas d'une projection en perspective, on pourra par exemple considérer la transformation projective suivante : $u = (p_x/p_z, p_y/p_z)$ pour $p_z > 0$.

La valeur de p_z pouvant être stocké séparément afin d'être réutilisé pour le Zbuffer.

Question 7 *Complétez la méthode*

```
v3 render_engine::project(const v3& x)
```

tel que x soit projeté suivant une projection en perspective. La valeur x.z() sera stocké dans la coordonnée z du vecteur de retour.

5.3 Calculs d'illumination

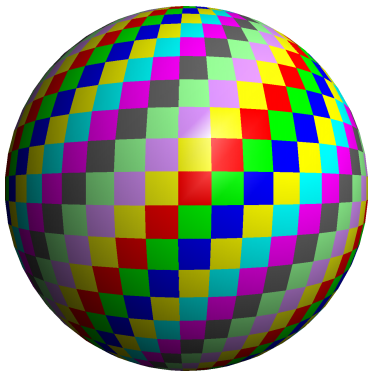


FIGURE 9 – Exemple de sphere. La triangulation utilisée est rendue apparente en modifiant la couleur associée à chaque patch de la sphère.

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.

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) : $I_s = a_s\ < \mathbf{s}, \mathbf{t} >^{k_s}\ c_L$.
Le calcul de l'illumination est généré par l'appel à la méthode

```
color v3_shaded::shading(
          const color& c,
          const v3& normal,
          const shading_parameters& shading,
          const light_parameters& light) const
```

avec

```
struct shading_parameters
{
   double ambiant;
   double diffuse;
   double specular;
   double specular_exponent;
};
struct light_parameters
{
   v3 light_position;
};
```

Question 8 *Implémentez le calcul d'illumination dans la méthode* v3 shaded : :shading.
A l'aide de l'appel render engine : :render triangle, *réalisez le rendu de triangles 3D. Testez vos résultats en générant une sphère telle qu'illustrée en fig. 9 et 10.*

Note : Similairement au *fragment program*, la suite d'instruction gérant la projection et le calcul de sommets de manière individuelle sur un GPU se désigne sous le terme *vertex program*. La gestion de chaque sommet se réalisant en parallèle.

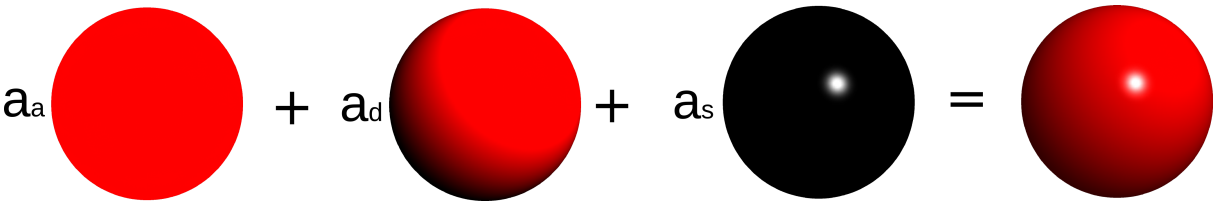


FIGURE 10 – Exemple d'illumination dans le cas d'une sphère. Les 3 illuminations : ambiante, diffuse et spéculaire sont montrés séparément. L'image finale étant obtenue comme somme pondérée de ces 3 illuminations.

5.4 Affichage d'un objet 3D

La classe *mesh* permet de gérer un maillage triangulé.

Question 9 *Appelez la suite d'instruction initialement fournie dans le* main *comme montré en fig.2, et visualiser des maillages 3D tel que ceux illustrés en fig. 11.*

