SUPPORT DE COURS

Théorie et algorithmique d'OpenGl
PARTIE I

Avant propos

Ce support de cours n'est pas une introduction à OpenGl. Il s'agit de comprendre le mécanisme « interne » d'une API graphique 3D, comme l'est OpenGl. La pratique d'OpenGl se fera par le biais des séances de TP accompagnant ce cours. Il s'agit de présenter ici les principaux algorithmes ainsi que la structure logique d'une API graphique 3D, c'est-à-dire par exemple le tracé de triangles, l'élimination des parties cachées, la notion de frame buffer object, etc.

Contenu

1. Rappels : vecteurs, matrices, projection
2. Calculer une image par ordinateur : généralités
3. Représentation des objets géométriques : les primitives graphiques
4. Le pipeline graphique 2D/3D programmable
5. Modèles de caméra
6. Le tramage (rasterization) et le clipping
7. Elimination des parties cachées
8. Eclairage : modèle de Phong
9. Gestion de la transparence : alpha-blending
10. Buffers d'OpenGl, image et rasterization

1. Rappels: vecteurs, matrices, projection

Ceci n'est pas un cours de géométrie. Il s'agit simplement de faire quelques rappels concernant les outils de base nécessaires à la bonne compréhension d'une API graphique 3D.

Un POINT géométrique 3D est un triplet (x,y,z) de coordonnées indiquant une position dans un espace Euclidien par rapport à un **repère** donné.

Un VECTEUR 3D est également un triplet (vx,vy,vz) de coordonnées, indiquant une direction dans un espace par rapport à une **base** donnée. La taille (longueur ou élongation) du vecteur s'appelle la **norme** (elle est mesurée en mètres).

Deux points A et B définissent un vecteur V=AB. Les coordonnées de ce vecteur V sont obtenues par calcul de différences: Vx=Bx-Ax, Vy=By-Ay, Vz=Bz-Az

Si la norme du vecteur vaut 0, alors toutes les coordonnées sont nulles, et le vecteur est dit NUL. Si la norme d'un vecteur vaut 1, on dit que le vecteur est normalisé. Pour normaliser un vecteur il suffit de diviser ses coordonnées par sa norme.

Deux vecteurs non nuls V et W définissent un angle qui est mesuré en degrés (0 à 360) ou radians (0 à 2π). Si l'angle formé par les deux vecteurs vaut 0 ou 180 degrés, on dit que les vecteurs sont **colinéaires** (ils ont alignés). Si l'angle formé par les deux vecteurs vaut 90 degrés, on dit que les vecteurs sont **orthogonaux**.

Le **produit scalaire** (en anglais, *dot product*) entre deux vecteurs V et W est défini par:

$$V \cdot W = V_x W_x + V_y W_y + V_z W_z$$

Si le produit scalaire de deux vecteurs non nuls vaut 0 alors les vecteurs sont orthogonaux.

La **norme** (L2) d'un vecteur est égale à la longueur de ce vecteur. Elle est donnée par la racine carrée du produit scalaire de ce vecteur avec lui même:

$$\|V\| = \sqrt{V \cdot V} = \sqrt{V_x V_x + V_y V_y + V_z V_z}$$

Si le vecteur est défini par deux points A et B alors la norme du vecteur représente la distance Euclidienne entre A et B. Le produit scalaire d'un vecteur V avec un vecteur W est égal au cosinus de l'angle α entre ces deux vecteurs multipliées par leurs normes respectives.

$$V \cdot W = \|V\| \cdot \|W\| \cos(\alpha)$$

Un vecteur N orthogonal à deux vecteurs non nuls V et W peut être calculé par produit en croix ou produit vectoriel (en anglais, *cross product*): N = VxW

$$N_x = V_y W_z - V_z W_y$$
$$N_y = V_z W_x - V_x W_z$$
$$N_z = V_x W_y - V_y W_x$$

Deux vecteurs peuvent être additionnés ou soustraits. Il suffit d'additionner / soustraire les coordonnées. Le vecteur opposé de V=(x,y,z) est obtenu en prenant l'opposé des coordonnées: -V = (-x,-y,-z)

DROITE: une droite est définie par un point A et un vecteur V non nul, dit directeur. Son équation est donnée par: $A+\lambda V$, λ étant un réel.

PLAN: un plan est défini par un point A et deux vecteurs V,W, qui ne doivent pas être colinéaires, ni nuls. Son équation est donnée par: $A+\lambda_1 V+ \lambda_2 W$.

L'équation paramétrique du plan est: $ax+by+cz+d=0$
Dans cette équation (a,b,c) représentent les coordonnées d'un vecteur orthogonal à tout vecteur du plan. On dit que le vecteur (a,b,c) est la **normale** au plan.

Deux plans sont soit confondus, soit parallèles, soit ils ont une intersection qui est une droite. Soit un ensemble de points de l'espace Euclidien. Ces points sont dits **coplanaires**, s'il existe un plan unique les contenant tous. Trois points sont toujours coplanaires. En effet, trois points A,B,C définissent un unique plan: $A+\lambda_1 AB+ \lambda_2 AC$

L'équation paramétrique du plan donné par trois points A,B,C est obtenue en prenant: $(a,b,c) = N = AB \times AC$ et $d = - N \cdot A$

Un plan divise l'espace 3D en trois zones. Tout point de l'espace se trouve soit sur le plan, soit d'un coté du plan, soit de l'autre. Pour déterminer si un point P est d'un coté ou de l'autre, il suffit de choisir un point A sur le plan. Le signe du produit scalaire entre la normale au plan et le vecteur AP donne le coté. Avec l'équation paramétrique il suffit d'injecter les coordonnées de P dans l'équation à la place de x,y,z et le signe du résultat détermine le coté.

Une droite est soit parallèle à un plan, soit complètement incluse dans le plan, soit elle le traverse en un unique point (l'intersection). Le calcul de l'intersection entre une droite définie par $A+\lambda V$ et un plan défini par $ax+by+cz+d=0$, tel que N=(a,b,c) se fait comme suit. Soit P le point d'intersection: $P = A + u/v\ V$, avec $v = V \cdot N$ et $u = - A \cdot N - d$

REPERE: un repère dans l'espace Euclidien 3D est dit cartésien s'il est constitué d'une **origine** O (un point) et d'une **base** définie par trois vecteurs (I,J,K) tous orthogonaux entre eux. Nous appellerons le **repère du monde** le repère: O(0,0,0), I(1,0,0), J(0,1,0), K(0,0,1).

Un même vecteur V peut s'exprimer dans deux bases différentes. Il existe alors une **transformation linéaire** permettant d'exprimer les coordonnées du vecteur dans une base en fonction de l'autre base.

Toute transformation linéaire peut être définie par une matrice 3x3. Une matrice **M** est la données de trois vecteurs (U,V,W)

$$M = \begin{pmatrix} U.x & V.x & W.x \\ U.y & V.y & W.y \\ U.z & V.z & W.z \end{pmatrix}$$

On peut aussi noter sous la forme de double indices, les scalaires composants la matrice: M[i][j], i étant l'indice de la colonne et j l'indice de la ligne.
Deux matrices peuvent êtres additionnées. Il suffit d'additionner les trois vecteurs respectifs.
Deux matrices peuvent être multipliées: **R** = **A***__B__

tel que: $R[i][j]= \Sigma A[k][j] \cdot B[i][k]$, pour k de 1 à 3

Attention la multiplication de matrices n'est pas commutative.

La **matrice identité** est la matrice correspondant aux trois vecteurs du repère du monde. C'est une matrice diagonale, dont toutes les valeurs sur la diagonale sont égales à 1.

La **transposée** d'une matrice est: $M^T[i][j]=M[j][i]$.

Le **déterminant** d'une matrice **M**=(U,V,W) est le scalaire:

det(**M**) = U.x*(V.y*W.z-W.y*U.z) - U.y*(V.x*W.z-W.x*V.z) + U.z*(V.x*W.y-W.x*V.y)

Une matrice **M** multipliée par un vecteur B donne un vecteur A tel que: $A^T = B * \mathbf{M}$
avec $A = (B \cdot U, B \cdot V, B \cdot W)$

Une matrice **M** représente une transformation linéaire *t*. Le résultat de la multiplication d'un vecteur V avec **M**, a pour résultat V' le vecteur transformé, cad V'=*t*(V).

La multiplication de matrices correspond à une **composition de transformations**. Le sens de composition est inversé: la dernière matrice multipliée correspond à la première transformation appliquée. Il est donc possible de combiner plusieurs transformations et de les appliquer simultanément en effectuant qu'un seul produit vecteur par matrice.

Quelques transformations linéaires usuelles

Une transformation usuelle est le **changement de base**. Soit V un vecteur exprimé dans la base I,J,K du repère du monde. Les coordonnées V' de V dans un nouveau repère défini par: **R**=(Ei,Ej,Ek) sont données par une transformation linéaire définie comme **l'inverse** de **R**, càd: $V'=V*\mathbf{R}^{-1}$

L'inverse d'une matrice **A** est donnée par:

$$A^{-1} = \frac{1}{\det(A)} \begin{bmatrix} \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} & -\begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} & \begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix} \\ -\begin{vmatrix} a_{12} & a_{13} \\ a_{32} & a_{33} \end{vmatrix} & \begin{vmatrix} a_{11} & a_{13} \\ a_{31} & a_{33} \end{vmatrix} & -\begin{vmatrix} a_{11} & a_{12} \\ a_{31} & a_{32} \end{vmatrix} \\ \begin{vmatrix} a_{12} & a_{13} \\ a_{22} & a_{23} \end{vmatrix} & -\begin{vmatrix} a_{11} & a_{13} \\ a_{21} & a_{23} \end{vmatrix} & \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} \end{bmatrix}^t , avec \begin{vmatrix} r & s \\ t & u \end{vmatrix} = ru - st$$

Dans un espace euclidien à 3 dimensions, les matrices de rotations suivantes correspondent à des rotations autour des axes *x*, *y* et *z* (respectivement) :

$$Rx = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{pmatrix} \quad Ry = \begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{pmatrix} \quad Rz = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Coordonnées homogènes: certaines transformations ne sont pas linéaires en dimension trois. Par contre, elles le deviennent en dimension quatre, en passant en coordonnées homogènes. La translation est un exemple simple.

Les coordonnées d'un vecteur/point 3D (x, y, z) s'expriment en coordonnées homogènes (donc en dimension 4) sous la forme (x, y, z, 1). Inversement, un vecteur de coordonnées homogènes (x,y,z,w) s'exprime en coordonnées classiques 3D par (x/w,y/w,z/w). Notons que lorsque w vaut zéro le point n'existe pas dans l'espace 3D et se trouve alors à l'infini dans la direction (x,y,z).

En coordonnées homogènes, on peut définir une matrice 4x4 correspondant à une translation d'un point P(x,y,z) par un vecteur V=(Vx,Vy,Vz), tq P'(x',y',z')=P+V, de la manière suivante:

$$P'\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} x & y & z & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ Vx & Vy & Vz & 1 \end{pmatrix}$$

La multiplication donne x'=x+Vx, y'=y+Vy, z'=z+Vz et w'=1. En divisant par w', nous obtenons bien P' translaté de P, càd: P' = P+V

Nous pouvons à présent définir une unique matrice permettant d'effectuer un **changement de repère** pour un point P de l'espace 3D. Soit (O', Ei, Ej, Ek) le nouveau repère. Il suffit de translaté P vers la nouvelle origine du repère (translation par le vecteur -O'), matrice:

$$M_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -O'x & -O'y & -O'z & 1 \end{pmatrix}$$

puis d'effectuer un changement de base tel que décrit précédemment (en ajoutant une 4ème coordonnée homogène):

$$M_2 = \begin{pmatrix} (R)^{-1} & & & 0 \\ & & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad R = \begin{pmatrix} Ei.x & Ej.x & Ek.x \\ Ei.y & Ej.y & Ek.y \\ Ei.z & Ej.z & Ek.z \end{pmatrix}$$

Ces deux matrices 4x4 peuvent être multipliées entre elles: M_2*M_1 pour définir la transformation correspondant au changement de repère.

Une autre transformation usuelle non-linéaire en dimension 3 est **la projection perspective** sur un plan. Soit un plan défini par un point A et deux vecteurs orthogonaux normalisés U et V. Ce plan est défini par l'équation : $A+\lambda_1 U+ \lambda_2 V$. Soit N la normale au plan (produit vectoriel U par V) tels que (U, V, N) forment une base. Pour projeter P nous commençons par l'exprimer dans le repère (A,U,V,N) à l'aide d'une matrice de changement de repère **M**. Soit P'=P*__M__ le point exprimé dans ce nouveau repère que nous appellerons **repère de l'observateur**. La projection perspective est alors définie par la matrice suivante:

$$\begin{pmatrix} \frac{2n}{T_X} & 0 & 0 & 0 \\ 0 & \frac{2n}{T_Y} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -1 \\ 0 & 0 & -\frac{2 \cdot n \cdot f}{f-n} & 0 \end{pmatrix}$$

Les paramètres (T_X,T_Y) définissent la taille selon les axes respectifs U et V d'une *fenêtre de projection* sur le plan. Cette fenêtre rectangulaire de cotés Tx et Ty est centrée en A. *f* et *n* définissent les distances à deux plans parallèles au plan de projection, cad deux plans distants de f et n le long de l'axe N. Ces deux plans permettent une normalisation entre -1 et 1 de la coordonnée z des points P' projetés et compris entre ces deux plans.

Pour le point P=A-nN , nous avons P'=(0,0,-n). La projection de P' donne alors (0,0,-1);
Pour le point P=A-fN, nous avons P'=(0,0,-f). La projection de P' donne (0,0,1);

De façon générale, pour tout point

$$P = A + \frac{dX \cdot T_X}{2} U + \frac{dY \cdot T_Y}{2} V$$

dX et dY étant deux réels, nous avons P'=(T_XdX/2, T_YdY/2, -k). La projection donne (dX,dY,k'), avec k' compris entre -1 et 1 si k est entre -n et -f.

Notons que le ratio entre $max(T_X,T_Y)$ et la distance *n* permet de contrôler l'ouverture angulaire de la projection perspective.

2. Calculer une image par ordinateur : généralités

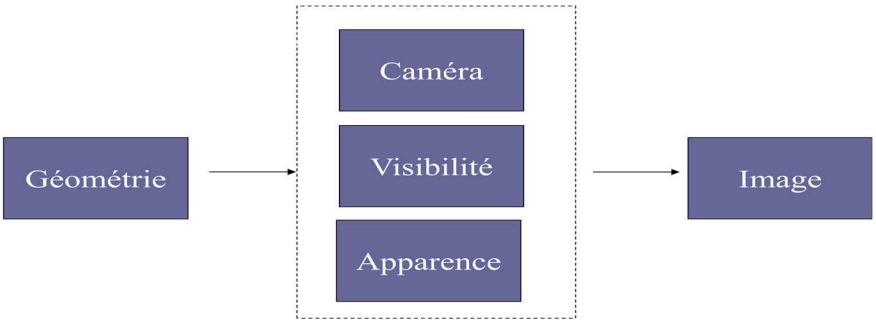
Pour calculer une image ou une séquence animée d'images de synthèse, un algorithme de « rendu » a besoin de :

- un modèle géométrique 3-D.
- un modèle d'animation (mouvement de camera, des objets, etc.), c'est une modélisation 4-D: 3-D+t.
- un modèle de caméra (projection) et de rendu graphique (stylisation graphique, éclairage, matériaux, etc.).

Par modèle géométrique, on entend un ensemble de points dans un espace affine Euclidien de dimension n. A ces points, en plus des coordonnées 3D, on associe souvent d'autres propriétés qui servent au rendu: couleur, brillance, etc. On distingue souvent les objets géométriques par leur dimension topologique :

- Points 3D (nuage de points, « systèmes de particules »)
- Courbes (ligne brisée, courbe spline)
- Surfaces (bord ou non d'un volume)
- Volumes « pleins » (cela permet le rendu de corps semi-transparents comme les nuages ou la fumée par exemple)

Schéma du calcul d'une image de synthèse :



Un algorithme de rendu se décompose en étapes : la définition d'un modèle de caméra, le calcul de la visibilité et le calcul de l'apparence.

