Rendu projectif
CPE
TP original de Damien Rohmer

5ETI IMI

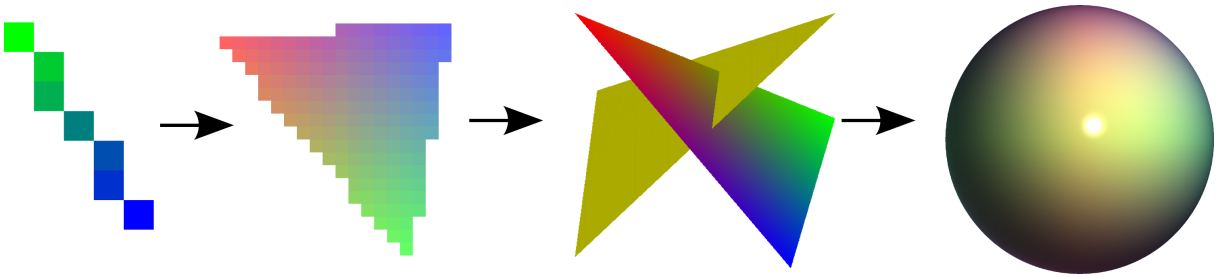

Figure 1: Différentes étapes du rendu projectif: Tracé de segments discrets, tracé de triangles colorés, gestion de la profondeur, affichage d'un modèle 3D illuminé.

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 (type OpenGL/Direct3D), mais sera ici codé par un programme exécuté par le 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 Compilation

Question 1 *Compilez le projet et assurez-vous que celui-ci s'exécute. Assurez-vous que vous puissiez l'éditer depuis l'éditeur de votre choix (QtCreator ou autre).*

2.2 Les différents répertoires des sources

Les répertoires contiennent différentes bibliothèques qui vous sont fournies (soient complètes, soient à compléter au fur et à mesure du TP).

- `lib/` contient des bibliothèques que vous avez déjà utilisées en 4ETI. Elles donnent principalement accès à des classes de vecteurs à 2, 3, ou 4 dimensions, à des matrices, ainsi qu'à une classe de maillage. L'ensemble des fichiers contenus dans `lib/` sont déjà complets.
- `image/` contient les fichiers de gestions et de conteneurs d'images. La librairie propose la gestion de couleur (r,g,b), de conteneur d'images, d'un ZBuffer, de textures. Le fichier `drawer` permet la manipulation d'images.
- `discrete/` contient les algorithmes et structures spécifiques à la géométrie discrète. En particulier, l'algorithme de Bresenham, le conteneur de position d'une ligne discrète, et l'algorithme de scanline pour le tracé de triangle sur une image.
- `render_engine/` contient le code permettant les appels de haut niveau spécifique au rendu projectif. On réalisera ici la partie projective (le passage de 3D vers 2D) ainsi que le calcul d'illumination avant de demander l'affichage d'un triangle à partir des autres bibliothèques. Cette partie pourra être mise en parallèle avec l'utilisation d'OpenGL.
- `local/` contient la fonction `main()` et les appels de fonctions permettant de définir une scène spécifique.

2.3 Programme main

Question 2 *Observez le code contenu dans le programme main. Assurez-vous que vous voyiez l'image écrite sur votre disque dur lors de l'exécution de votre programme. N'oubliez pas que vous pouvez configurer l'emplacement du lancement de votre exécutable dans vos IDE tels que QtCreator.*

Question 3 *Ouvrez l'image exportée à l'aide du logiciel Gimp. Zoomez de manière à distinguer clairement les pixels.*

Notez que lorsque vous visualiser l'image à l'aide des outils par défaut de votre environnement, il est possible que l'image apparaisse floue lorsque vous zoomer. Cela provient de l'utilisation d'interpolation. Cet effet est généralement paramétrable et annulable dans la configuration de ces outils de visualisation.