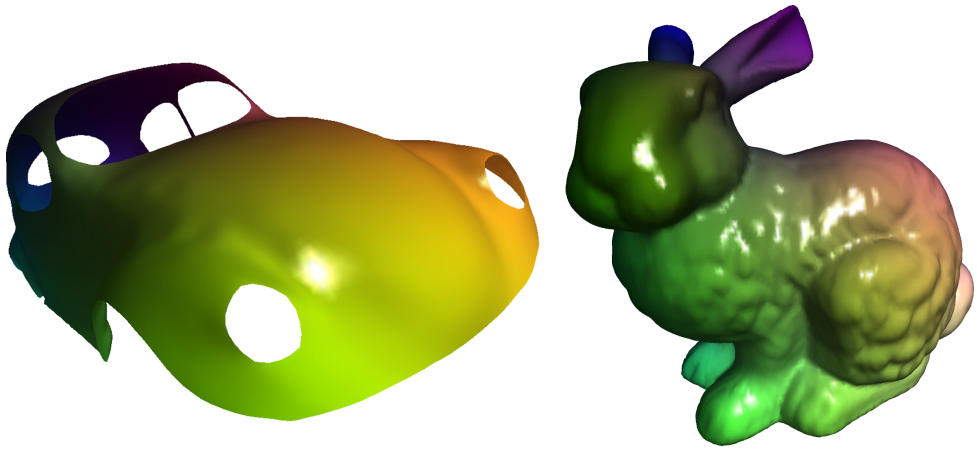


FIGURE 11 – Exemple de rendu projectif des maillages triangulés illuminés suivant la méthode de Gouraud.

6 Extensions possibles

6.1 Interpolation de couleurs générique

Augmentez la généricité de votre programme en rendant la méthode d'interpolation de couleur programmable. C'est-à-dire que l'on ne souhaite plus forcément utiliser constamment l'interpolation linéaire proposé, mais potentiellement d'autres procédures (ex. donner un effet de type *cartoon*, ...). On trouvera un exemple de rendu possible en fig. 12.

Plusieurs approches sont possibles :

- Duplication du code de tracé de lignes avec différente interpolation.
  Avantage : Trivial.
  Inconvenient : Rend le code ingérable : duplication de code+mélange tracé de lignes avec interpolation de couleurs.
- Envoi d'un pointeur de fonction pour l'interpolation des couleurs.
  Avantage : Générique, pas de duplication.
  Inconvenient : Plus lent qu'un appel direct.
- Passage d'une classe d'interpolation en tant que template.
  Avantage : Très générique, exécution rapide (code généré à la compilation).
  Inconvenient : Temps de compilation allongé.

Question 10 *Implémentez une interpolation générique basée sur le passage de classes d'interpolations sous forme de template.*

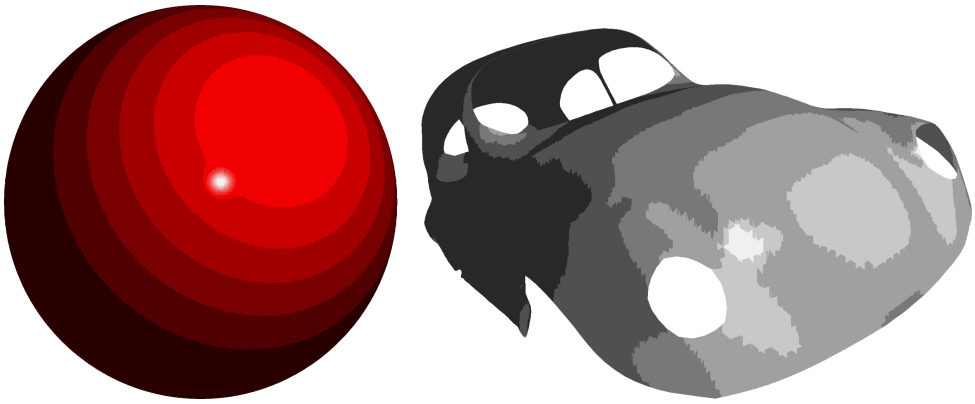


FIGURE 12 – Exemple de rendu utilisant d'autres méthodes d'illuminations/intepolation de couleurs.

Note : Exemple d'approche basée sur les templates

```
//classes d'interpolations possible:

class interpolation_linear
{
 public:
  color operator()(const color& c0,const color& c1,const double& alpha)
  {...}
};
class interpolation_2
{
 public:
  color operator()(const color& c0,const color& c1,const double& alpha)
  {if(alpha>0.5)return c1;return c0;}
};

//fonction de tracé de lignes générique:
template <typename INTERP_COLOR>
  void line(const p2d& u1,const p2d& u2,const color& c1,const color& c2)
  {
    ...
    INTERP_COLOR interpolate;
    ...
    color c=interpolate(c1,c2,alpha);
    set_pixel(u,c);
    ...
  }

//appel de la méthode de tracé de ligne générique
int main()
{
  image_drawable pic(50);
  im.line<interpolation_linear>(p2d(1,1),p2d(6,6),...);
  im.line<interpolation_2> (p2d(4,1),p2d(9,6),...);
}
```

6.2 Gestion des textures

Les fragments du code actuel 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 11 *Modifiez votre programme afin de gérer des textures.*

6.3 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 12 *Regardez et suivez des exemples de programmes simples d'affichages en OpenGL. Comparez à vos résultats.*