Le modèle de caméra sert à transformer l'information 3D en information 2D (on parle de projection, par exemple perspective si l'on souhaite simuler un appareil photo ou la vision humaine).

La visibilité sert à déterminer quels objets sont dans le champ de vision : un mur masque par exemple les objets qui sont derrière lui.

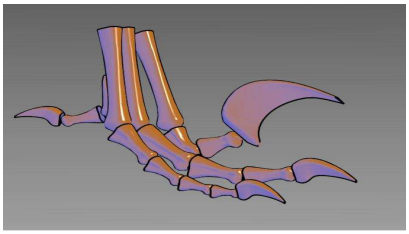
Le calcul d'apparence sert à reproduire l'interaction lumière-matière : Les objets réel ne sont visibles que parce qu'ils sont éclairés par des sources de lumière (soleil, ampoule, bougie, etc.). Une partie de la lumière est absorbée par les objets et transformée en chaleur ou en

électricité… Une autre partie est restituée et c'est ce qui donne à l'objet sa couleur (fonction de sa forme):

- un objet noir absorbe toute la lumière,
- un objet blanc la réémet presque entièrement;
- un objet est perçu comme étant rouge s'il absorbe toutes les longueurs d'onde sauf le rouge.

Dans certains cas, le calcul de l'apparence cherche à reproduire des phénomènes physiques optiques complexes: ombrage, transparence, transport de lumière, etc. pour augmenter l'expressivité des images (voire même créer des images similaires à des photographies). Dans d'autre cas, on cherchera à reproduire des effets de style : peinture expressionniste, tracé au crayon, etc. Trois exemples de « styles graphiques » sont illustrés ci-dessous :





3. Représentation des objets géométriques : primitives graphiques

La plupart des API graphiques 3D « temps réel » n'inclut pas de notion d'objet géométrique de « haut niveau ». Il n'y a donc pas de notion de cylindre, de sphère, etc. La représentation d'un objet géométrique se limite à une **liste de points** (3D ou 4D en coordonnées homogènes) qui peuvent correspondre aux sommets d'un polyèdre par exemple, selon la **connectivité** que l'on choisira. La connectivité des points se fait en définissant un type de **primitive**. Il n'existe généralement qu'un nombre très restreint de primitives prédéfinies.

Par exemple, pour modéliser un cube, composé de 6 faces, chacune composées de 4 sommets, on utilisera une liste de 24 points. On indiquera ensuite que ces points sont à considérés comme des quadruplets de sommets (définissant ainsi les faces du cubes). La liste de points est rangée dans un tableau contigüe de flottants (pour un cube en 3D, il y a 72 flottants, 3 flottants par point *x,y,z*). En appliquant une primitive de type QUAD, les séquences consécutives de 4 points (soit 12 flottants) sont interprétées comme des quadrilatères orientés (l'ordre des points est important). La structure en mémoire est :

float cube[24][3] ;

De même il est possible de définir un tétraèdre en utilisant une liste de 12 points (4 faces triangulaires), soit 36 flottants. La primitive associée est de type TRIANGLE.

A chaque point, on peut associer des **attributs** supplémentaires, comme la couleur, la normale, etc. Les attributs sont des scalaires, des vecteurs ou des matrices rangés dans des tableaux contiguës et dont la taille dépend de la taille de l'attribut et du nombre de points (il faut nécessairement une valeur d'attribut par point / sommet). Pour associer une couleur RVB à chaque sommet du cube :

unsigned char colcube[24][3] ;

Primitives graphiques usuelles

Les **seules** primitives graphiques proposées par OpenGl sont les suivantes : points, lignes, triangles et quads. Les points de la liste sont assemblés en fonction du choix de la primitive.

Points

Pour ce type de primitive, chaque point de la liste est complètement isolé. Les points sont déconnectés les uns des autres. On parle de ***systèmes de particules ou de splats*** si le sommet dispose d'un attribut supplémentaire de type vecteur « normal ». La taille du point affiché peut être contrôlée. Elle est donnée en nombre / fraction de pixels.

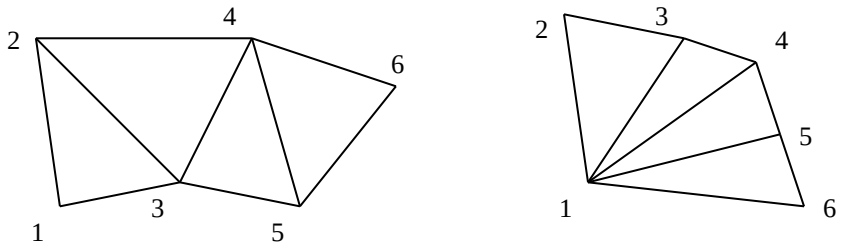
Lignes

Des lignes (séquence de segments) peuvent être définies sous forme de LINE_STRIP pour une ligne brisée non fermée, ou LINE_LOOP pour une ligne brisée qui sera fermée. La primitive LINES définit un ensemble de lignes déconnectées (les points sont considérés par paires). L'épaisseur d'une ligne peut être contrôlée.

Triangles

Les triangles représentent un cas particulier du polygone. Un triangle est un polygone à trois sommets. Des triangles isolés, non connectés les uns aux autres, sont dessinés par une primitive de type TRIANGLES. On considère alors les triplets consécutifs de sommets.

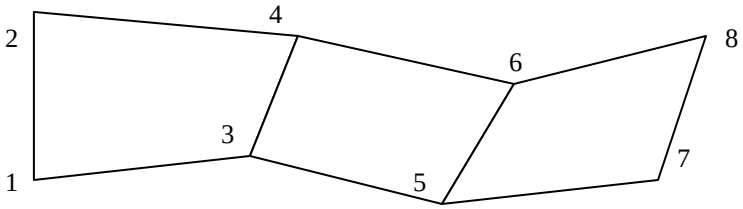
Lorsque les triangles sont connectés les uns aux autres, on peut définir des chaînes de triangles : TRIANGLE_STRIP ou des faisceaux de triangles : TRIANGLE_FAN, comme illustré par la figure ci-dessous (gauche « strip », droite « fan »).



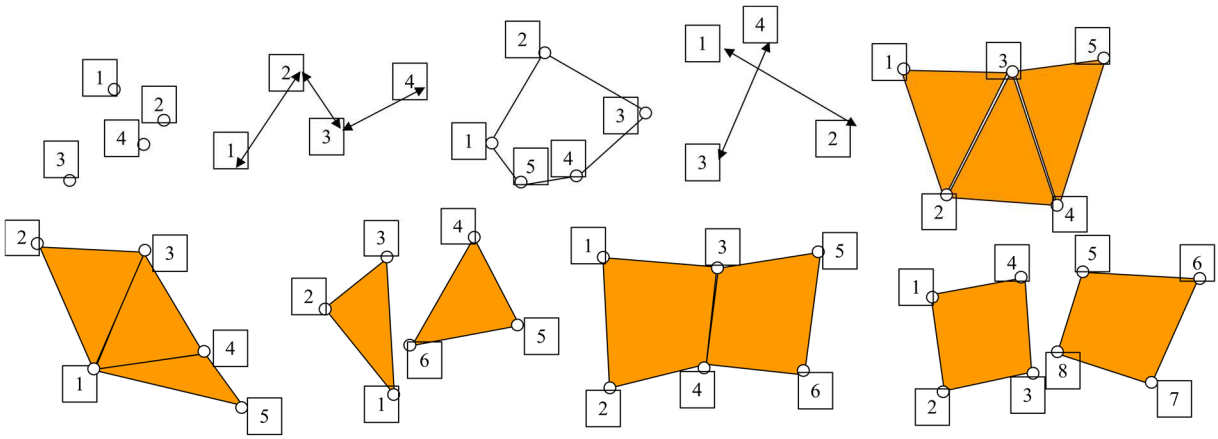
Les numéros sur cette figure indiquent l'ordre des sommets rangés dans le tableau.

Quadrilatères

Les « quads » représentent également un cas particulier de polygone : il s'agit d'un polygone à quatre coté (un quadrilatère). Comme pour les triangles, les quads peuvent être connectés ou non. Avec QUADS, ils ne sont pas connectés et on considère donc des quadruplets. Avec QUAD_STRIP, ils sont connectés comme illustré par la figure ci-dessous.



Récapitulatif des primitives :

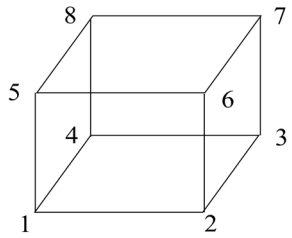


Notons que le « quad » n'existe pas en temps que primitive propre : il est converti automatiquement en triangles.

Il n'existe donc fondamentalement que 3 types de primitives : point, ligne et triangle.

Indexation des sommets

La représentation consistant à utiliser une entrée de tableau par sommet de polygone peut engendrer d'importantes redondances en mémoire lorsqu'il s'agit de modéliser un polyèdre. Un cube par exemple a six faces chacune ayant 4 sommets, soit 24 points. Mais dans les faits le cube n'a que 8 sommets car des faces différentes se partagent un même sommet. Une façon d'éviter de dupliquer des sommets en mémoire consiste à utiliser des tableaux d'indirection (pointeurs) :



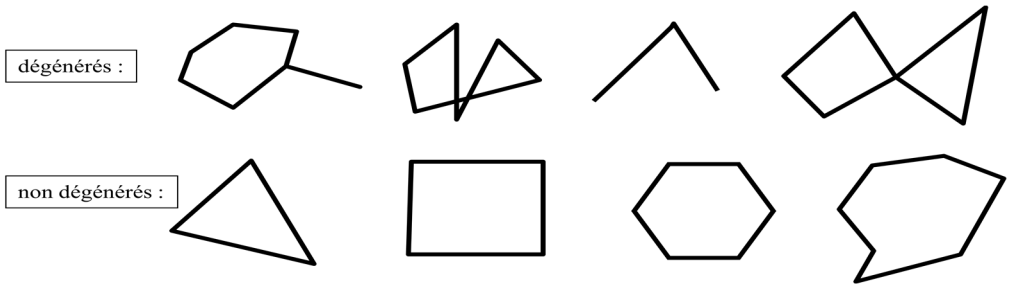
S1=(0,0,0) F1=(S1,S2,S3,S4)
S2=(1,0,0) F2=(S5,S6,S7,S8)
S3=(1,1,0) F3=(S1,S2,S6,S5)
S4=(0,1,0) F4=(S2,S6,S7,S3)
S5=(0,0,1) F5=(S3,S4,S8,S7)
S6=(1,0,1) F6=(S1,S4,S8,S5)
S7=(1,1,1)
S8=(0,1,1)

Nous utilisons alors deux tableaux : le tableau des coordonnées de sommets (8 sommets) et un tableau d'indices : tableau des faces (6*4 entiers).

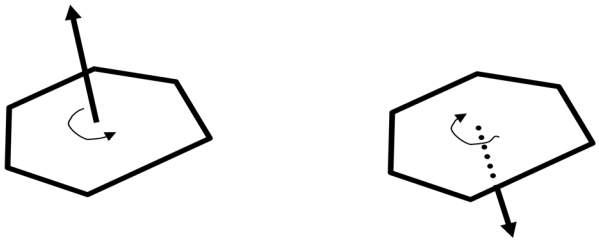
Mais attention aux attributs associés aux sommets. Si les sommets sont « fusionnés » alors leurs attributs le sont aussi. Par exemple, chaque sommet du cube ne peut avoir qu'une seule normale.

Quelques remarques concernant les polygones :

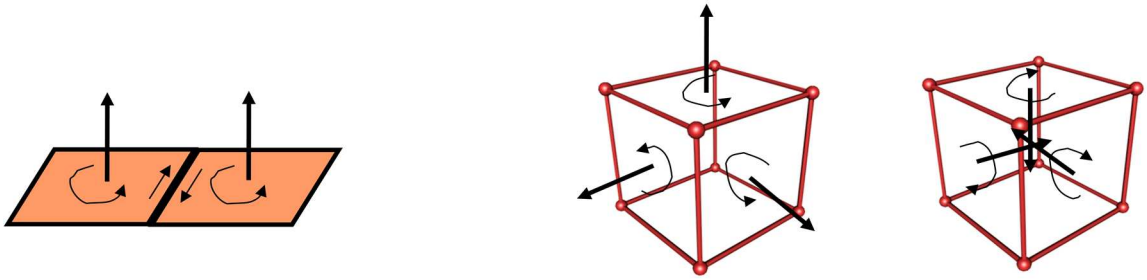
Un polygone est dit dégénéré, s'il n'est pas planaire, si son aire est nulle, ou s'il a des sommets confondus, ou une arête pendante, ou s'il a des arêtes qui se croisent (intersection non nulle) :



Tout polygone a deux cotés : choisir un sens de parcours des sommets revient à choisir l'orientation du polygone. Le vecteur normal à un polygone est le vecteur normal au plan du polygone. Il peut être dirigé dans deux directions selon le sens de l'orientation du polygone :



Orienter un polyèdre c'est orienter chaque polygone de façon à ce que deux polygones voisins aient la même orientation :



4. Le pipeline graphique standard

Un rendu GPU est toujours structuré sous la forme d'une suite de procédures appelées successivement (en utilisant une notation de type grammaire):

$$\text{Programme graphique GPU} = = (\Im\ ^{+}\Lambda^{*}\wp^{*})^{+}$$

$\Im$: procédures d'initialisation de paramètres globaux
Λ : procédures de transfert de données (buffers, constantes + variables)
$\wp$: tracé de primitives en **trois étapes:** transformation, rasterisation + coloriage

Il peut y avoir plusieurs **passes**: c'est à dire que des primitives graphiques sont envoyées non pas toutes d'un seul coup, mais « entrecoupées » d'initialisations et transferts. Il est même possible que certaines primitives puissent servir à créer des paramètres/attributs pour d'autres primitives.

Pour pouvoir utiliser la carte graphique, il faut impérativement **convertir les modèles géométriques "haut niveau" en primitives graphiques.**

Les procédures d'initialisation $\Im$ servent à mettre à jour des valeurs constantes (identiques pour toutes les primitives). Il existe deux types de constantes :

1) les **constantes globales** du **contexte** OpenGl, qui servent à configurer les composants non programmables ;
2) les paramètres des primitives, qui sont des **attributs**. Il y en a deux types. Les attributs:
   - **constants** (restent identiques d'une primitive à l'autre - ce sont les variables dites **uniformes**) ;

- **variables associées aux sommets** (les valeurs varient pour chaque primitive - ce sont les paramètres de type **in**).

