



# REAL-TIME RENDERING PIPELINE

## I - Vertex Operations

La scène 3D est décrite par comme étant un ensemble de triangles i.e. :

- Une collect de vertex, de points 3D.
- Une collect de face / triangle i.e. de connectivité entre les vertex.
- Une collect d'attributs, chaque vertex peut avoir une normale, une texture coordinate ...

Les vertex du triangular mesh sont décrits dans un système de coordonnées locales, on parle d'object space / objet coordinates / model space.

des objets 3D sont représentés par un triangular mesh, car le triangle à certaines propriétés que n'ont pas les autres polygones. En effet :

- Tout triangle dans l'espace est plat, ces 3 points définissent un plan. Cela garantit que chaque face / triangle possède une unique normale, facilitant le back-face culling. (4 vertex pour projeter).
- Il est simple d'interpoler sur le triangle des valeurs définies sur les sommets. C'est l'interpolation barycentrique. Pour un polygone quelconque c'est moins évident, en particulier lorsqu'ils ne sont pas convexes (ex quad).
- Tout triangle dans l'espace est convexe, cela permet de pouvoir facilement tester lors de la rasterisation si un fragment appartient ou non au triangle projeté.

### I.1 - Vertex processing :

Chaque vertex sont traité indépendamment et sont transformés de l'objet space au clip space. Le clip space est un espace 3D obtenu après application de la model matrice, puis de la vieu matrice et enfin de la projekt matrice. Ces opérations sont réalisées par le vertex-shader. En plus de spécifier la position du vertex dans le clip space, le vertex shader peut output d'autres valeurs associées à la vertex afin de le rendre accessible (par interpolation) à chaque fragment.

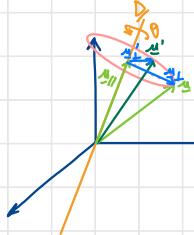
#### \* De l'object space au world space :

Un triangular mesh peut être positionné dans l'espace en appliquant une transformation affine représentée par matrice  $4 \times 4$ . Elle correspond habituellement à l'application successive d'une matrice de scaling, puis de rotation et enfin de translation. Par exemple pour on souhaite écheler le mesh d'un facteur 2, puis lui appliquer une rotation de  $\theta$  rad autour de l'axe  $\Delta$ , enfin on souhaite le traduire à une position  $t_0$  dans le world space: ( $\|t_0\| \| \Delta \| = 1$ )



- scaling :  $\begin{pmatrix} x \\ y \\ z \end{pmatrix} \mapsto \begin{pmatrix} \lambda x \\ \lambda y \\ \lambda z \end{pmatrix}$  d'où matrice de transform' est  $S = \begin{pmatrix} \lambda I_3 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & 1 \end{pmatrix}$

- rotation :



Donc  $u = u_{\parallel} + u_{\perp} \mapsto u' = u_{\parallel} + u'_{\perp}$ . Or  $u_{\parallel} = "projeté de u sur \Delta (unitaire)" = \langle u | \Delta \rangle \Delta$

Alors  $u_{\perp} = u - u_{\parallel} = u - \langle u | \Delta \rangle \Delta$

Pour obtenir  $u'_{\perp}$  on va utiliser une base du plan contenant le diagonal de rotation.

Ainsi  $v = \Delta \times u_{\perp}$  (de norme  $\|u_{\perp}\|$ ) et  $u'_{\perp} = \cos\theta u_{\perp} + \sin\theta v = \cos\theta u_{\perp} + \sin\theta (\Delta \times u_{\perp})$

$$\begin{aligned} \text{Finalement : } u' &= u_{\parallel} + u'_{\perp} = \langle u | \Delta \rangle \Delta + \cos\theta u_{\perp} + \sin\theta (\Delta \times u_{\perp}) \\ &= \langle u | \Delta \rangle \Delta + \cos\theta (u - \langle u | \Delta \rangle \Delta) + \sin\theta (\Delta \times (u - \langle u | \Delta \rangle \Delta)) \\ &= (1 - \cos\theta) \langle u | \Delta \rangle \Delta + \cos\theta u + \sin\theta (\Delta \times u) \quad \rightarrow \text{mettre sous forme R.u} \\ &= (1 - \cos\theta) \underbrace{\left( \Delta^T u \right)}_{\in \mathbb{R}} \Delta + \cos\theta I_3 u + \sin\theta [\Delta]_x u \end{aligned}$$

$$\begin{aligned} &= (1 - \cos\theta) (\Delta \Delta^T) u + \cos\theta I_3 u + \sin\theta [\Delta]_x u \quad \text{avec } \left[ \begin{pmatrix} x \\ y \\ z \end{pmatrix} \right]_x = \begin{pmatrix} 0 & -z & y \\ z & 0 & -x \\ -y & x & 0 \end{pmatrix} \\ &= \left[ (1 - \cos\theta) (\Delta \Delta^T) + \cos\theta I_3 + \sin\theta [\Delta]_x \right] u, \text{ pour } c = \cos\theta \text{ et } s = \sin\theta \text{ et } \Delta = \begin{pmatrix} x & y & z \\ y & z & x \\ z & x & y \end{pmatrix} \end{aligned}$$