Question 4 *Ajoutez le code permettant d'obtenir une ligne verticale à votre image telle que celle-ci soit un dégradé du noir au blanc.*

Question 5 *Ouvrez l'image à l'aide d'un éditeur de texte. Observez que le contenu du fichier est un texte ascii (vous avez déjà rencontré ce format préalablement).*

De part sont format ascii non compressé, le format d'image .ppm peut être aisément écrit ou lu sans avoir besoin d'outil externe. Cependant, les images au format .ppm sont particulièrement lourde sur le disque dur. Pour éviter de stocker des images de taille trop importante, en particulier pour vos rapports, il est important de les convertir dans des formats compressés avant leurs inclusions.

Question 6 *Convertissez l'image au format .png (compression non destructive) et au format .jpg (compression destructive) à l'aide*

- *De l'outil ImageMagick en ligne de commande en tapant* `convert mon_image.ppm mon_image.png`.
- *De Gimp, grâce au menu d'export.*

Notez l'intérêt de l'outil en ligne de commande pouvant s'automatiser efficacement à l'aide de scripts.

Question 7 *Testez l'utilisation des exceptions en ajoutant la ligne suivante dans la fonction* `main()`
`std::cout<<im(91,10)<<std::endl;`

Observez le message d'erreur obtenue vous aidant à déterminer la cause du problème. Notez qu'un fichier `backtrace_log.txt` *est créé, et que celui-ci contient la trace d'exécution permettant de savoir à quelle suite d'appels correspond le lancement de l'exception. Il s'agira de messages permettant de vous aider à debuger votre programme (n'oubliez pas l'utilisation des debuggers: gdb, kdbg, valgrind, etc).*

Question 8 *Observez la classe d'image. De quelle classe dérive une* `image` *? Comment sont stockés en interne les couleurs d'une image?*

3 Tracé de segment

Dans un premier temps, nous nous intéressons au cas du tracé d'un segment discret de couleur uniforme (constante).

Étant donné deux positions discrètes 2D $(x_0, y_0), (x_1, y_1)$ extrêmes, nous cherchons à calculer toutes les positions discrètes intermédiaires définissant ce segment sur l'image. Les positions intermédiaires reliant ces deux points seront stockées dans la structure `discrete/line_discrete`. Cette structure peut être vue comme un vecteur contenant des positions entières.

Question 9 *Testez l'utilisation de la structure* `line_discrete` *dans votre fonction* `main` *à l'aide du code suivant.*

```
line_discrete line;          // creation d'une ligne discrete
line.push_back(ivec2(4,5)); // ajout d'une position
line.push_back({5,6});       // autre methode d'ajout
ivec2 const p(6,7);
line.push_back(p);           // autre methode d'ajout
std::cout<<line<<std::endl; // affichage du contenu de la ligne

//premiere methode de parcours de la ligne
for(int k=0;k<line.size();++k)
    std::cout<<line[k]<<" ";
std::cout<<std::endl;

//deuxieme methode de parcours de la ligne
for(ivec2 const& p : line)
    std::cout<<p<<" ";
std::cout<<std::endl;
```

3.1 Algorithme de Bresenham

L'objectif de cette partie consiste à remplir la structure `line_discrete` par des positions discrètes représentatives d'un segment discret comme illustré en figure 2, puis à utiliser cette structure pour dessiner un segment sur une image.

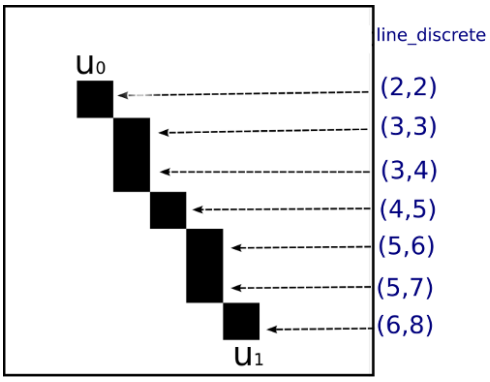

Figure 2: Segment discret et vecteur d'informations associé.

Nous allons utiliser pour cela l'algorithme de Bresenham. Afin de simplifier la tâche, nous allons coder l'algorithme dans le premier octant puis déduire les autres types de lignes par symétrie.