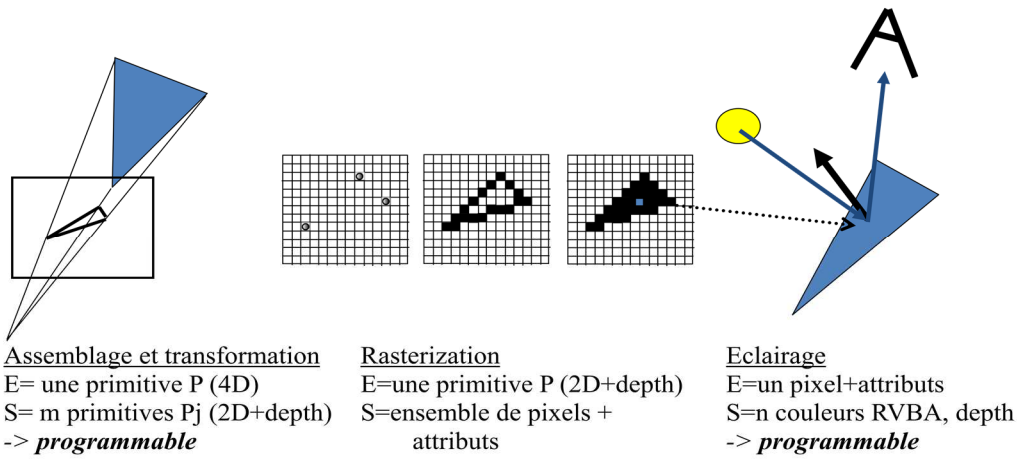
Les transferts de données Λ ont lieu : soit depuis la mémoire RAM centrale vers la mémoire RAM GPU, soit depuis la mémoire RAM GPU vers une autre zone de la mémoire RAM GPU, soit depuis la mémoire RAM GPU vers la mémoire RAM centrale. Les transferts se font par blocs : il s'agit de zones contigües représentant des tableaux 1D, 2D ou 3D de nombres flottant ou entiers regroupés par paquets d'une, 2, 3 ou 4 valeurs. Les tableaux sur GPU se nomment des **buffers objects**. Il existe essentiellement deux types distincts de buffers : **vertex buffer objects** et **frame buffer objects**.

- **Vertex buffer object** : contient les attributs des primitives et des données pour construire des primitives (par exemple le tableau des coordonnées 3D ou bien le tableau d'indirection). La fonction de transfert se nomme **DrawBuffer**.

- **Frame buffer object** : contient des « images » 1D, 2D ou 3D. Pour ce type de tableau, un paquet est appelé : **pixel**. Les fonctions de transfert se nomment alors respectivement : **DrawPixels, CopyPixels et ReadPixels**.

Lors du transfert il y a **transcodage des données** : c'est-à-dire que le format source n'est pas nécessairement le format de la destination. Par exemple, il est possible de transférer une matrice 2D de triplet RVB stockée au niveau de la RAM au format « float », puis de stocker en mémoire GPU une matrice RVB au format « unsigned char ». Chaque float est alors converti automatiquement lors du transfert en unsigned char. Il est également possible d'appliquer des **filtres de convolution** lors du transfert, le filtrage étant alors directement calculé sur le GPU (donc très rapide).

La figure ci-dessous résume le pipeline de la procédure ℘ de tracé d'une primitive :



L'assemblage et la transformation correspondent à deux « shader » : vertex shader et geometry shader (appliqués dans cet ordre). Ces programment permettent d'effectuer respectivement des traitements sur les sommets et sur les primitives.

La **rasterization** correspond au tramage. Cette étape n'est pas programmable. Il s'agit de déterminer quels sont les pixels appartenant à la primitive.

Le **calcul d'éclairage** (ou **coloriage**) consiste à associer à chaque pixel fourni par tramage une couleur+éventuellement d'autres attributs (un tel pixel se nomme un ***fragment***). Il s'agit d'un programme appelé le **fragment shader**.

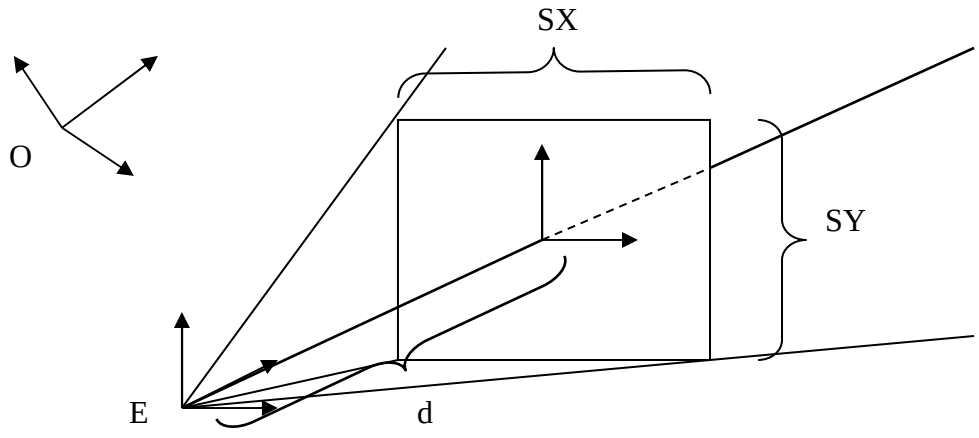
Ces trois programmes utilisent des **paramètres constants (uniform) et locaux (in)**. Ils fournissent des **paramètres en sortie (out)**.

Les paramètres des vertex / geometry shaders sont les deux types d'attributs qui peuvent être associés aux sommets. Les attributs: 1) **constants** (restent identiques d'une primitive à l'autre - ce sont les variables dites **uniformes**) et 2) **associées aux sommets** (les valeurs varient pour chaque primitive - ce sont les variables de type **in**). Ces deux shaders fournissent en sortie (out) des attributs, qui seront les attributs (in) du fragment shader. Comme le calcul se fait par sommet, la valeur d'attribut par pixel est obtenue par **interpolation bilinéaire**. Le geometry shader a la particularité de fournir en sortie (donc en entrée de la rasterization) une ou plusieurs primitives, différentes de celles qui sont envoyées à la carte. Par exemple le geometry shader peut prendre en entrée des primitives LINES et fournir en sortie, pour chaque segment, trois TRIANGLES. Le fragment shader fournit en sortie nécessairement une **couleur** RVBA (il peut en fournir plusieurs), ainsi qu'une **profondeur Z**.

5. Modèle de caméra sténopé

Visualiser une structure (ou un modèle) en trois dimensions signifie en tout premier lieu être capable d'effectuer une transformation d'un espace 3D en un espace 2D (l'écran). C'est ce que font naturellement les capteurs optiques, comme par exemple notre œil ou un appareil photographique. Les capteurs optiques sont basés sur un composant matériel essentiel : la lentille. C'est la lentille qui est à l'origine d'une transformation dite *perspective*, que nous allons décrire par la suite.

Dans le chapitre 1 (rappels), nous avons défini tous les éléments nécessaires à la réalisation d'une projection perspective. Une telle projection est définie par une matrice de transformation en coordonnées homogènes résultant de la combinaison de plusieurs transformations appliquées au point, qu'il convient à présent de définir. Considérons le schéma ci-dessous :



La projection perspective est définie par une position d'œil notée E=(Ex,Ey,Ez) dans le **repère du monde**. C'est le repère absolu dans lequel sont exprimés les éléments 3D de la scène. Dans notre cas, c'est un repère classique direct orthonormé que nous notons (O, i, j, k). Sur cette position d'œil, on place un autre repère, un repère local noté E, Ei(Eix,Eiy,Eiz), Ej(Ejx,Ejy,Ejz), Ek(Ekx,Eky,Ekz) où Ek définit la direction vers laquelle "on regarde" et Ej l'inclinaison par rapport à la ligne d'horizon. Ensuite, on définit un écran virtuel (perpendiculaire à Ek et sur lequel on projettera les points 3D) de taille SX et SY. Cet écran se trouve à une distance *d* de l'œil. Intuitivement, plus la distance *d* (focale) est faible plus les distorsions perspectives seront importantes.

Pour pouvoir appliquer la projection perspective, il nous faut d'abord ramener le repère vers l'œil (on combine une translation avec un changement de repère), puis il nous faut appliquer la transformation perspective et enfin, remettre à l'échelle de l'écran (selon SX et SY) de façon à avoir des coordonnées normalisées entre –1 et 1.

La phase de projection est généralement appliquée dans le « vertex shader » ou le « geometry shader ». La matrice suivante permet de définir une projection perspective où (xg, xd) donnent les positions gauche et droite de l'écran, (yb, yh) les positions basses et hautes de l'écran (si l'écran a une taille SX et SY, il s'agit typiquement de –SX/2,SX/2,-SY/2,SY/2), zp la distance focale de l'écran (z proche) et zl la distance maximale de considération des objets (z loin). Cette matrice combine donc la perspective brute avec la mise à l'échelle de l'écran. Après combinaison / multiplication des matrices correspondantes, elle a la forme suivante :

$$\begin{pmatrix} \frac{2zp}{xd-xg} & 0 & \frac{xd+xg}{xd-xg} & 0 \\ 0 & \frac{2zp}{yh-yb} & \frac{yh+yb}{yh-yb} & 0 \\ 0 & 0 & -\frac{zl+zp}{zl-zp} & -\frac{2\cdot zp\cdot zl}{zl-zp} \\ 0 & 0 & -1 & 0 \end{pmatrix}^T$$

Cette matrice normalise également la profondeur z entre –1 et 1 d'où la définition de zp et zl. A ce niveau il est important de remarquer qu'OpenGL utilise une discrétisation de la profondeur z des points (après projection), d'où l'importance de définir une distance maximale zl qui ne peut pas être infiniment grande. En fait, plus le rapport entre zp et zl est grand, moins la précision sera bonne pour les algorithmes de visualisation. Nous reviendrons sur ce problème lorsque nous introduirons l'algorithme du tampon de profondeur (le *Z-Buffer*). Notons qu'OpenGl effectue une projection avec un Z inversé (le z du point projeté doit être compris entre -zp et -zf (et non entre zp et zf).

La matrice suivante permet quant à elle de définir une projection orthogonale avec la même signification des paramètres:

$$\begin{pmatrix} \frac{2zp}{xd-xg} & ..0 & ..0 & ..-\frac{xd+xg}{xd-xg} \\ 0 & ..\frac{2}{yh-yb} & ..0 & ..-\frac{yh+yb}{yh-yb} \\ 0 & ..0 & ..-\frac{2}{zl-zp} & ..-\frac{zl+zp}{zl-zp} \\ 0 & ..0 & ..0 & ..1 \end{pmatrix}^T$$

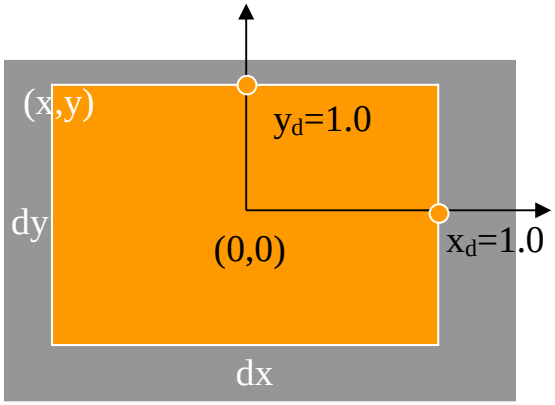
6. Le tramage (rasterization) et le clipping

6.1 L'écran discret : une matrice de pixels

Un écran est défini par un ensemble de pixels (petits carrés lumineux). La fonction la plus basique consiste à *allumer* un pixel en lui affectant une certaine couleur RVB. Un pixel est identifié par ses coordonnées sur l'écran.

Pour faire de l'imagerie de synthèse, l'écran est considéré comme un espace planaire 2D **continu**, sur lequel on repère la position d'un point par des coordonnées 2D (nombres réels). On se place dans le cas où le système de coordonnées est normalisé entre (-1.0, 1.0). Le point d'origine (0.0,0.0) représente le centre de l'écran. Le point de coordonnées (1,1) correspond au coin supérieur droit, et celui de coordonnées (-1,-1) au coin inférieur gauche.

Pour convertir les coordonnées réelles d'un point P en position de pixel, il faut définir la **résolution** de l'écran visible dans la fenêtre, appelé le « viewport ».



Pour pouvoir afficher des pixels à l'écran, il nous faut donc encore définir une dernière transformation : le passage de coordonnées réelles dans l'écran virtuel normalisé entre -1 et 1 à des coordonnées discrètes d'écran allant de 0 à résolution X (respectivement 0 à résolution Y). Ceci se fait en définissant un *Viewport* (une fenêtre ou cadre de vue). La commande OpenGl est :

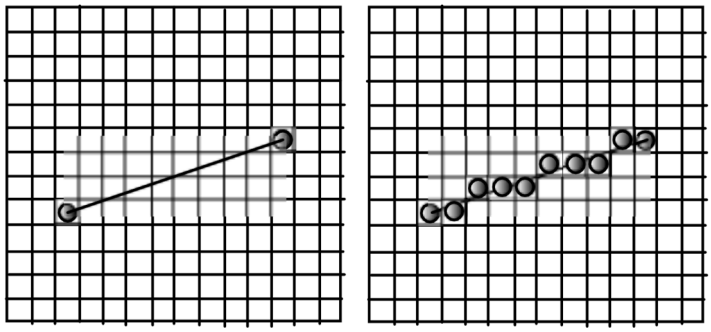
```
void glViewport(int x, int y, int dx, int dy) ;
```

Généralement, les valeurs sont respectivement 0,0, résolution X et résolution Y (pour couvrir tout l'écran).

Ainsi à chaque point continu de l'écran correspondra un pixel.

6.2 Tracer de lignes discrètes : Algorithme de Bresenham

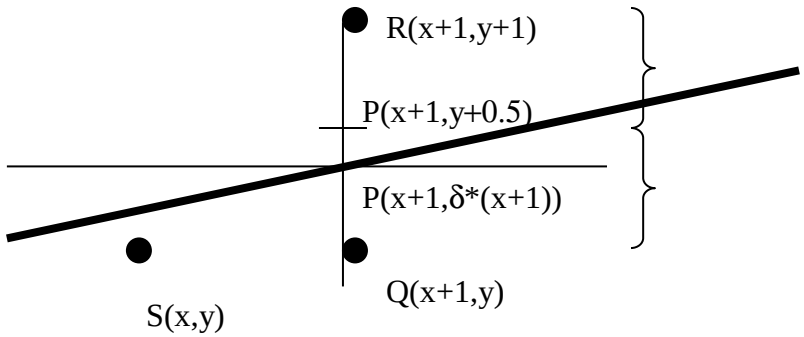
Tracer un segment de droite, c'est identifier un certain nombre de pixels pour former un ensemble de paliers reliant le point de départ avec celui d'arrivée. On est donc amené à proposer un algorithme de tracé de segment discret permettant de relier deux pixels par une suite de pixels proches de la droite continue joignant ces deux points.



Le premier algorithme de ce type a été proposé par Bresenham. Nous rappelons ici brièvement les principes de cet algorithme.

Soit [A,B] le segment à tracer, où A et B ont par exemple été obtenus par projection sur l'écran. Les points A et B appartiennent respectivement à 2 pixels P1(x1,y1) et P2(x2,y2). Nous allons considérer le cas où x1<x2, y1<=y2 et y2-y1<=x2-x1 (c'est à dire que P2 se trouve dans le premier octant par rapport à un repère dont l'origine serait en P1). Les autres cas se déduisent de celui-ci par simple symétrie.

Dans le cas du premier octant (nous supposons l'origine en P1), le tracé consiste à avancer pixel par pixel selon l'axe x et à décider à chaque pas s'il faut monter d'un cran ou non. En d'autres termes, si nous nous trouvons en un pixel S(x,y), nous devons faire un choix entre le pixel Q(x+1, y) ou R(x+1, y+1). Le dessin ci-dessous illustre ceci (les points représentent les centres des pixels):



Ce choix doit se faire par rapport au segment continu joignant A à B. Pour faire ce choix, nous pouvons utiliser le point médian P(x+1, y+0.5) et déterminer s'il se trouve au dessus ou en dessous du point idéal P(x+1, δ*(x+1)) où δ représente la pente de la droite, c'est à dire $\delta = \frac{y2-y1}{x2-x1}$. Si le point médian est au dessus alors c'est Q qui est retenu sinon c'est R.