$$\begin{aligned} &= \left[ (1 - c) \begin{bmatrix} x^2 & xy & xz \\ xy & y^2 & yz \\ xz & yz & z^2 \end{bmatrix} + \begin{bmatrix} c & 0 & 0 \\ 0 & c & 0 \\ 0 & 0 & c \end{bmatrix} + s \begin{bmatrix} 0 & -z & y \\ z & 0 & -x \\ -y & x & 0 \end{bmatrix} \right] u \\ &= \begin{bmatrix} (1 - c)x^2 + c & (1 - c)xy - sz & (1 - c)xz + sy \\ (1 - c)xy + sz & (1 - c)y^2 + c & (1 - c)yz - sx \\ (1 - c)xz - sy & (1 - c)yz + sx & (1 - c)z^2 + c \end{bmatrix} u = \hat{R} u \end{aligned}$$

D'où matrice de transform'  $R = \begin{bmatrix} \hat{R} & 0 \\ 0 & 1 \end{bmatrix}$

- translation :  $\begin{pmatrix} x \\ y \\ z \end{pmatrix} \mapsto \begin{pmatrix} x+t_x \\ y+t_y \\ z+t_z \end{pmatrix}$  d'où matrice de transform'  $T = \begin{bmatrix} I_3 & t \\ 0 & 1 \end{bmatrix}$

Finalement la world coordinate de  $u$  en object coordinate est  $u_w = TRS u$  avec  $u = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$  et  $M_{\text{world}} = TRS$

On peut alors chercher à exprimer les normales dans le world space. Considérons une normale  $n$ , on cherche à trouver son expression en world space. Pour cela on raisonne sur le plan de normale  $n$  et comment  $M_{\text{world}}$  modifie le plan.

Ce plan est  $\pi = \{u \mid t_y n_x + t_z n_y = 0\}$ , or  $u$  transformé en  $M_w u$ , on remarque  $0 = u^T \pi = u^T M_w^{-1} M_w u = (M_w^{-1} u)^T M_w u$ .

D'où  $M_w u$  appartient au plan  $M_m^{-1} \left( \frac{n}{1} \right)$  donc la normale est donnée par  $M_m^{-1} \left( \frac{n}{1} \right)$  reste à normaliser.

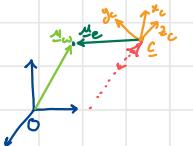
Une autre méthode est juste d'appliquer  $\hat{R}$  à  $n$ .

P : Ici on suppose que shading en world space. Si c'était en eye space on remplace  $M_w$  par  $M_v M_w$ , mais dans ce cas il faut aussi exprimer points des lights en eye space ...

## \* Du world space à l'eye space :

L'eye space s'appelle aussi camera space, c'est l'espace 3D depuis le point de vue de la caméra. Ainsi le world space à l'eye space correspond simplement à un changement de base. On aura la convention que la caméra regarde directement -z :

On décrit la caméra par sa position dans le world space  $\underline{u}_w$ , et une base orthonormale  $(x_c, y_c, z_c)$ .



$$\text{Alors } \underline{u}_e = (\underline{u}_w - \underline{c}) \begin{vmatrix} & \\ & (x_c, y_c, z_c) \end{vmatrix} = \begin{bmatrix} \underline{x}_c^T \\ \underline{y}_c^T \\ \underline{z}_c^T \\ 0 \end{bmatrix} (\underline{u}_w - \underline{c}) = R^T \underline{u}_w - R^T \underline{c} \text{ ainsi c'est une rotation de } R^T \text{ et translation de } -R^T \underline{c}$$

"R<sup>T</sup> : projection  $\underline{u}_w - \underline{c}$  sur base  $(x_c, y_c, z_c)$ "

$$\text{D'où : } M_{\text{view}} = \begin{bmatrix} \underline{x}_c^T & -\underline{x}_c^T \underline{c} \\ \underline{y}_c^T & -\underline{y}_c^T \underline{c} \\ \underline{z}_c^T & -\underline{z}_c^T \underline{c} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$(x_c, y_c, z_c)$  peuvent être décrit plus simplement par  $\underline{c}$  et un target point  $\underline{f}$  fixé par la caméra et une direction  $\underline{s}$  qui oriente la caméra :

$$z_c = \frac{\underline{c} - \underline{f}}{\|\underline{c} - \underline{f}\|} \quad x_c = \underline{u}_p \times z_c \quad y_c = z_c \times x_c \quad \text{avec } \|\underline{u}_p\|=1$$

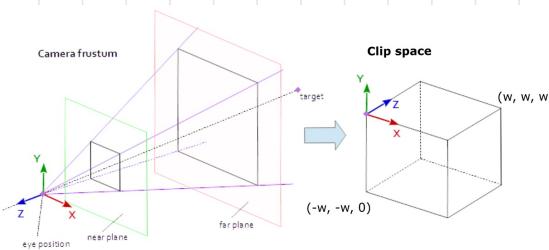
On peut alors combiner les deux matrices  $M_{\text{modelview}} = M_{\text{view}} M_{\text{model}}$ , et  $\underline{u}_e = M_{\text{modelview}} \underline{u}$ .