3.1.1 Premier octant

Question 10 *Complétez tout d'abord le code de la fonction* `bresenham_first_octant` *suivant l'algorithme de Bresenham.*

Question 11 *Vérifiez que l'appel suivant s'exécute convenablement et indique un résultat plausible (appel dans la fonction* `main()`*). Sur quels critères vous basez-vous ?*

```
line_discrete line;
bresenham({5,5},{12,9},line);
std::cout<<line<<std::endl;
```

3.2 Affichage d'un segment sur une image

La fonction

```
void draw_line(image& im,ivec2 const& p0,ivec2 const& p1,
                                          color const& c)
```

du fichier `drawer.cpp` a pour but de tracer un segment de couleur uniforme sur l'image passée en paramètre entre les positions `p0` et `p1`.

Question 12 *Complétez cette fonction, et testez celle-ci entre les points* `(5,5)`*, et* `(12,9)`*. Vérifiez visuellement sur l'image que le segment est affiché correctement.*

3.3 Autres quadrants

Pour l'instant, seuls les segments du premier quadrant peuvent être affichées.

Nous allons compléter le code de manière à se ramener par symétrie au premier octant lorsque cela est nécessaire. Pour cela, nous disposons des fonctions d'aides situées dans les fichiers `bresenham_octant`.

Question 13 *Que se passe t-il si l'on tente de tracer un segment d'un autre octant ?*

Question 14 *Modifiez le code de la fonction* `bresenham` *de manière à réaliser des symétries sur* `p0` *et* `p1` *tel que donné sur l'exemple suivant.*

```
void bresenham(ivec2 const& p0,ivec2 const& p1,line_discrete& line)
{
    // Compute the corresponding p0,p1 in the symmetrical first octant
    int const octant = octant_number(p0,p1);

    ivec2 const p0_octant = symmetry_octant(p0,octant);
    ivec2 const p1_octant = symmetry_octant(p1,octant);

    // Compute Brensehman in the first octant
    bresenham_first_octant(p0_octant,p1_octant,line,octant);

    // Sanity check ...
    // (laisser le code deja present)
}
```

Question 15 *Dans la fonction* `bresenham_first_octant()`*, utilisez l'appel à* `symmetry_octant_inverse()` *lorsque vous ajouter le pixel courant afin de convertir une position du premier octant vers l'octant d'origine au moment de l'ajout dans la structure* `line_discrete`*.*

Question 16 *Quelle démarche utilisez-vous pour vérifier que votre code fonctionne correctement ?*

Note: L'utilisation d'une structure stockant les positions dans un vecteur redimensionné de manière dynamique est coûteuse. Il serait plus efficace de compléter directement la couleur du pixel de l'image dans l'algorithme de Bresenham lui même. Cependant, cette approche permet d'être très générique en permettant la mise en place de couleurs et textures par la suite tout étant simple d'utilisation au niveau du code.

3.4 Interpolation de couleurs

Nous cherchons désormais à tracer une ligne entre un point p_0 et p_1 telle que la couleur varie linéairement entre c_0 et c_1.

Pour cela, nous allons utiliser une structure intermédiaire: `line_interpolation_parameter` qui vient calculer le poids correspondant à chaque pixel du segment. C'est à dire que cette structure va stocker le paramètre α_k tel que la couleur courante c_k à l'index k puisse être calculée sous la forme

$$c_k = (1 - \alpha_k)\, c_0 + \alpha_k\, c_1 \,.$$

Le paramètre α_k est calculé en tant que distance du point courant p_k par rapport à la longueur totale du segment. C'est à dire que

$$\alpha_k = \frac{\|p_k - p_0\|}{\|p_1 - p_0\|} \,.$$

Le vecteur contenant les paramètres α_k sont calculé lors de la construction de la structure en passant en paramètre un `line_discrete`

Testez le code suivant

```
line_discrete line;
bresenham({5,5},{12,9},line);
std::cout<<line<<std::endl;

line_interpolation_parameter interpolation(line);
std::cout<<interpolation<<std::endl;
```

Question 17 *Entre quelles valeurs varient les paramètres d'interpolation ? Quelle valeur prend le paramètre d'interpolation lorsque la ligne discrète ne contient qu'un seul point ?*

Complétez la fonction suivante

```
void draw_line(image& im,ivec2 const& p0,ivec2 const& p1,
                         color const& c0,color const& c1)
```

Question 18 *Assurez vous que le résultat soit correcte. Quelle est votre démarche?*

Note: Encore une fois, l'utilisation d'un tableau de taille dynamique stockant des poids d'interpolation n'est pas la solution la plus efficace pour calculer l'interpolation de couleur. Elle permettra cependant par la suite de calculer de manière simple et unifiée l'interpolation de profondeurs et coordonnées de textures par exemple.