Pour obtenir un algorithme efficace, Bresenham propose de calculer successivement et incrémentalement une erreur par rapport au point idéal en n'utilisant que des opérations sur les entiers. L'équation de la droite continue est donnée par : $x\ dy - y\ dx - x_1\ dy + y_1\ dx = 0$ (avec $dx = x2\text{-}x1$ et $dy=y2\text{-}y1$). Lorsque l'on applique cette équation à un point P(x,y) on obtient un scalaire $e= x\ dy - y\ dx - x_1\ dy + y_1\ dx$ négatif ou positif selon que le point se trouve d'un coté ou de l'autre de la droite (dans notre cas au dessus ou en dessous de la droite). Nous appliquons donc ce processus à P(x+1, y+0.5), d'où l'erreur : $e_p = e + dy - dx/2$, ou encore en multipliant par deux : $2\ e_p = 2\ e + 2\ dy - dx$ (utile si l'on veut éviter la division par deux). Si l'erreur est positive, e.g. $2\ e_p > 0$ alors on prend R sinon Q. L'erreur pour le pixel suivant e' se déduit de l'erreur du pixel précédent, selon la formule : $e'=e_p-dx$ si l'on prend R, et $e'=e_p+dx$ si l'on a pris Q.

Nous en déduisons un algorithme incrémental n'utilisant que des calculs sur des entiers :

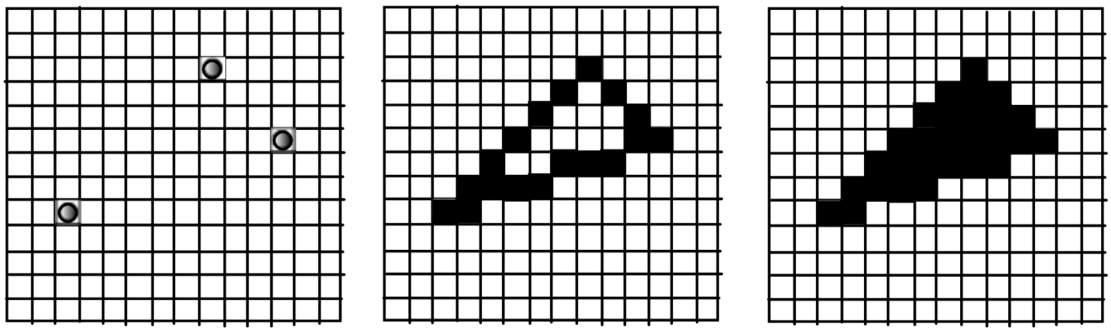
```
tracerVecteur (int x1, int y1, int x2, int y2)
      avec dx = x2 – x1, dy = y2 – y1, e = 0, y = y1
      pour x de x1 jusqu'à x2 faire
          tracerPixel(x,y);
          em = e + 2*dy - dx;
          si(em >= 0) alors
            y=y+1;
            e = em - dx;
          sinon
            e = em + dx;
```

Il suffit à présent de généraliser cet algorithme par symétrie pour l'appliquer à tous les cas de figure (les 7 autres octants).

Sous OpenGL le tracé d'un segment discret est évidemment implémenté. Aujourd'hui toutes les cartes graphiques proposent une accélération matérielle pour le tracé de segment.

6.3 Remplissage de polygones

Dans le paragraphe précédent, nous avons étudié un algorithme permettant d'effectuer un tracé de segment discret 2D, voire 3D sur un écran graphique. Ceci nous a permis d'effectuer de premiers dessins en 2D et 3D, dits de type "fil de fer". Nous voyons à présents comment les polygones que nous traçons peuvent être remplis d'une couleur afin de pouvoir afficher des formes 2D ou des polyèdres 3D « pleins ». La figure ci-dessous illustre ce que l'on entend par remplissage de polygone. Pour cela il existe plusieurs algorithmes.



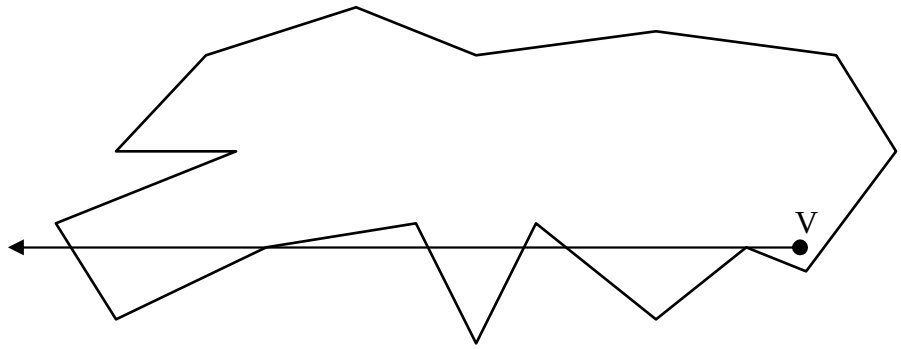
Remplissage d'un polygone par balayage de lignes (scan-line)

Cet algorithme consiste à balayer la boite englobante du polygone ligne par ligne en utilisant la propriété du test d'appartenance d'un point V à un polygone P. Nous rappelons ici brièvement ce test d'appartenance :

Soit D une des deux demi-droites issue de V et parallèle à l'axe Ox (axe horizontal). Soit n_r le nombre d'intersections de D avec les cotés de P et soit n_q le nombres de sommets P_i de P situés exactement sur D tels que :

- soit les sommets P_{i-1} et P_{i+1} sont de part et d'autre de la droite qui porte D ;
- soit P_iP_{i+1} est inclus dans D et les sommets P_{i-1} et P_{i+2} sont de part et d'autre de la droite qui porte D.

Si n_r+n_q est impair alors V est à l'intérieur sinon à l'extérieur. Un exemple est illustré par la figure ci-dessous : ici $n_r= 4$ et $n_q= 1$, donc V est à l'intérieur.



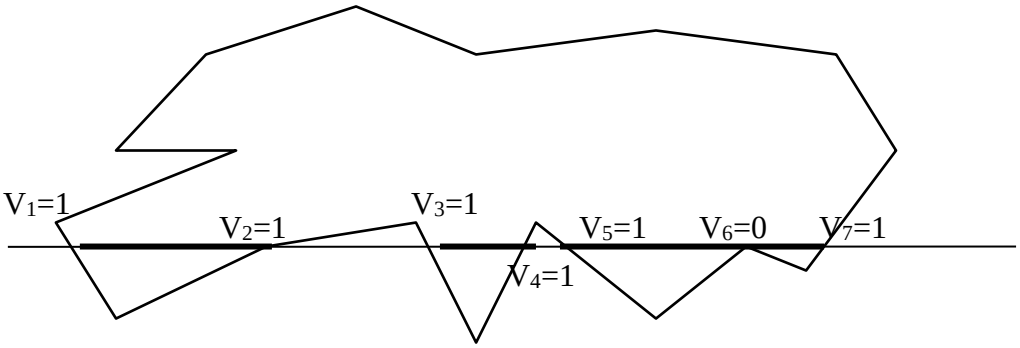
Nous pouvons exploiter ce test d'appartenance de la manière suivante : pour chaque ligne *j* de la boite englobante du polygone, calculer les points Qi d'intersection avec les arêtes du polygone et ordonner ces points en ordre croissant des abscisses. A chacun de ces points, on associe une valeur vi=0 ou 1 correspondant au nombre d'intersections virtuelles que ce point d'intersection représente. S'il s'agit d'un point à l'intérieur d'une arête, alors vi=1. S'il s'agit d'un sommet Pi alors :

- si les sommets précédent et suivant ne sont pas du même coté de la ligne de balayage alors vi=1 ;
- sinon vi=0 ;

Si l'arête PiPi+1 est horizontale (confondue avec la ligne de balayage), alors on ne compte que le second sommet Pi+1 comme intersection en suivant la même règle de test de coté en ignorant Pi.

Il suffit ensuite d'avancer horizontalement pixel par pixel en cumulant les valeurs de vi dans une variable tampon v initialisée à 0 et incrémentée à chaque fois que l'on atteint une intersection (e.g. v = v+vi). Si v est pair on ne trace pas, sinon on trace les pixels.

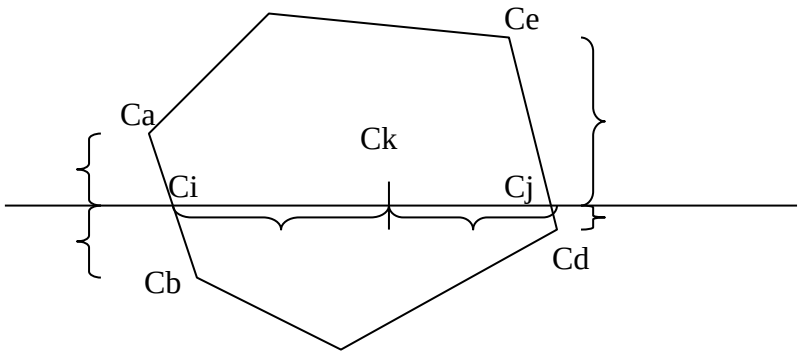
Pour l'exemple précédent nous obtenons :



Initialement v=0. Nous atteignons le premier point d'intersection sans faire de tracé (car v est pair). v devient égal à 1 (donc impair), nous commençons le tracé, jusqu'à v2, où v prend la valeur 2, donc le tracé est interrompu jusqu'à atteindre v3, etc.

6.4 Interpolation bilinéaire des attributs

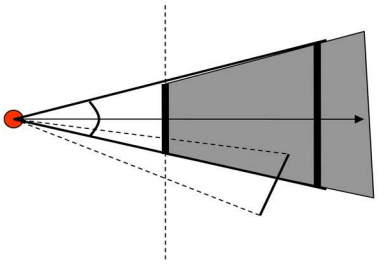
L'avantage de cette approche est que le remplissage peut se faire en interpolant les couleurs aux sommets du polygone. Pour chaque point d'intersection Qi nous pouvons déterminer une couleur obtenue par interpolation linéaire entre les sommets Pj et Pj+1 de l'arête correspondante. Entre les extrémités Qi, il est également possible d'interpoler linéairement. Le schéma ci-dessous représente le principe de cette interpolation *bi-linéaire* (doublement linéaire) :

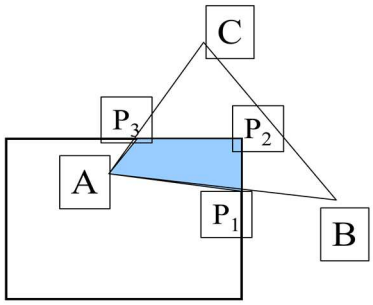


6.5 Clipping

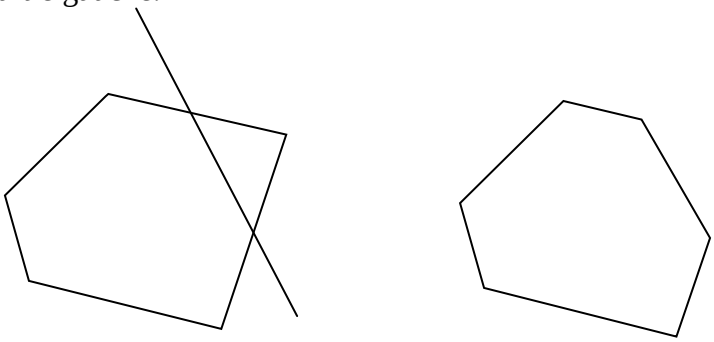
Lorsque les lignes dépassent les dimensions de l'écran, OpenGL applique une phase de ***clipping*** (découpage) avant de commencer le tracé. Cette phase consiste à découper le polygone ou les lignes par rapport à la forme rectangulaire de l'écran pour éviter les problèmes de débordement. La phase de clipping consiste simplement à utiliser un algorithme de découpage de polygone.

La figure ci-dessous illustre le cas du clipping d'un triangle. Le clipping est en 2D.



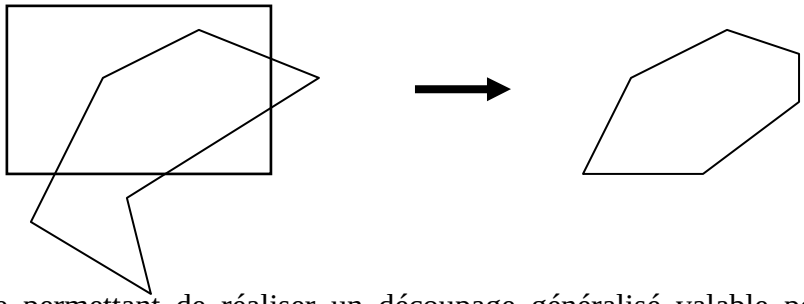


La première étape de découpage consiste en la scission d'un polygone par une droite. Si une droite traverse un polygone elle le coupe au moins en deux moitiés (ou plus si le polygone est concave). La figure ci-dessous illustre le découpage d'un polygone par une droite, en ne conservant que la partie gauche.



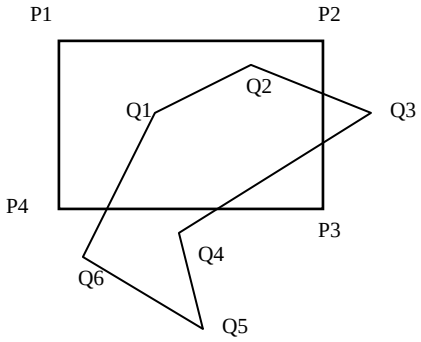
Pour simplifier le problème nous allons supposer le polygone *convexe*. L'algorithme est le suivant : pour chaque sommet Pi du polygone, on détermine si le segment [Pi, Pi+1] coupe la droite D en un point V. S'il y a intersection on ajoute V au polygone. Ensuite on détermine si Pi+1 est du bon coté de D. Si oui, on l'ajoute au polygone, sinon on ne l'ajoute pas. Puis on recommence pour l'arête suivante.

Calculer une intersection avec un polygone concave est plus complexe. Le résultat d'un tel découpage peut engendrer plusieurs nouveaux polygones. Une façon encore plus générale de considérer le problème est le suivant : découpage d'un polygone par un autre. La figure ci-dessous illustre un exemple de découpage généralisé :



Un algorithme permettant de réaliser un découpage généralisé valable pour tout type de polygone (concave et convexe) est celui de Weiler-Atherton, dont voici un bref descriptif.