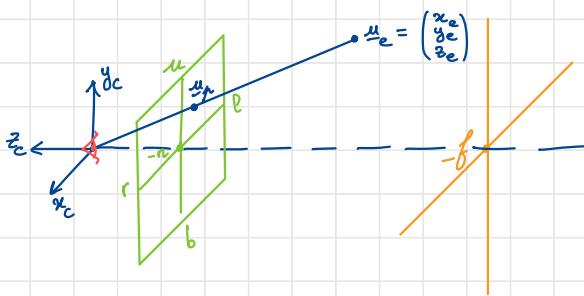
## \* Du eye space au clip space :

Le clip space est l'espace obtenu après application de la matrice de projection au eye space. C'est un espace 3D qui est tel qu'après perspective division les coordonnées soit dans un volume normalisé  $[-1, 1] \times [-1, 1] \times [-1, 1]$  (OpenGL) ou  $[-1, 1] \times [-1, 1] \times [0, 1]$  (Vulkan) appeler Normal Device Coordinate.

Le clip space est en coordonnées homogènes. Il s'appelle ainsi car les vecteurs  $\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix}$  qui ne vérifient pas  $-w_c \leq x_c \leq w_c$  and  $-w_c \leq y_c \leq w_c$  and  $-w_c \leq z_c \leq w_c$  sont clippés automatiquement, cela correspond aux vecteurs en dehors du NDC.

Pour construire la matrice de projection, on considère un viewing frustum (champ de vision) défini par son near plane (profondeur n), far plane (profondeur f), son es largeur [l, r] et sa hauteur [b, t] :





la perspective project transforme  $u_e$  en  $u_p$ , on remarque  $u_p = \lambda u_e$ , or sur le word  $z$  on a :  
 $-n = \lambda z_e$  donc  $\lambda = -\frac{n}{z_e}$  et  $u_p = -\frac{n}{z_e} u_e$   
Or on souhaite que la coordonnée  $x, y$  de  $u_p$  soit dans  $[-w, w] \times [-h, h]$  soit mises à  $[-1, 1]^2$  pour être en NDC ( $u_n$ )

D'où  $u_{n,x} = \frac{2}{r-l} (u_{p,x} - l) - 1 = \frac{2}{r-l} u_{p,x} - \frac{r+l}{r-l} = -\frac{2n}{(r-l)z_e} x_e - \frac{r+l}{r-l}$  à mettre sous forme homogène i.e.  $\frac{1}{w} u_n^T$ ,  $u_{n,x}$

$$u_{n,x} = \frac{1}{-z_e} \left[ \frac{2n}{(r-l)} x_e + \frac{r+l}{r-l} z_e \right], \text{ on factorise } -\frac{1}{z_e} \text{ car } -z_e \text{ profondeur positive.}$$

Idem  $u_{n,y} = \frac{1}{-z_e} \left[ \frac{2n}{(t-b)} y_e + \frac{t+b}{t-b} z_e \right]$

Enfin on souhaite que  $u_{n,z}$  (i.e. la coord  $z$  en NDC)  $\in [-1, 1]$ . Pour cela doit transformer  $z_e$  tel qu'après division par  $-z_e$ ,  $-n$  mape sur  $-1$  et  $-f$  mape sur  $1$ . On cherche donc à etablir  $\begin{cases} \frac{1}{z_e} (az+b) (z=-n) = -1 \\ \frac{1}{z_e} (az+b) (z=-f) = 1 \end{cases}$   
i.e.  $\begin{cases} \frac{1}{z_e} [-an+b] = -1 \\ -af+b = 1 \end{cases}$  i.e.  $\begin{cases} b = n+an \\ af-an = n+f \end{cases}$  (1)-(2)

$$\begin{cases} b = n+an \\ a = \frac{n+f}{f-n} \end{cases}$$

D'où  $u_{n,z} = \frac{1}{z_e} \left[ \frac{n+f}{f-n} z_e + \frac{2f}{f-n} \right] = \frac{1}{-z_e} \left[ \frac{-(f+n)}{f-n} z_e - \frac{2f}{f-n} \right]$

Finalement :