4 Affichage de triangle

Question 19 *Complétez la fonction*

```
void draw_triangle_wireframe(image& im,
                             ivec2 const& p0,
                             ivec2 const& p1,
                             ivec2 const& p2,
                             color const& c);
```

affichant les 3 cotés d'un triangle à l'aide d'une couleur unique.

4.1 Algorithme scanline

L'objectif de cette partie consiste à remplir le triangle avec une couleur variable.

Dans un premier temps, considérons un triangle de couleur uniforme. L'algorithme de scanline consiste à parcourir les 3 cotés du triangle, et, pour chaque ligne rencontrée (paramètre y) stocker la position la plus à gauche et la plus à droite (paramètre x minimal et maximal) sur le coté du triangle. Une fois cette étape réalisée, chaque ligne appartenant au triangle est parcourue et un segment horizontal est tracé entre le point le plus à gauche et le point le plus à droite comme illustré en figure 3.

Considérons ensuite que l'on souhaite interpoler une valeur associée à chaque sommet du triangle (telle qu'une couleur par exemple). Dans ce cas, en plus de stocker les positions les plus à gauche/droite de chaque ligne, nous stockons également la valeur interpolée linéairement le long du segment pour chaque pixel du côté correspondant. Lors de l'affichage des droites horizontales, les valeurs interpolées de chaque côté seront alors considérées comme les valeurs extrêmes associées au segment courant.

L'algorithme de scanline est déjà codé, et la structure `triangle_scanline` permet de sauvegarder les sommets les plus à gauches/droites du triangle ainsi que les valeurs interpolées pour des types génériques de paramètres (encodée en tant que type template).

Considérons le cas du code suivant

```
draw_triangle_wireframe(im , {15,12},{2,17},{6,3} , {0,0,0});
auto scanline = triangle_scanline_factory({15,12},{2,17},{6,3} ,
                                          0.0f,1.0f,2.0f);
std::cout<<scanline<<std::endl;
```

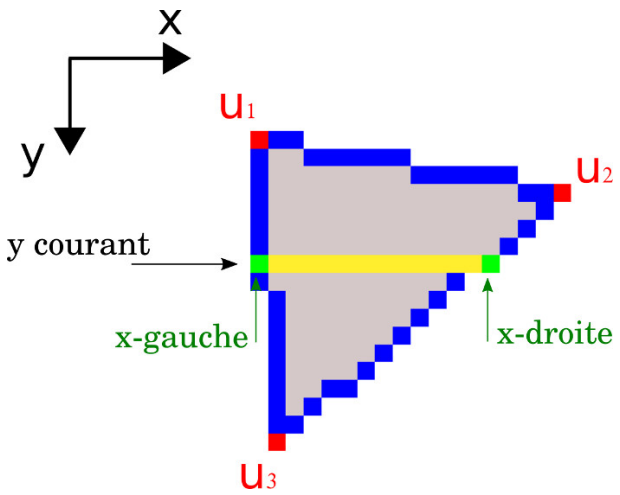

Figure 3: Remplissage de triangle par la méthode dite de *scanline*. Pour toutes les valeurs de y, x-gauche et x-droite sont stockés. Un segment horizontal est ensuite tracé entre toutes les valeurs (y,x-gauche) et (y-x-droite).