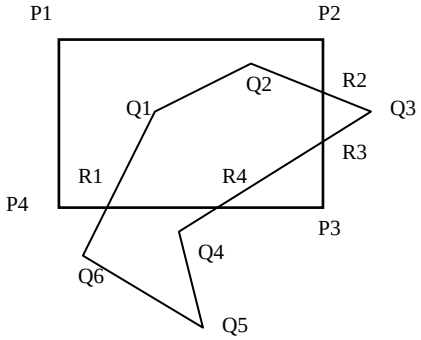
Tout d'abord les deux polygones sont placés dans des listes chaînées. Avec l'exemple ci dessous nous obtenons :

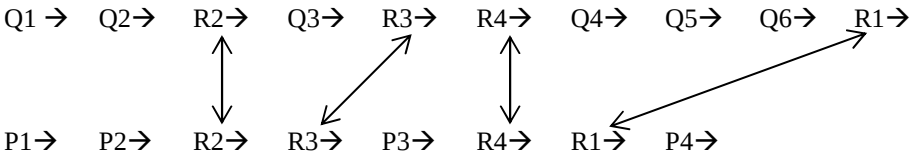


Q1 → Q2→ Q3→ Q4→ Q5→ Q6→

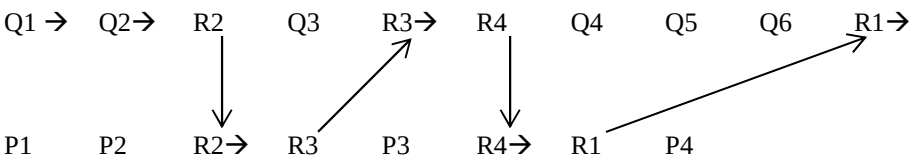
P1→ P2→ P3→ P4→

Puis on calcule les intersections Ri avec toutes les arêtes en les insérant au bon endroit dans les **deux** listes. Tous les points d'intersections sont donc présents dans les deux listes. Les couples identiques de points d'intersections sont également reliés par une relation bidirectionnelle. Dans l'exemple nous obtenons :





On part à présent d'un sommet intérieur (par exemple Q1), puis on suit le chaînage en ajoutant successivement tous les sommets visités. Lorsque l'on rencontre un point d'intersection on bascule d'une liste à l'autre en suivant la relation bidirectionnelle. On s'arrête lorsque l'on retombe sur le sommet initial. Dans l'exemple ci-dessus le chaînage donne en partant de Q1:

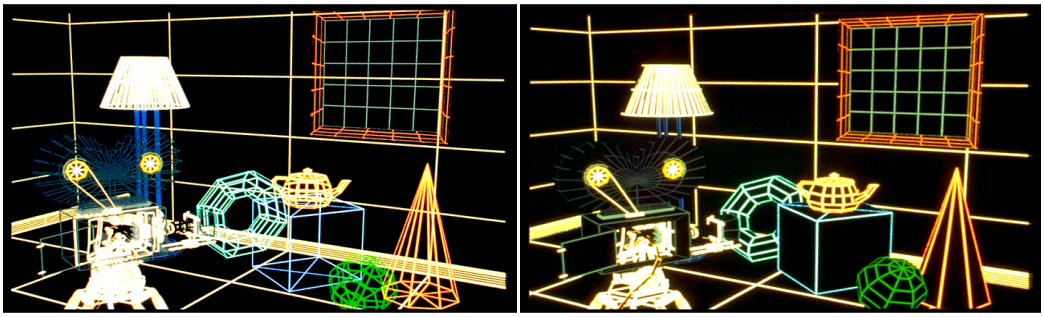


On obtient donc le polygone défini par :

Q1→ Q2→ R2→ R3→ R4→ R1→

7. Elimination des parties cachées

Lorsque l'on trace une scène (nous appellerons *scène un ensemble d'objets*) en faces pleines (avec remplissage) ou même en fil de fer, il faut tenir compte d'un phénomène physique essentiel permettant d'éviter de corrompre la perception visuelle humaine: les objets opaques bloquent la lumière et cachent ceux qui sont plus en arrière, si bien que l'on ne doit voir dans la scène que les objets les plus proches de l'observateur. Certains objets sont donc complètement invisibles ou bien partiellement occultés car ils sont cachés par d'autres objets plus en avant. Ce phénomène d'occlusion se traduit en informatique graphique par un calcul *d'élimination des parties cachées*. Dans ce paragraphe, nous étudions deux approches pour faire ce calcul : l'algorithme du peintre et l'algorithme du tampon de profondeur.

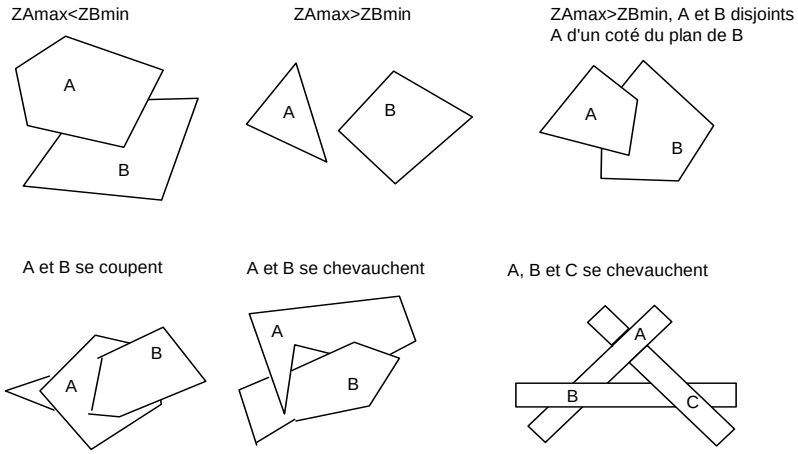


7.1 Tri en profondeur : algorithme du peintre

L'idée de base de cet algorithme proposé par Newell en 1972 est très simple: les polygones sont triés par ordre décroissant de distance à l'observateur (le plus loin en premier). Puis ils sont affichés dans cet ordre sachant que les plus proches recouvrirons automatiquement les plus éloignés et, par conséquence, les éliminerons intrinsèquement.

Malheureusement, un tel algorithme présente un certain nombre de défauts: l'éloignement du polygone n'est pas constant sur celui-ci, en effet cet éloignement peut varier entre un Zmin et un Zmax, or en Z deux polygones peuvent se croiser (on dit que l'ordre en Z n'est pas un ordre total).

Le schéma ci-dessous illustre différents cas de figure : en haut, il s'agit des cas qui ne posent pas de problèmes (ou du moins des problèmes triviaux faciles à résoudre), en bas il s'agit des cas qui posent problème.



Dans le premier cas les Z sont disjoints, il n'y a pas de problème. Dans le second cas les Z ne sont pas disjoints par contre les polygones se projètent sans chevauchement : il n'y a donc pas non plus de problème. Dans le troisième cas, les Z se chevauchent également, de plus les polygones ne se projètent pas de façon disjointe. On teste donc si le polygone A se trouve entièrement du même coté du plan formé par B : si oui il n'y a aucun problème.

Les trois figures du bas présentent les cas à conflit qu'il n'est pas possible de résoudre trivialement. Il faut dans ces cas couper les polygones en morceaux. Newell propose donc de résoudre les conflits en découpant les polygones qui posent problème.

7.2 Tampon de profondeur : *Z-Buffer*

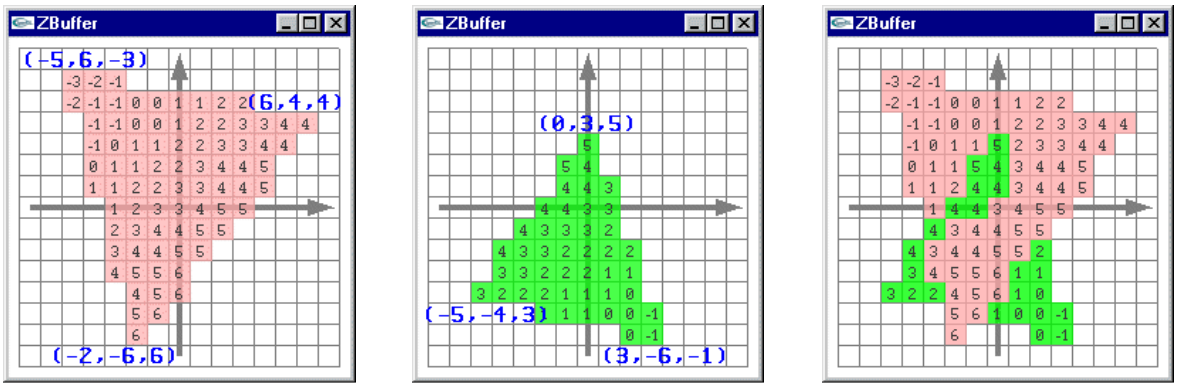
Selon la nature de la scène, l'algorithme précédent peut s'avérer très difficile à mettre en oeuvre. Nous présentons ici un algorithme proposé par Catmull en 74 ne nécessitant pas de découpage de polygone et de surcroît très simple à mettre en oeuvre. Il s'agit de l'algorithme du tampon de profondeur.

La technique du tampon de profondeur consiste à utiliser en plus de l'image R, V et B une image de profondeurs Z. A chaque pixel est alors associée une couleur et une profondeur (la distance à l'observateur).

Lors de la phase de remplissage du polygone, la couleur n'est mise à jour sur l'écran que si un test de comparaison entre la profondeur courante et la profondeur stockée est positif. L'algorithme est la suivant :

```
Effacer l'écran et initialiser tous les Z du buffer à l'infini
Pour chaque polygone P faire
      Projeter et remplire P avec interpol. des Z aux sommets
      Pour chaque pixel de P comparer sa valeur Z interpolée
       avec celle qu'il y a dans le buffer
      Ne remplacer le pixel que si la valeur Z est inférieure
```

La figure ci dessous illustre un exemple de tracé :



Les deux premières figures, à gauche et au milieu, illustrent les tracés séparés de deux triangles avec interpolation des valeurs Z (chiffres à l'intérieur des pixels). Celle de droite illustre le tracé complet avec comparaison des Z. Notons qu'avec le test de profondeur l'ordre dans lequel les polygones sont parcourus n'a pas d'importance contrairement à l'algorithme du peintre.

Le principal inconvénient de cet algorithme est qu'il nécessite le stockage d'une information supplémentaire : la profondeur Z. Si cette profondeur est stockée par des réels simples cela signifie 4 octets par pixel en plus des 3 octets pour l'image RVB. Avec une résolution de 1280x1024 cela fait 8,75 Mo en tout (RVB + Z). Généralement la valeur en Z est discrétisée sur 16 ou 24 bits selon la précision voulue. Il faut alors définir un Z-infini correspondant à une distance maximale au delà de laquelle les polygones seront ignorés (profondeur +1 si elle est normalisée entre –1 et +1). Nous avions abordé le sujet lors de la définition des matrices de projection perspective sous OpenGl.

Bien choisir la précision du buffer Z en fixant le Zmax (Z loin dans la définition de la matrice de projection) est très important, car cela peut avoir des conséquences déterminantes sur la qualité de l'affichage. Avec la discrétisation de la profondeur se pose par ailleurs un problème de choix : en effet, on peut remplacer un pixel si la valeur en Z est strictement inférieure ou si elle est inférieure ou égale.

L'élimination des parties cachées est un élément fondamental de la visualisation. Il est donc implémenté sous OpenGl et supporté matériellement par toutes les cartes graphiques actuelles. L'algorithme utilisé est celui du tampon de profondeur. Lors de l'affichage, nous devons indiquer à OpenGl qu'il faut procéder au test de profondeur. Nous devons également définir de quelle manière ce test doit s'opérer. Enfin, lorsque l'on efface l'écran avec glClear il ne faut pas oublier d'initialiser également le tampon en Z.

8. Eclairage et lissage

Les objets que nous voyons autour de nous ne sont visibles que parce qu'ils sont éclairés par des sources de lumière : le soleil, le ciel, une lampe, etc.. En effet, chaque objet réémet une partie de la lumière qu'il reçoit et c'est ce qui les rend visibles. Si nous voulons visualiser des objets 3D virtuels de façon convaincante, il faut donc tenir compte de ce phénomène.

Il est facile de se convaincre que la lumière réémise par une face est proportionnelle à la quantité de lumière qu'elle a reçu. Plus la face reçoit de "photons" plus elle sera "éclaircie", moins elle en reçoit, plus elle paraîtra sombre. Un objet ne recevant aucune lumière reste

complètement noir, puisqu'il ne peut rien réémettre. En dehors de la quantité reçue, il existe des objets qui réémettent plus de lumière que d'autres : plastique blanc par rapport à du charbon. Cela dépend donc aussi de la nature du matériau et plus particulièrement d'une propriété qui s'appelle la *réflectance*. Un "objet noir ou très sombre" est un objet qui absorbe une grande partie de la lumière qu'il reçoit.

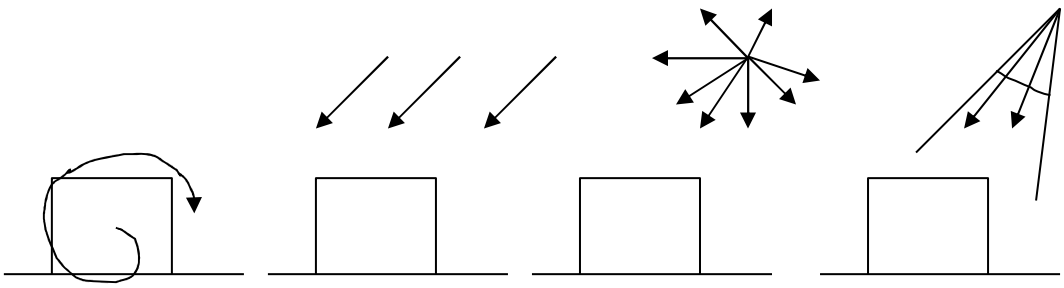
En résumé, deux éléments caractérisent l'éclairage d'une surface :

- La source de lumière ;
- La façon (qualitativement et quantitativement) de ré-émettre la lumière reçue ;

Nous allons donc voir dans un premier temps différents types de sources de lumière couramment utilisées en visualisation, puis comment modéliser la façon de ré-émettre cette lumière.

8.1 Sources lumineuses

Le schéma ci-dessous illustre 4 types de sources « classiques »: la lumière ambiante (source omniprésente), la source directionnelle, la source ponctuelle et le spot.



La lumière ambiante correspond à ce que nous avons fait jusqu'ici. Il s'agit de donner à la surface une couleur>0. On peut imaginer qu'il s'agit d'un éclairage omniprésent et constant à travers l'espace. Avec un tel éclairage il n'est pas possible de voir le relief des objets.

La source directionnelle est définie par un vecteur *V* matérialisant le sens constant de parcours des photons. Il peut s'agir par exemple d'une source placée à l'infini.

La source ponctuelle émet de la lumière dans toutes les directions en partant d'un point *P*. Il peut s'agir par exemple d'une ampoule. Cette source est caractérisée par une position dans l'espace 3D.

Le spot est une variante directionnelle de la source ponctuelle. Elle est caractérisée par une position (un point) plus un vecteur directeur et un angle d'ouverture. Il s'agit donc d'un cône à base circulaire. Au delà de ce cône aucune lumière n'est émise. Au sein du cône, la lumière est émise comme pour une source ponctuelle. Il est possible de moduler l'intensité d'un spot en fonction de l'éloignement angulaire par rapport à l'axe central du cône.

8.2 Modèle d'éclairage Lambertien

Voyons à présent comment nous pouvons calculer une couleur C (un vecteur R,V,B) pour une face donnée de polyèdre en fonction d'une ou plusieurs sources de lumière comme décrite précédemment.

Le modèle le plus simple consiste à supposer que la lumière réémise par un élément de surface ne dépend que de la quantité reçue. Une partie de l'énergie lumineuse est absorbée par l'objet (transformée en chaleur) et l'autre partie est restituée de manière équiprobable : même quantité dans toutes les directions. Un tel modèle s'appelle un modèle **Lambertien**. La *réflectance* est caractérisée dans ce cas par un simple scalaire $k_d<1$. On parle aussi de la composante diffuse de réflectance. Si la face reçoit une intensité lumineuse I alors elle réémettra une couleur : $C=IC_sk_d$. où C_s est la couleur de la face et k_d sa constante de réflectance.

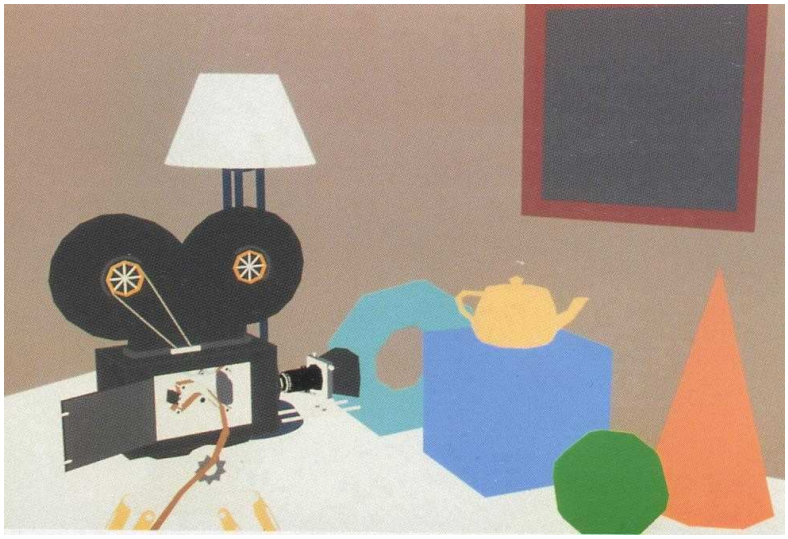
Pour procéder à l''évaluation de la couleur sur la face du polyèdre, il nous faut aussi évaluer la quantité de lumière I reçue par rapport au type de source considéré.

Lumière ambiante

Dans le cas d'une lumière ambiante, la lumière est la même partout : la couleur C est donc obtenue par :

$$C=I_aC_sk_d$$

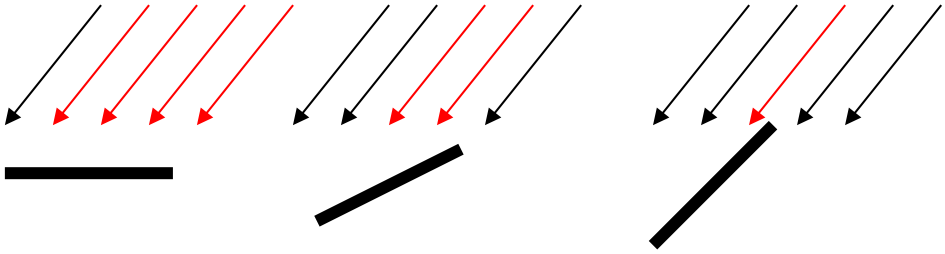
où I_a est l'intensité de la lumière ambiante et C_s la couleur de la surface. La figure ci-dessous illustre un exemple de scène plus complet utilisant ce type d'éclairage ambiant :



Lumière directionnelle

Dans le cas d'une lumière directionnelle, la quantité de la lumière reçue dépend de l'orientation de la surface. Le schéma ci-dessous illustre que plus la face est orientée vers la direction de la source plus le nombre de photons reçus (flèches rouges) est important, alors

que plus cette face est orientée perpendiculairement, moins elle en reçoit. Ce terme de proportionnalité correspond à une aire projetée et s'exprime à l'aide d'un cosinus.

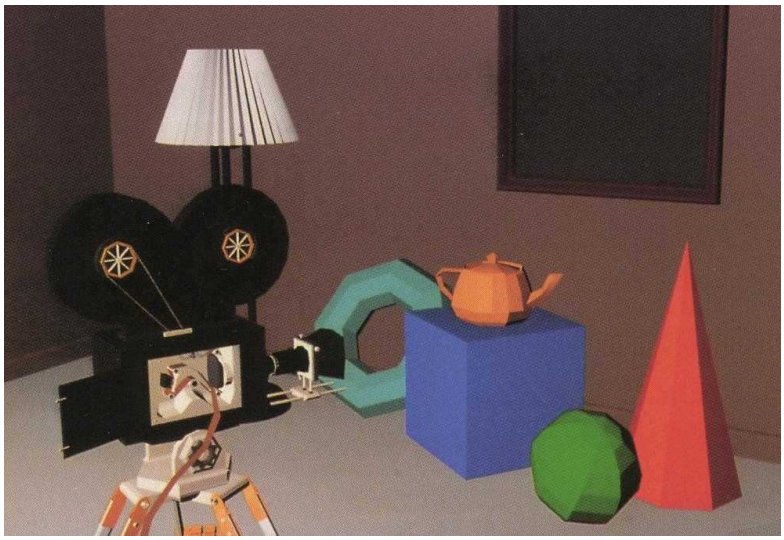


Sur ce schéma les flèches rouges indiquent les « photons » qui atteignent la surface. Plus la surface est dans l'orientation moins il y a de flèches rouges.

L'énergie *I* reçue par la face correspond donc à l'intensité de la source I_d pondérée par le cosinus de l'angle entre la direction V de la source et la normale N de la face : $I=I_d cos(V,N)$. Nous obtenons donc la formule suivante :

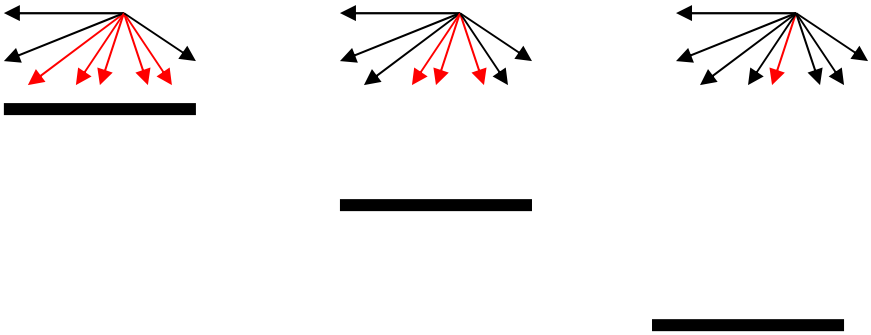
$$C = I_d C_s k_d \cos(V, N)$$

où I_d est l'intensité de la lumière directionnelle, C_s la couleur de la surface, V le vecteur directeur de la lumière et N la normale de la surface. La figure ci-dessous illustre la même scène que précédemment en utilisant une source directionnelle. On remarque qu'avec ce type d'éclairage le relief et volume de la scène apparaît très clairement.



Lumière ponctuelle

Le cas de la lumière ponctuelle est très similaire au cas de la lumière directionnelle. L'énergie reçue dépend également de l'angle entre la source et la normale, sauf que cet angle n'est pas constant sur la surface, les rayons lumineux étant issus d'un point. Mais en plus de cet angle, il faut également tenir compte de l'éloignement de la face par rapport à la source. Le schéma ci-dessous illustre que plus on s'éloigne de la source, moins on reçoit de photons.



L'énergie I reçue par la face correspond donc à l'intensité de la source I_p pondérée par le cosinus de l'angle entre la direction V de la source et la normale N de la face, ainsi que par la distance: $I=\frac{I_p \cos(\overrightarrow{SP},N)}{\overrightarrow{SP}^2}$. Nous obtenons donc la formule suivante :

$$C=I_p C_s k_d \frac{\cos(\overrightarrow{SP},N)}{\overrightarrow{SP}^2}$$

où I_p est l'intensité de la lumière directionnelle, C_s la couleur de la surface, S la position de la lumière, P le point sur la surface et N la normale en P.

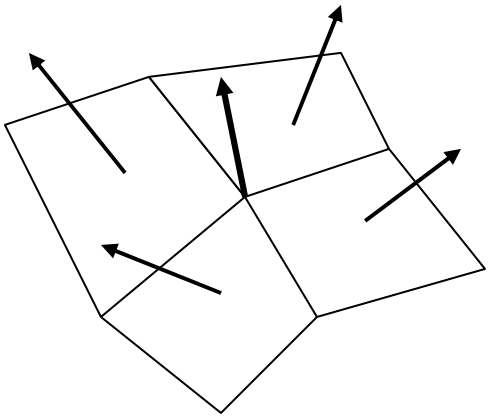
Lumière de type "spot"

Dans le cas d'une lumière de type spot la formule est similaire, sauf que l'on teste préalablement si le point de la surface se situe à l'intérieur du cône de lumière définie par le vecteur directeur et l'angle du spot.

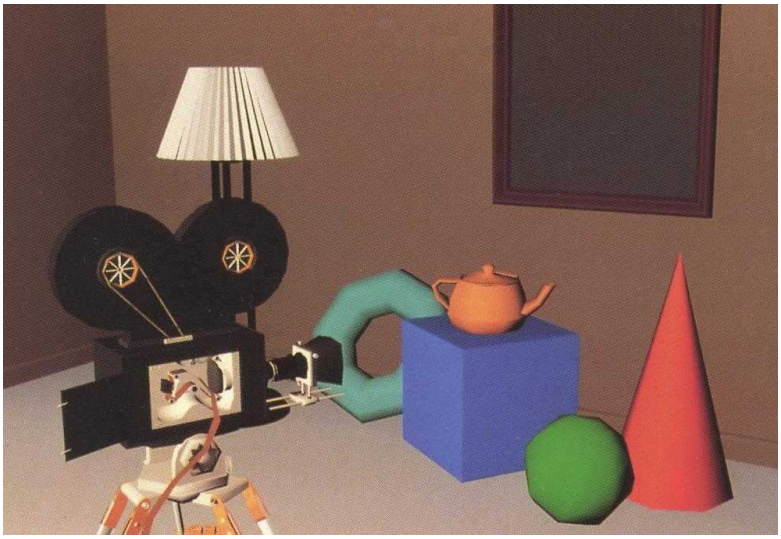
8.3 Lissage de Gouraud

Lorsque l'on affiche une scène définie sous la forme d'un ensemble de polyèdres en modulant la couleur à l'aide d'une source de lumière, le relief devient clairement visible. C'est le cas de la figure précédente qui montre une scène contenant un certain nombre d'objets géométriques : cube, boule, cône, théière, etc. Cependant avec une source de lumière, la visualisation révèle aussi la nature discrète de la scène : les facettes polygonales des polyèdres apparaissent très visiblement. Or généralement, les polyèdres servent à approcher des surfaces continues : par exemple, la boule verte devrait apparaître comme une boule lisse et non pas comme un polyèdre.

Pour donner l'impression d'être en présence d'une surface lisse et continue sans toutefois augmenter la précision des polyèdres, Henri Gouraud propose d'utiliser une interpolation bilinéaire. Plutôt que de calculer l'intensité lumineuse pour toute la face à l'aide de sa normale N, il propose de calculer l'intensité lumineuse aux sommets du polyèdre, en utilisant en chacun des sommets une normale moyennée : une normale obtenue en effectuant la moyenne de toutes les normales des faces incidentes au sommet (figure ci-dessous).



Pour chacun de ces sommets nous pouvons alors utiliser une des formules de calcul de couleur par rapport au type de source considéré. En chaque sommet du polygone à tracer, nous disposons d'une couleur différente que nous pouvons afficher par interpolation lors du remplissage de polygone par algorithme de type "scanline". La figure ci-dessous illustre le résultat d'un lissage de Gouraud dans le cas de la scène précédente.

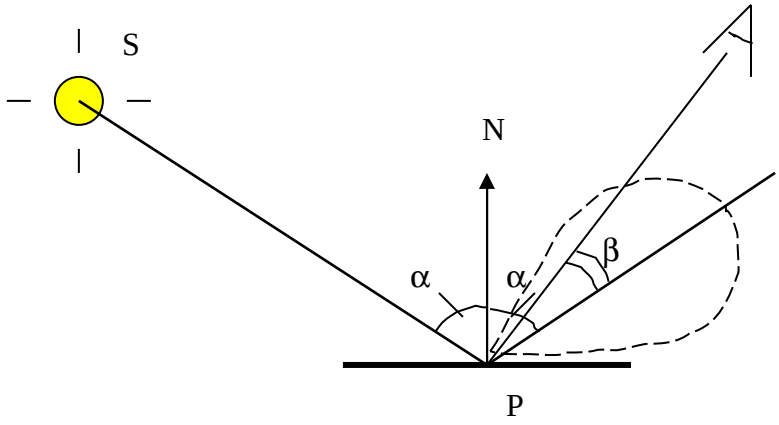


Noter que les surfaces semblent continues et que seuls les bords des polyèdres révèlent encore leur nature discrète.

8.4 Modèle d'éclairage de Phong

Le modèle Lambertien que nous avons décrit précédemment permet de visualiser des objets en 3D en introduisant une notion de lumière et d'éclairage. Cependant tous les objets visualisés de cette manière prennent une apparence matte sans caractériser la matière réelle de la surface. Nous avons en effet supposé que la lumière était réémise de façon équiprobable. Or, la plupart des matériaux et objets réels ont souvent une légère brillance qui font apparaître sur leur surface le reflet de la source de lumière. On parle de reflet ***spéculaire***.

Plus l'observateur se trouve face à la direction idéale de réflexion de la source sur la surface, plus ce reflet est intense. Ceci peut se schématiser de la façon suivante :

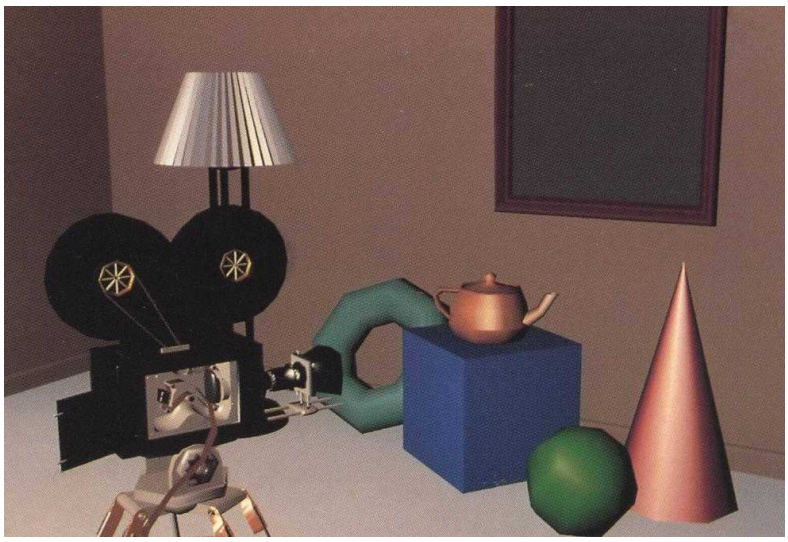


Plus l'angle β est proche de zéro plus on voit le reflet de la source de lumière (ce qui est représenté en pointillés sur le schéma). Phong propose un modèle d'éclairage plus sophistiqué pour tenir compte de reflets spéculaires. Il divise la réflectance en deux composantes : une composante diffuse caractérisée par le scalaire k_d et une composante spéculaire caractérisée également par un scalaire noté k_s. Une partie de la lumière est diffusée selon la loi de Lambert (partie matte) et une autre est réfléchie sur la surface (partie brillante). Pour modéliser l'amplitude de la réflexion, Phong propose d'utiliser le cosinus de l'angle β élevé à une puissance *n* permettant de contrôler l'amplitude du "reflet spéculaire", un peu comme pour une lumière de type spot (plus l'observateur s'éloigne de l'angle idéale β, plus le reflet faiblit). Pour une source ponctuelle, la formule devient alors la suivante :

$$C = I_p C_s k_d \frac{\cos(\overrightarrow{SP}, N)}{\overrightarrow{SP}^2} + I_p k_s \frac{\cos^n(\overrightarrow{EP}, R)}{\overrightarrow{SP}^2}$$

où E représente la position de l'œil et R le vecteur de réflexion de SP par rapport à N.

La figure ci-dessous illustre l'exemple de la scène en utilisant le modèle d'éclairage de Phong avec le principe de lissage de Gouraud (interpolation bilinéaire d'une couleur calculée aux sommets). On note les taches spéculaires sur certains objets.



8.5 Lissage de Phong et “fragment shader”

On remarque sur l’image précédente que l’interpolation bilinéaire de Gouraud est male adaptée à des variations importantes sur la surface. C’est le cas des reflets spéculaires, somme toute assez peu visibles sur cette image. Phong a donc proposé une approche plus sophistiquée toujours basée sur une interpolation bilinéaire : plutôt que de faire une interpolation des *couleurs* au sommet, Phong propose de faire une interpolation des *normales* et **d’appliquer la formule d’éclairage en chaque pixel lors du remplissage** en utilisant la normale interpolée.

La figure ci-dessous en illustre le résultat.

Sur cette images les reflets spéculaires apparaissent très nettement, donnant ainsi aux objets une apparence moins matte et plus brillante (plus plastique). L’impression de lissage est également renforcée.