Question 20 *Après avoir observé la figure correspondant aux cotés du triangle sur l'image obtenue, analysez l'affichage en ligne de commande de scanline. Retrouvez les positions gauches et droites des segments horizontaux. À quoi correspond les valeurs situées après les positions. Sur quelles positions les valeurs sont elles exactement de 0, 1, et 2. Expliquez pourquoi.*

Considérons désormais le code pouvant correspondre à un triangle possédant des couleurs à ces trois sommets.

```
draw_triangle_wireframe(im , {15,12},{2,17},{6,3} , {0,0,0});
auto scanline = triangle_scanline_factory({15,12},{2,17},{6,3} ,
                                color(1,0,0),color(0,1,0),color(0,0,1));
std::cout<<scanline<<std::endl;
```

Question 21 *Interprétez désormais l'affichage obtenue.*

Question 22 *Quel est le type de la classe* `scanline` *pour chacun des deux exemples précédents ? Quel intérêt voyez-vous à l'utilisation de la fonction* `triangle_scanline_factory()` *par rapport à l'appel direct du constructeur de la classe associée ?*

Notez les différentes méthodes de parcours de la classe scanline permettant d'accéder à chacun des attributs de la structure.

```
//en utilisant un parcours generique de l'ensemble du conteneur
for(auto const& value : scanline)
{
    int const x = value.first;
    auto const& left = value.second.left;
    auto const& right = value.second.right;

    ivec2 const& p_left = left.coordinate;
    ivec2 const& p_right = right.coordinate;

    color const& param_left = left.parameter;
```

```
    color const& param_right = right.parameter;

}

//en utilisant les iterateurs
auto it = scanline.begin();
auto it_end = scanline.end();
for( ; it!=it_end ; ++it)
{
    int const x = it->first;
    auto const& left = it->second.left;
    auto const& right = it->second.right;

    ivec2 const& p_left = left.coordinate;
    //... idem ...

}
```

Question 23 *Assurez vous que vous compreniez chaque étape de ces parcours. Quel est le type des variables* `left` *et* `right` *? Quel est le type de la variable* `it` *?*

4.2 Affichage de triangle

Question 24 *Utilisez la structure scanline pour définir la fonction suivante de tracé d'un triangle plein de couleur uniforme*

```
void draw_triangle(image& im,
                   ivec2 const& p0,ivec2 const& p1,ivec2 const& p2,
                   color const& c)
```

Question 25 *Utilisez la structure scanline pour définir la fonction suivante de tracé d'un triangle plein de couleur variable*

```
void draw_triangle(image& im,
                   ivec2 const& p0,ivec2 const& p1,ivec2 const& p2,
                   color const& c0,color const& c1,color const& c2)
```

5 Gestion de la profondeur

Pour l'instant, les triangles sont définis dans un espace 2D et sont affichés dans l'ordre de leur appels respectifs.
Nous allons ajouter un paramètre de profondeur aux objets que l'on affiche de manière à afficher au premier plan les objets qui sont les plus proches, alors que les plus éloignés seront cachés derrière les autres. Pour cela, nous utilisons une carte de profondeur appelée *zbuffer* (ou *depth buffer*) qui va stocker pour chaque pixel de l'image la profondeur de l'objet au premier plan. La couleur d'un pixel est mise à jour uniquement si sa profondeur associée est plus faible que la profondeur courante stockée dans le zbuffer.

Une classe `image_zbuffer` est déjà codée, elle peut être vue comme une image stockant en chaque pixel non pas des couleurs, mais une valeur (nombre flottant) caractéristique de la profondeur. Classiquement les profondeurs sont normalisées entre -1 et 1. -1 étant la profondeur minimum, alors que 1 est la profondeur maximum des objets pouvant être affichés.

Question 26 *À quelle valeur est initialisée par défaut les valeurs du zbuffer ? Pourquoi.*

Question 27 *Complétez la fonction suivante permettant d'afficher ou non un pixel de couleur en fonction de sa profondeur.*

```
void draw_point(image& im,image_zbuffer& zbuffer,
                ivec2 const& p,float z,color const& c)
```

Question 28 *Implémentez la fonction permettant d'afficher un segment ayant une couleur et une profondeur variable entre ses deux extrémités.*

Question 29 *Implémentez ensuite la fonction suivante permettant d'afficher une triangle ayant une couleur et une profondeur associée à chacun de ses sommets (voir figure 4).*

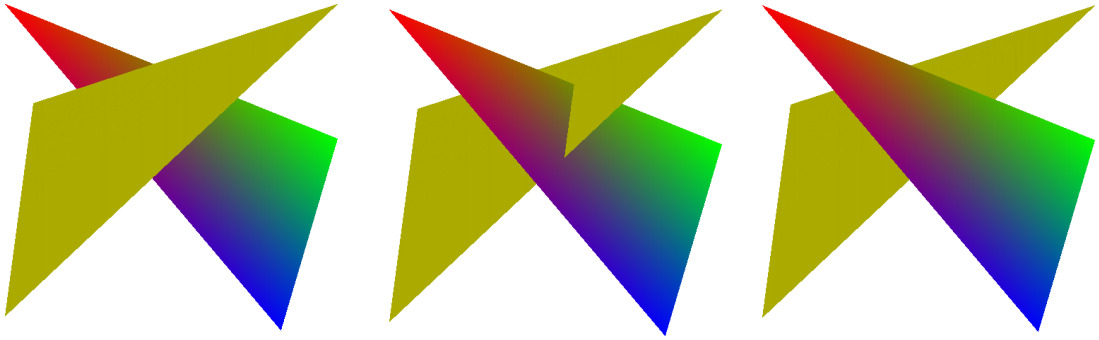

Figure 4: Trois triangles affichés avec différentes valeurs de profondeur. La gestion du zbuffer permet d'afficher uniquement les pixels les plus proches.

6 Projection

Nous allons désormais considérer un triangle possédant des coordonnées dans l'espace 3D. L'objectif est d'être capable d'afficher ce triangle dans l'espace 2D.

6.1 Rappel sur la transformation projective

Soit un point 3D exprimée dans l'espace projectif $p = (x, y, z, 1)$, et une matrice 4×4 de transformation projective P. Soit $p' = \mathrm{P}\, p$, et $p' = (x', y', z', w')$. Les coordonnées 2D de la projection du point p sont $u = (x'/w', y'/w')$, et la *profondeur* associée à ce point est z'/w'.

6.2 Fonctionnement du code

Le code correspondant au traitement de l'affichage des objets 3D est définit dans le fichier `render_engine`.

La première fonction que nous souhaitons implémenter est la suivante

```
void render(image& im,image_zbuffer& zbuffer,
            vec3 const& p0,vec3 const& p1,vec3 const& p2,
            color const& c0,color const& c1,color const& c2,
            vec3 const& n0,vec3 const& n1,vec3 const& n2,
            mat4 const& model,mat4 const& view,mat4 const& projection);
```

Cette fonction doit permettre l'affichage d'un triangle ayant pour coordonnées dans l'espace 3D (p_0, p_1, p_2). À chaque sommet est associé une couleur et une normale (pour le calcul de

l'illumination similairement à OpenGL). La fonction prend également en paramètre l'image sur laquelle le triangle est affiché et le zbuffer. Enfin, les matrices de model, view, et projection sont également passées en paramètres.

- La matrice `model` est la matrice de transformation associée à un objet. Elle permet par exemple de translater, ou tourner un objet en particulier.
- La matrice `view` est la matrice de transformation associée à la position et l'orientation de la caméra. Généralement, cette matrice est la même pour l'ensemble des objets de la scène.
- La matrice `projection` est la matrice de projection permettant de passer de l'espace 3D à un espace 2D en suivant une transformation projective permettant l'impression de perspective.

Dans notre cas, la transformation projective appliquée à un sommet du triangle est donnée par la matrice $\text{projection} \times \text{view} \times \text{model}$. Notez que l'on note modelview la matrice associée à $\text{view} \times \text{model}$. Notons également que dans l'espace de la caméra, les coordonnées des normales n' sont données par $n' = (\text{modelview}^{-1})^T n$, avec n, les coordonnées 4D des normales d'origine (on rappelle que dans l'espace projectif, la 4ème coordonnée d'un vecteur est 0).