Si l’on considère plusieurs sources de lumière en plus d’une émission propre et d’une lumière ambiante, on obtient le « fragment shader de Phong » suivant :

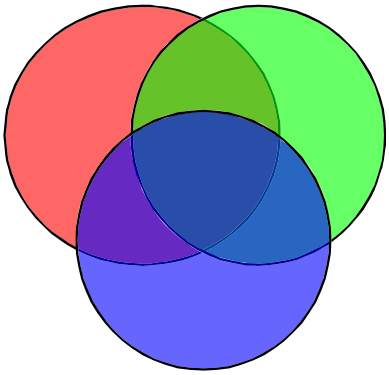
$$C = C_e + I_a K_d + \sum_{i=1}^{n_s} I_p^i K_d \frac{\cos(\overrightarrow{S_i P}, N)}{k_0 + k_1 \overrightarrow{S_i P} + k_2 \overrightarrow{S_i P}^2} + I_p^i K_s \frac{\cos^n(\overrightarrow{EP}, R_i)}{k_0 + k_1 \overrightarrow{S_i P} + k_2 \overrightarrow{S_i P}^2}$$

Cette formule nous permet de calculer la couleur en chaque pixel d’un polygone en fonction d’une ou plusieurs sources de lumière. Notons que cette formule n’est, en réalité, pas physiquement correcte. Elle a tendance à beaucoup trop simplifier le problème de la réflectance, si bien que les résultats visuels obtenus ne peuvent pas encore être qualifiés de photo-réalistes. Tous les objets prennent une apparence trop “plastique”. Néanmoins, il s’agit d’un modèle simple à mettre en œuvre et facile à micro-programmer. La plupart des cartes graphiques actuelles supportent ce type de shader (il s’agit d’ailleurs souvent du « seul » shader implanté par défaut dans les cartes si l’on ne programme pas soi-même le fragment shader).

9. Transparence par alpha-blending

Dans les deux paragraphes précédents, “élimination des parties cachées” et “éclairage”, nous avons supposés les objets toujours opaques. Or dans la réalité, il existe des objets transparents qui laissent passer la lumière et qui permettent donc de voir d’autres objets au travers. C’est le cas d’une vitre ou d’un verre par exemple. La transparence correspond à une $4^{ième}$ composante dans l’espace des couleurs RVB. C’est le coefficient *alpha* que nous avons déjà rencontré mais pas encore utilisé.

Dans certains cas très simples, la transparence peut être considérée comme un **filtrage de la couleur** (sans réfraction - déviation - du rayon lumineux) : lorsqu’un rayon passe à travers un tel objet transparent, une portion de son énergie lumineuse est retenue, le reste passe au travers. La lumière qui passe au travers dépend de la couleur de la surface transparente. La figure ci-dessous illustre le filtrage par surface transparente.



Lorsqu’un rayon d’énergie lumineuse *I* traverse un objet transparent de couleur C_s et de transparence k_t, l’énergie portée par le rayon en quittant la surface traversée est $I'=IC_sk_t$.

La prise en compte de la transparence requiert un algorithme d’élimination des parties cachées différent du Z-Buffer, car celui-ci suppose les objets opaques. Par ailleurs lorsque l’on affiche une surface transparente, il ne faut pas *remplacer* les couleurs qu’il y a déjà à l’écran, mais les combiner avec celles-ci: cette combinaison se fait de la manière suivante :

$$C = C_e(1-k_t)+ C_fC_sk_t$$

C_f représente la valeur qu’il y a dans l’écran, C_s la couleur de la surface transparente, k_t sa transparence et C_e la couleur de la surface transparente après calcul d’éclairage selon sa composante diffuse et spéculaire. Si k_t vaut 0 alors la surface est complètement opaque et ne laisse pas passer la lumière. C correspond à la valeur C_e calculée par éclairage de Phong exactement comme le modèle complètement opaque. Si la transparence k_t de la surface vaut 1 et sa couleur C_s est blanche (1.0, 1.0, 1.0) alors la surface est complètement invisible, la valeur de C reste inchangée et correspond exactement à la valeur C_f qu’il y a déjà à l’écran.

L’algorithme permettant de traiter la transparence est le suivant :

1. Appliquer un Z-buffer classique en ne considérant que les surface opaques
2. Utiliser un algorithme du peintre pour les surfaces transparentes :

2.1 Trier les surfaces selon Z de la plus éloignée à la plus proche
2.2 Dessiner les surfaces en appliquant la formule de combinaison des couleurs

Mélanger les couleurs en utilisant la formule précédente s'appelle plus généralement en anglais faire du *blending*. Une formule générale de blending est la suivante :

$$C = AC_e + BC_f$$

Où A et B sont des vecteurs de couleur. Cette formule est implantée sous OpenGl et est contrôlée par la commande glBlendFunc. Cette fonction prend deux paramètres permettant de fixer A et B. Par exemple :

```
glBlendFunc(GL_ONE_MINUS_SRC_ALPHA, GL_SRC_ALPHA) ;
```

permet d'appliquer la formule de transparence classique où A est donné par "un moins alpha" (donc $1-k_t$) de la source (la *source* SRC est la surface à dessiner, la *destination* DST est la valeur dans le framebuffer) et où B est donné par cette même transparence. Les constantes possibles pour A et B sont : GL_ZERO, GL_ONE, GL_SRC_ALPHA, GL_ONE_MINUS _SRC_ALPHA, GL_DST_ALPHA, GL_ONE_MINUS_DST_ALPHA, GL_SRC_COLOR, GL_DST_COLOR, GL_ONE_MINUS_SRC_COLOR, GL_ONE_MINUS_DST_COLOR. La formule précédente et les différentes valeurs que peuvent prendre A et B permettent de faire des combinaisons entre couleurs qui vont au-delà de la simple transparence.

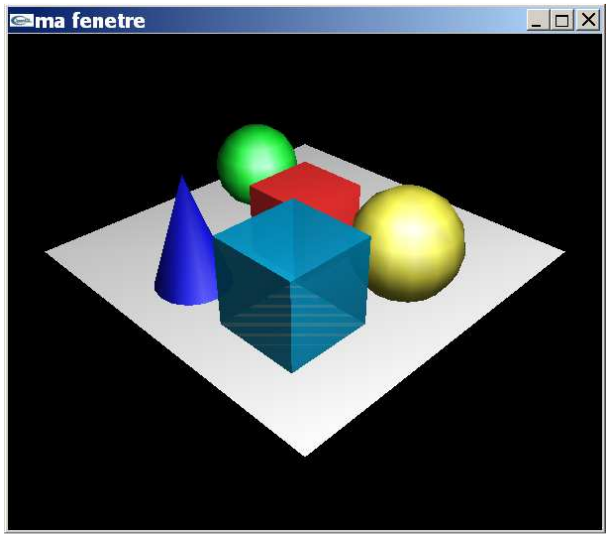
La valeur alpha d'une surface est fixée en même temps que sa couleur avec glColor4f, qui prend 4 paramètres au lieu de trois, le quatrième étant alpha.

Pour activer la calcul du blending, il faut utiliser : glEnable(GL_BLEND) ;

Par exemple:

```
glEnable(GL_BLEND);
glBlendFunc(GL_ONE_MINUS_SRC_ALPHA, GL_SRC_ALPHA);
glColor4f(0.0, 0.6, 0.8, 0.2);
// dessin
glDisable(GL_BLEND);
```

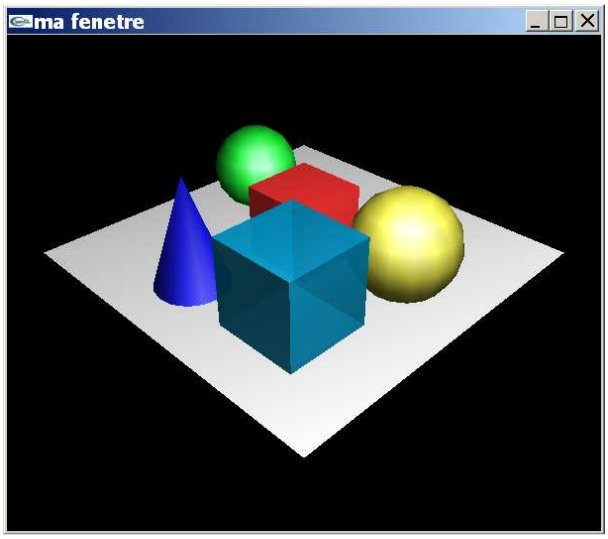
on obtient par exemple le résultat graphique suivant:



La valeur alpha de 0.2 représente une transparence de 20%. On note que sur le fond du cube transparent apparaissent des lignes qui ressemblent à un “bug” dans le programme. Il s’agit effectivement d’une erreur, mais pas dans la programmation. Il s’agit d’une erreur d’imprécision sur le Z-buffer, le fond du cube étant confondu avec le sol et les nombres flottants engendrant des erreurs de calcul. A cause de ces imprécisions, il se peut, lorsque des surfaces sont confondus, que le test en Z passe ou ne passe pas (souvent alternativement). OpenGl prévoit ce type de problème et permet à l’utilisateur de définir des marges d’erreurs, sous la forme d’un “offset” (petite valeur que l’on ajoute au Z). Le programme suivant corrige ce problème :

```
glEnable(GL_BLEND);
glEnable(GL_POLYGON_OFFSET_FILL);
glPolygonOffset(0, -1.0);
glBlendFunc(GL_ONE_MINUS_SRC_ALPHA, GL_SRC_ALPHA);
// dessiner
glDisable(GL_BLEND);
glDisable(GL_POLYGON_OFFSET_FILL);
```

on obtient alors le résultat correct suivant:



Notons que dans ce programme nous n'avons pas effectué de tri en Z, par rapport à la position de l’observateur. Si la position de l’observateur change il faut aussi changer le tracé du cube de façon à toujours afficher les faces les plus éloignées en premier, sans quoi une incohérence visuelle peut apparaître.

10. Buffers d’OpenGl, Images et “Rasterization”

10.1 Frame buffer

OpenGl utilise des tampons rectangulaires appelés *frame buffers* pour stocker les informations associées aux pixels : couleur et profondeur en Z. Le mode “double buffering” permet d'éviter de voir des scintillements à l'écran, dus à la non synchronisation entre le rafraichissement de l'écran et le tracé de primitives. Lorsqu'il est activé, il existe deux tampons de couleur, appelés FRONT et BACK. En mode stéréo vision, ces deux buffers sont encore dédoublés en deux buffers chacun: FRONT_LEFT, FRONT_RIGHT et BACK_LEFT, BACK_RIGHT. Seuls les

buffers FRONT sont visibles à l'écran. Par défaut, les écritures ont lieu dans les buffers BACK. Une fois toutes les primitives envoyées et tracées, les buffers peuvent être échangés sans scintillement, càd en veillant à la synchronisation avec le moniteur. On parle de buffer "swapping".

Les buffers de couleur contiennent les composantes couleur Rouge, Vert et Bleu sur 8 bits chacune. Il peut y avoir également une composante *Alpha*. Cette composante alpha est également codée sur 8 bits. Ceci requiert 3, voire 4 octets par pixel. Noter que dans le buffer de couleur, la composante alpha n'est pas nécessaire pour faire de la transparence (le buffer de couleur RVB suffit). Stockée dans le buffer de couleur, la composante alpha peut cependant servir à ajouter des données associées aux surfaces ou à faire des combinaisons complexes avec la commande *glBlendFunc*. Le buffer en Z contient les informations de profondeur codées sur 24 bits généralement. Ce buffer sert à l'élimination des parties cachées.

L'écriture dans un des buffers de couleur peut être forcé avec glDrawBuffer(BUFFER) où BUFFER peut prendre les valeurs GL_NONE, GL_FRONT_LEFT, GL_FRONT_RIGHT, GL_BACK_LEFT, GL_BACK_RIGHT, GL_FRONT, GL_BACK, GL_LEFT, GL_RIGHT, GL_FRONT_AND_BACK.

En plus des buffers de couleur et du Z-buffer, il existe un buffer supplémentaire prévu par OpenGl dans sa première version: le buffer "stencil". Ce buffer peut se limiter à un seul bit plan (1 bit par pixel). Il sert à appliquer un masque lors de l'affichage des objets.

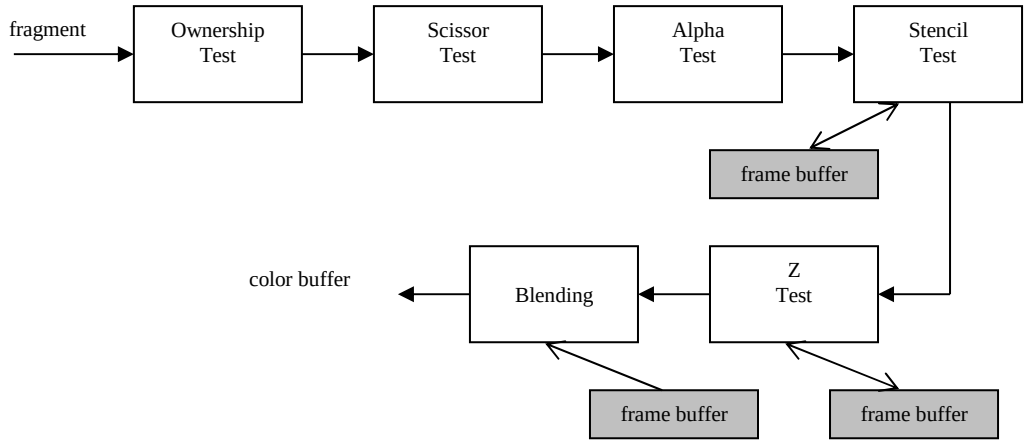
Ensemble tous ces buffers : couleur (gauche, droite, avant, arrière), profondeur et stencil forment le *frame buffer* (le buffer d'écran). En fait le frame buffer peut être considéré comme un empilement de bitplans (couches de bit), où les couches sont regroupées sémantiquement pour former les différents « sous-buffers ».

Depuis la version 3 d'OpenGl il est possible de définir des frame **buffers auxiliaires ou supplémentaires** et de dessiner dans ces buffers les primitives envoyées à la carte graphique, plutôt que de les dessiner dans le frame buffer standard GL_BACK. Ceci a de nombreuses applications que nous verrons en abordant la notion de texture. En particulier, la fonction glGenFramebuffers () permet de générer un buffer auxiliaire et la commande glBindFramebuffer(DRAW_FRAMEBUFFER,glBufferId) permet d'activer l'écriture dans ce buffer (le buffer ayant pour identifiant *glBufferId*).

10.2 Rasterization et fragments

Le processus de rasterization consiste à décomposer les primitives graphiques de haut niveau (points, lignes, triangles) en **fragments** par la méthode de tracer de segment ou de remplissage par balayage de lignes. Un fragment est un pixel auquel on associe des informations supplémentaires (voir chapitre concernant les *textures*). En pratique, OpenGl intègre la rasterization des 5 primitives graphiques suivantes : point, segment, triangle, rectangle et bitmap. Un fragment correspond à une zone carrée de la grille d'affichage à laquelle est assignée une couleur et une profondeur en Z (plus d'autres paramètres que nous verrons plus tard). Un fragment ainsi généré, de coordonnées (x,y), modifiera le pixel lui correspondant dans le frame buffer selon un certain nombre de règles et de conditions.

Le schéma ci-dessous illustre la pipeline à laquelle est soumise chaque fragment au cours du processus de rasterization **après avoir fait appel au fragment shader qui calcule la ou les couleurs (ainsi que la profondeur Z)**:



On voit sur ce schéma que certaines étapes accèdent au frame buffer en lecture (blending), d'autres accèdent en lecture et écriture (stencil et Z).

Initialement le fragment est soumis à un tout premier test en interne appelé le ownership test. Ce test n'a pas d'importance pour nos dessins car il s'agit d'un test d'appartenance au contexte X11 ou windows. En effet, une partie de la fenêtre OpenGl peut être recouverte ou invisible. Ce test est donc géré par le système de fenêtrage de l'OS (donc par GLX et WGL).

Suit un test de découpage. OpenGl permet de définir un cadre rectangulaire en dehors duquel les dessins sont prohibés. Ce test est activé avec glEnable(GL_SCISSOR_TEST). Le cadre rectangulaire est simplement défini par :

```
glScissor(int x, int y, int dx, int dy)
```

Suit un test en alpha. **Ce test n'a aucun rapport ni aucun lien avec la transparence.** Il s'agit juste d'un test qui peut servir à décimer certains polygones. Ce test consiste en effet à comparer la valeur alpha du polygone avec une constante *v*. Le test ainsi que la constante *v* sont définies par :

```
glAlphaFunc(TEST, float v)
```

TEST vaut soit GL_NEVER, GL_ALWAYS, GL_LESS, GL_LEQUAL, GL_GREATER, GL_GEQUAL, GL_NOTEQUAL. Ce mode doit être activé avec glEnable(GL_ALPHA _TEST).