6.3 Moteur de rendu pour un triangle

La fonction `render` vient dans un premier temps appeler la fonction `vertex_shader` qui sera à compléter. Cette fonction, similairement au cas d'OpenGL, à pour rôle de traiter chaque sommet de manière indépendante. Son rôle est de projeter les coordonnées 3D dans le plan 2D. Le vertex shader doit compléter 2 paramètres:

- La variable `p_proj` correspondant aux coordonnées `p` après projection. Notez que les coordonnées `p_proj` sont exprimées en coordonnées normalisées et non en nombre de pixels. Nous considérerons la même norme qu'en GLSL, à savoir que les coordonnées visibles sont comprises dans le cube unité $[-1,1] \times [-1,1], \times [-1,1]$. Le point 2D de coordonnées normalisées $(-1,-1)$ est en haut à gauche, alors que le point $(1,1)$ est en bas à droite. De plus, un point est visible si sa profondeur est comprise entre $[-1,1]$.
- La variable `c_shading` correspondant à la couleur du sommet après calcul de l'illumination dépendant potentiellement de la position de la lumière et des normales.

Dans un second temps, la fonction `render` convertit les coordonnées normalisées en nombre de pixels en fonction de la taille de l'image. Enfin, le triangle 2D correspondant est affiché.

Question 30 *Complétez la fonction* `vertex_shader` *de manière à réaliser la projection des sommets. Dans un premier temps, ne considérez pas le calcul d'illumination et affectez la couleur d'origine à chaque sommet. Testez votre programme en considérant une matrice identité pour* `model` *et* `view`*. Notez que pour qu'un triangle soit visible vis-à-vis de ce système de caméra, ses coordonnées en z doivent être négatives.*

On pourra par exemple considérez les matrices suivantes:

```
mat4 model;  //identitee
mat4 view;   //identitee
mat4 projection;
//matrice de projection angle de vue de 60 degres,
//  image de taille carree,
//  sommets visibles entre z=0.1 et z=20.
projection.set_projection_perspective(60*M_PI/180.0f,1.0f,0.1f,20.0f);
```

Notez que les triangles n'étant pas illuminés, il n'y a pas d'effet de profondeur au niveau de la couleur des triangles. Il est cependant possible de visualiser cette profondeur en exportant le zbuffer comme une image dans un fichier.

6.4 Illumination

On rappelle que l'illumination dite de Phong permet de calculer la couleur d'un sommet suivant 3 paramètres: l'illumination ambiante, diffuse, et spéculaire. Soit un sommet de coordonnées p, de couleur c et de normale unitaire n. On note u_L le vecteur unitaire pointant de p vers la source de lumière, et s son symétrique par rapport à la normale. t est le vecteur unitaire pointant de p vers la caméra.

Les coefficients d'illuminations sont données par les relations suivantes (voir figure 5)

- Illumination ambiante (éclairage homogène): $I_a = K_a$.
- Illumination diffuse (effet de profondeur): $I_d = K_d \ < n, u_L >_{[0,1]}$.
- Illumination spéculaire (effet de brillance): $I_s = K_s \ < s, t >^{e_s}_{[0,1]}$.

Avec $< a, b >_{[0,1]}$ signifiant le produit scalaire entre a et b tel que le résultat soit tronqué entre les valeurs $[0, 1]$ (clamp).
Au final, la couleur du sommet est donné par

$$c_{\text{shading}} = (I_a + I_d)\, c + I_s \,\text{blanc}\,.$$

On pourra considérer $K_a \simeq 0.2$, $K_d \simeq 0.8$, $K_s \simeq 0.6$, $e_s \simeq 128$

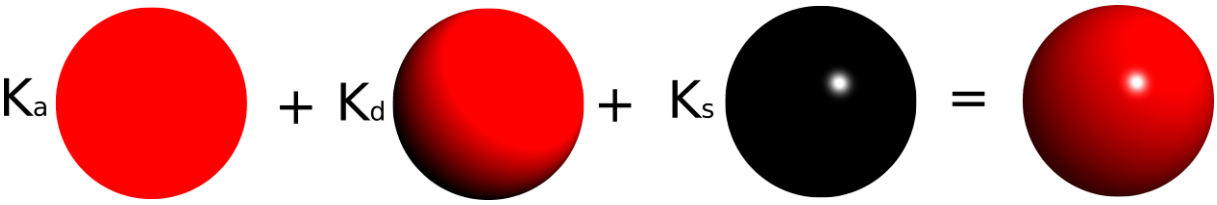

Figure 5: Exemple d'illumination dans le cas d'une sphère. Les 3 illuminations: ambiante, diffuse et spéculaire sont montrées séparément. L'image finale étant obtenue comme la somme pondérée de ces 3 termes.