Suit le test du stencil. En fait ce test est un peu semblable à celui du scissor test, sauf que la zone en dehors de laquelle les dessins sont prohibés peut être plus complexe et définie à l'aide d'un certain nombre de bitplans. Nous en détaillerons l'utilisation dans le paragraphe suivant.

Suit le test en Z. Nous avons déjà vu comment ce test est initialisé et activé.

Suit enfin, le blending, que nous avons également déjà décrit.

Il existe un certain nombre d'opérations permettant de gérer les différents buffers d'OpenGl, notamment de les initialiser et de contrôler l'écriture à l'intérieur de ces buffers. Les commandes :

```
glColorMask(bool r, bool v, bool b, bool a)
glDepthMask(bool x)
glStencilMask(int x)
```

permettent d'autoriser ou d'interdire l'écriture dans les buffers concernés. Nous donnerons une explication plus précise pour le stencil buffer dans le paragraphe suivant.

La commande :

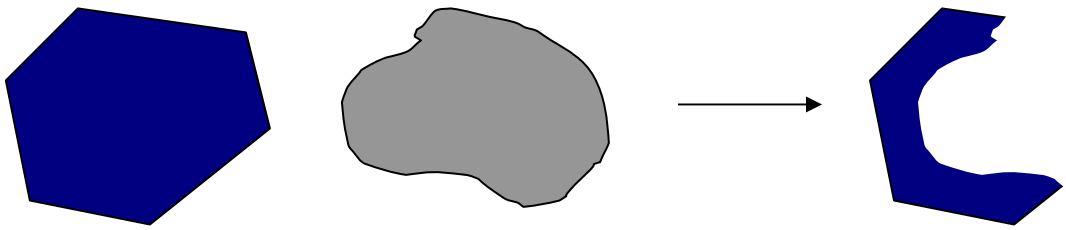
```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT
            GL_ACCUM_BUFFER_BIT )
```

permet d'initialiser les 4 buffers. La valeur d'initialisation est définie à l'aide des commandes :

```
glClearColor(float r, float v, float b, float a)
glClearDepth(float z)   // entre 0 et 1
glClearStencil(int x)
glClearAccum(float r, float v, float b, float a)
```

10.3 Utilisation du stencil buffer

Le stencil buffer permet de moduler l'écriture dans les buffers de couleur et de profondeur en fonction d'un « masque » de bits. La figure ci-dessous illustre un exemple de tracé de polygone modulé par un stencil buffer. Dans cet exemple une partie du polygone n'est pas affichée.



Le stencil buffer peut être composé de 1 à 8 bitplans selon les architectures. Deux commandes permettent de contrôler ce buffer. Il s'agit des commandes :

```
glStencilFunc(TEST, int v, unsigned int mask)
glStencilOp(OP sfail, OP zfail, OP zpass)
```

La première commande permet de définir le test. TEST peut prendre les valeurs : GL_NEVER, GL_ALWAYS, GL_LESS, GL_EQUAL, GL_LEQUAL, GL_GREATER, GL_GEQUAL, GL_NOTEQUAL. *v* représente une valeur constante à laquelle le contenu du stencil buffer est comparé selon TEST. *mask* représente un masque binaire appliqué sous la forme d'un ET logique à la fois à v et au contenu du buffer avant le test comparatif.

La seconde commande permet de définir l'action à appliquer au stencil buffer si un certain nombre de tests passent ou échouent. OP peut prendre les valeurs GL_KEEP, GL_ZERO, GL_REPLACE, GL_INCR, GL_DECR, GL_INVERT. *sfail* représente le cas où le stencil test échoue, *zfail* le cas où le test en Z échoue et *zpass* le cas où le test en Z réussit.

Dans l'exemple suivant :

```
glStencilFunc(GL_EQUAL, 1, 1)
glStencilOp(GL_INVERT, GL_KEEP, GL_INCR)
```

Le stencil test consiste à comparer la valeur du stencil buffer du fragment en question avec 1, après avoir éliminé tous les bits sauf le bit numéro 0 (le bit de parité) car le masque vaut 1. Si le test du stencil échoue alors la valeur du stencil buffer est modifiée en l'inversant bit à bit. Si le test en Z échoue, on ne touche pas au stencil buffer et s'il réussit alors on incrémente la valeur dans le buffer de 1.

Si le stencil test est activé avec *glEnable(GL_STENCIL_TEST)* seuls les fragments dont la valeur correspondante du stencil buffer est paire seront écrits dans le buffer de couleur et le buffer en Z (si l'écriture y est autorisée).

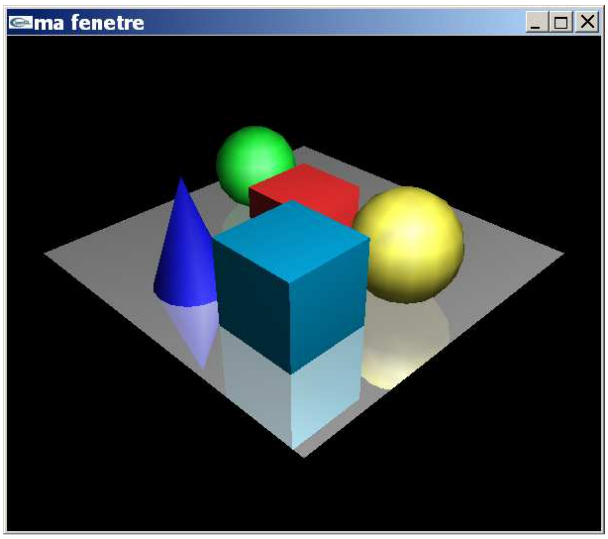
On peut initialiser le stencil buffer avec la commande *glClear* que nous avons vu précédemment.

L'écriture dans le stencil buffer se fait en dessinant des polygones dans le frame buffer et en choisissant la bonne opération OP. Si l'on ne veut écrire un polygone de valeur *v* que dans le stencil buffer pour créer un masque, sans l'écrire dans le buffer de couleur ou dans le buffer en Z, il suffit d'en interdire l'écriture avec glColorMask et glDepthMask, puis de positionner le test à GL_NEVER et l'opération sfail à GL_REPLACE.

Un exemple d'application du stencil buffer peut être la simulation de la réflexion sur le sol de la scène que nous avons affichée dans l'exemple précédent (nous allons considérer l'exemple sans transparence). L'astuce est la suivante : voir une scène dans un miroir planaire revient simplement à voir la scène par symétrie par rapport au plan du miroir. Il suffit donc d'afficher la scène deux fois : une fois de manière inversée mais uniquement dans la limite du plan définie par le sol (c'est là qu'intervient le stencil buffer), puis une seconde fois sans symétrie.

Dans la fonction de dessin nous procédons en quatre étapes : 1) créer le stencil buffer en affichant le plan qui délimite la réflexion, 2) afficher la scène à l'envers en utilisant la matrice GL_MODELVIEW et en activant le stencil test, 3) afficher le plan en imposant les Z du plan (pour écraser ceux de la scène à l'envers) avec une certaine transparence pour moduler le degré de réflexion du sol, 4) afficher le reste de la scène à l'endroit en réactivant le test en Z, mais en désactivant le stencil test.

Note : il faut penser à inverser également la position des sources. Voici le résultat que l'on obtient à l'écran. L'impression de réflexion est bien rendue.

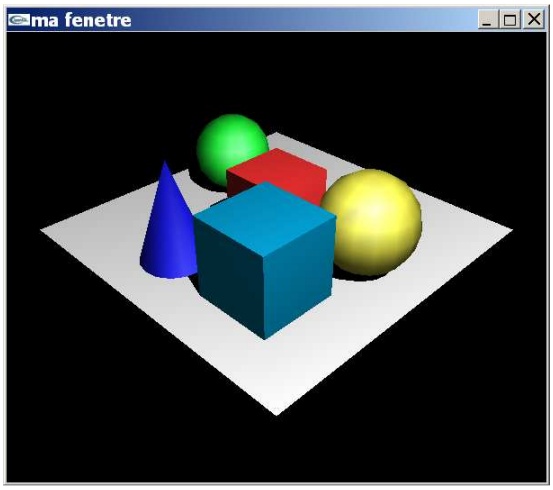


Notons que cette approche pour calculer la réflexion reste extrêmement élémentaire : elle ne fonctionne qu'avec une surface plane. Les surfaces courbent ne peuvent pas être traitées, ni un nombre plus important de surfaces même planes (en raison du manque de réflexions multiples).

En dehors de la réflexion, le stencil buffer peut également être utilisé pour créer des ombres sur le sol. En effet, une ombre correspond à une zone noire, donc une zone dans laquelle les dessins ne sont pas effectués. Il suffit de projeter les objets sur le plan selon la direction du vecteur de lumière pour créer le stencil buffer. La matrice de projection sur un plan de normale N(Nx, Ny, Nz) passant par l'origine, selon une direction L(Lx,Ly,Lz) est la suivante :

$$M=\begin{pmatrix} c-LxPx & -LxPy & -LxPz & -LxPw \\ -LyPx & c-LyPy & -LyPz & -LyPw \\ -LzPx & -LzPy & c-LzPz & -LzPw \\ -LwPx & -LwPy & -LwPz & c-LwPw \end{pmatrix}$$

On obtient le résultat graphique suivant :

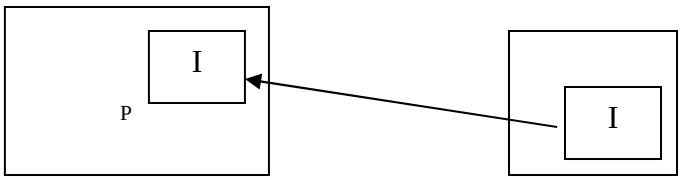


Notons que cette approche pour calculer les ombres n'est pas une approche générale. Elle ne fonctionne que sur un plan. Les ombres ne sont pas projetées sur les objets entre eux (cube sur cône par exemple) ce qui limite grandement ce type d'approche.

10.5 Copie depuis/vers/au sein du frame buffer

OpenGl prévoit de faire des transferts de rectangles de pixels : soit de la mémoire centrale vers le frame buffer (ce qui permet d'afficher par exemple une image chargée d'un fichier), soit l'inverse, c'est à dire du frame buffer vers la mémoire, soit de faire une copie de pixels directement au sein du frame buffer (recopie d'un rectangle vers une autre position). Les trois commandes concernées sont : *glDrawPixels*, *glReadPixels* et *glCopyPixels*.

Ces commandes dépendent d'un contexte de transfert qui est défini avec : *glPixelStore*, *glPixelTransfer* et *glPixelMap*. La figure ci-dessous illustre le transfert de la mémoire vers le framebuffer avec glDrawPixels :



Le coin P de l'image I en bas à droite dans le framebuffer est donné par la position courante du raster. Cette position peut être fixée par glRasterPos. Cette commande réagit comme glVertex, c'est à dire qu'elle existe avec 2, 3 ou 4 coordonnées en float, int, short, etc. Par exemple : glRasterPos3f(x, y, z), fixe la position du raster après avoir subi les mêmes transformations matricielles que le ferait glVertex3f(x,y,z).

La copie de la mémoire vers le frame buffer se fait avec :

```
glDrawPixels(int long, int larg, FORMAT, TYPE, void *pixels)
```

(long, larg) représente la taille en X et Y de la zone I à copier.

FORMAT indique dans quel buffer du frame buffer la recopie se fera. FORMAT peut prendre les valeurs suivantes :

- *GL_RGB ou GL_RGBA* : copie dans le buffer de couleur les composantes rouge, vert, bleue et alpha. Chaque pixel correspond à un paquet de trois ou quatre données de type TYPE. Avant l'écriture les valeurs sont multipliées respectivement par GL_RED_SCALE, GL_GREEN_SCALE, GL_BLUE_SCALE, GL_ALPHA_ SCALE et additionnées à une constante GL_RED_BIAS, GL_GREEN_BIAS, GL_BLUE_BIAS, GL_ALPHA_BIAS.

  Ces constantes sont définies avec `glPixelTransferf(CONST, float v)` où CONST prend les valeurs GL_RED_SCALE, GL_GREEN_SCALE, GL_BLUE_SCALE, GL_ALPHA_ SCALE, GL_RED_BIAS, GL_GREEN_BIAS, GL_BLUE_BIAS, GL_ALPHA_BIAS.

- *GL_DEPTH_COMPONENT* : copie dans le Z-buffer. Chaque pixel correspond à une valeur de type TYPE. Avant l'écriture les valeurs sont multipliées par GL_DEPTH_SCALE et additionnées à une constante GL_DEPTH_BIAS. Ces constantes sont également définies avec `glPixelTransferf(CONST, float v).`

- *GL_STENCIL_INDEX* : copie dans le stencil buffer. Chaque pixel correspond à une valeur de type TYPE. Avant l'écriture les valeurs sont décalées de GL_INDEX_SHIFT bits vers la gauche (si négatif vers la droite) et additionnées à une constante GL_INDEX_OFFSET. Ces constantes sont également définies avec `glPixelTransferi(CONST, int v).`

TYPE indique le format des données de I pointées par *pixels*. TYPE peut prendre les valeurs GL_UNSIGNED_BYTE (1 octet), GL_BYTE, GL_UNSIGNED_SHORT (2 octets), GL_SHORT, GL_ UNSIGNED_INT (4 octets), GL _INT (4 octets), GL_FLOAT (4 octets), GL_BITMAP (1 bit).

La commande glPixelStore permet de préciser la manière dont ces données sont stockées dans la mémoire (en dehors du type donné par TYPE) :

```
glPixelStorei(PARAM, int v)
```

PARAM prend les valeurs : GL_UNPACK_ALIGNMENT, avec v=1, 2, 4 ou 8. Ceci indique que les données pointées par *pixels* sont alignés sur un octet, un mot court, un mot long, etc. ; GL_UNPACK_ROW_LENGTH indique la taille globale de l'image en mémoire dont I est issue (souvent cette valeur vaut 0, ce qui signifie que l'image est contiguë en mémoire) ; GL_UNPACK_SKIP_ROWS et GL_UNPACK_SKIP_PIXELS indiquent la position de I en mémoire.

La copie du frame buffer vers la mémoire se fait avec :

```
glReadPixels(int px, int py, int long, int larg, FORMAT, TYPE, void
*pixels)
```

(px, py) indique la position dans la fenêtre du frame buffer. Les autres paramètres sont les mêmes. Note : pour `glPixelStorei(PARAM, int v)` les constantes ne sont plus GL_UNPACK_x, mais GL_PACK_x, par exemple GL_PACK_ALIGNMENT au lieu de GL_UNPACK_ALIGNMENT.

La copie d'une zone du frame buffer vers une autre zone se fait avec :

```
glCopyPixels(int px, int py, int long, int larg, TYPE)
```

Cette commande copie une sous image qui se trouve à la position px, py, de taille long et larg, vers la position courante du raster. TYPE indique le buffer et prend les valeurs GL_COLOR, GL_DEPTH ou GL_STENCIL.

La commande glPixelMap permet d'utiliser une table pour moduler les valeurs avant leur recopie (une indirection) soit vers la mémoire soit vers le frame buffer. Nous ne détaillons pas ici cette commande qui trouve sont utilité plus dans le mode couleur indexée.

Enfin, la commande `glPixelZoom(float zx, float zy)` permet de faire des agrandissements / rétrécissement de la zone copiée (uniquement avec glDrawPixels et glCopyPixels). Par défaut (zx, zy)=(1.0, 1.0).