Question 31 *Complétez la fonction* `vertex_shader` *afin de calculer l'illumination pour les trois sommets du triangle. Observez son effet sur le triangle en fonction de l'orientation des normales.*

7 Maillage

Il est possible de charger un maillage complet à partir d'un fichier. Le code suivant fournit un exemple de chargement de maillage

```
//chargement du fichier
mesh m = load_mesh_file("data/Frankie/Frankie.obj");

//applique potentiellement une rotation, translation
//              (scaling possible egalement)
m.transform_apply_rotation({0,1,0},-M_PI/4.0f);
m.transform_apply_translation({0,0,-1.5f});
```

```
m.fill_color({1,1,1}); //applique la couleur blanche a
                       //       l'ensemble des sommets
m.fill_normal();       //calcul automatique des normales
```

Question 32 *Complétez la fonction permettant d'afficher un maillage complet sur votre image (voir figure 6 gauche).*

```
void render(image& im,image_zbuffer& zbuffer,mesh const& m,
            mat4 const& model,mat4 const& view,mat4 const& projection)
```

7.1 Texture

Question 33 *Ajoutez la gestion des textures au moteur de rendu (voir figure 6 droite). On pourra se servir de la classe* `texture` *permettant de récupérer la valeur d'une couleur pour une coordonnée de texture* (u, v) *avec* u *et* v *appartenant à l'intervalle* $[0, 1]$. *Les coordonnées de textures pourront être stockées en tant que* `vec2`.
Prenez le temps de réfléchir aux différentes étapes à mettre en place pour cela.

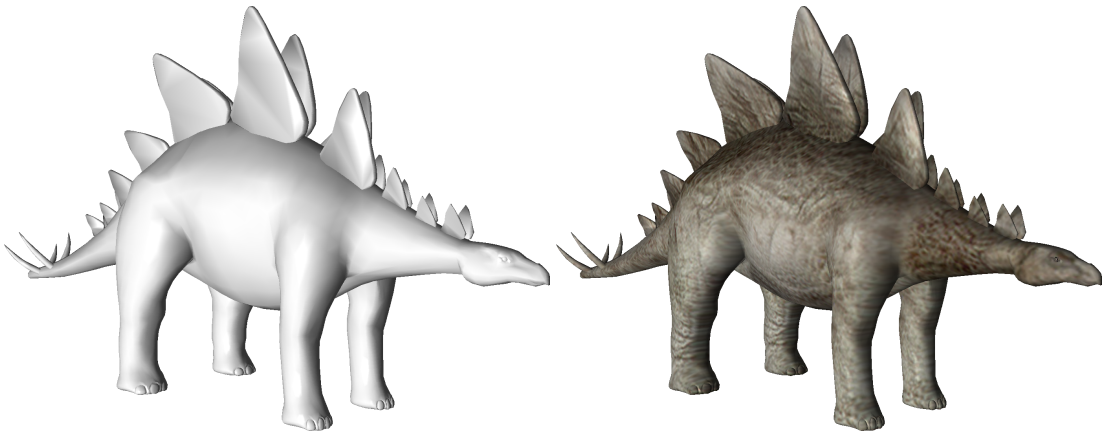

Figure 6: Gauche: Exemple d'affichage d'un maillage complet. Droite: Exemple d'affichage d'un maillage avec texture.

7.2 Extensions possibles

Question 34 *Étendez le code pour y ajouter des améliorations telles que: la gestion complète d'un fragment shader et de la mise en place d'une illumination de Phong calculée pour chaque fragment.*