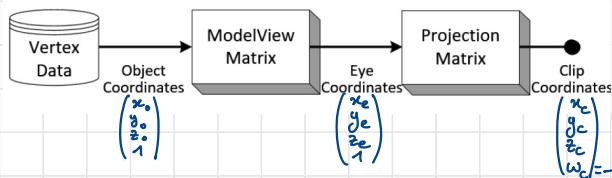
$$u_n = -\frac{1}{z_e} \left( \begin{array}{c} \frac{2n}{(r-l)} x_e + \frac{r+l}{r-l} z_e \\ \frac{n}{(t-b)} y_e + \frac{t+b}{t-b} z_e \\ \frac{-(f+n)}{f-n} z_e - \frac{2f}{f-n} \end{array} \right)$$

$$\left( \begin{array}{c} \frac{2n}{(r-l)} x_e + \frac{r+l}{r-l} z_e \\ \frac{2n}{(t-b)} y_e + \frac{t+b}{t-b} z_e \\ \frac{-(f+n)}{f-n} z_e - \frac{2f}{f-n} \\ -z_e \end{array} \right)$$

I.e.  $P = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2f}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$

$\Rightarrow$  on stocke la profondeur du point à la caméra.

\* Recap : Vertex Shader :



\* Etapes optionnelles :

Ensuite se trouve une étape optionnelle la tessellation qui permet d'augmenter le niveau de détails d'un mesh en le subdivisant en triangle plus petit. Cette étape est décrite à l'aide des tessellate shaders.

Une autre étape optionnelle est le geometry shader mais bien sûr peu utilisée.

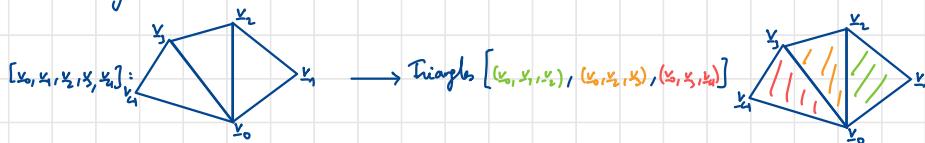
## I.2 - Vertex post-processing:

De nouvelles opérations sont réalisées sur les vertex et primitive (triangle), on parle de vertex post-processing, cette partie du pipeline n'est pas programmable.

\* Primitive Assembly :

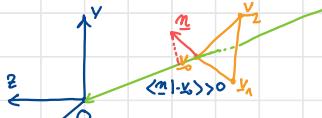
Cette étape décompose les vertex (en clip coordinate) en primitive de base (point, ligne, triangle) selon leur organisation (ex. triangle, triangle fan, triangle strip...).

Par exemple, considérons 5 vertex organisés en triangle fan, la primitive assembly va alors décomposer ces vertex en 3 triangles:



- Back-face culling : Ensuite pour chaque triangle obtenu précédemment, ceux qui ne font pas face à la caméra sont écartés (étape du rendering pipeline). Cela permet d'éviter de traiter de triangles qui n'auront pas d'influence sur l'image finale obtenue (appare opaque).

Pour obtenir le test qui permet de savoir si un triangle en clip coordinate fait face ou non à la caméra on raisonne en eye-coordinate :  $v_i = \begin{pmatrix} x_i \\ y_i \\ z_i \\ 1 \end{pmatrix} = \begin{pmatrix} x_i \\ y_i \\ z_i \\ 1 \end{pmatrix} \in \mathbb{R}^3$ .



L'eye space étant centré sur la caméra, elle se trouve en  $O = (0)$ . Alors, pour tester si le triangle fait face on calcule l'angle de  $\langle n | O - v \rangle$  i.e.

$$\langle n | -v \rangle, \text{ avec } n \text{ la normale du triangle i.e. } n = (v_1 - v_0) \times (v_2 - v_0)$$

Si  $\langle n | -v \rangle > 0$  fait face. L'idée pour faire passer ce test en clip space est de transformer  $\langle n | -v \rangle$  en un déterminant, puis de trouver 11 matrice  $4 \times 4$  dépendant de  $v_0, v_1, v_2$  avec le déterminant pour ensuite pouvoir lui appliquer la matrice de projection P pour passer de l'eye space au clip space et donc exprimer constitut en forme des coordonnées clip space !

On remarque que  $\langle \underline{v}_0 | \underline{n} \rangle = \langle -\underline{v}_0 | (\underline{v}_1 - \underline{v}_0) \times (\underline{v}_2 - \underline{v}_0) \rangle$  = "volume du parallépipède engendré par  $-\underline{v}_0, \underline{v}_1 - \underline{v}_0, \underline{v}_2 - \underline{v}_0$ "  
 $= \det([-\underline{v}_0 | \underline{v}_1 - \underline{v}_0 | \underline{v}_2 - \underline{v}_0])$

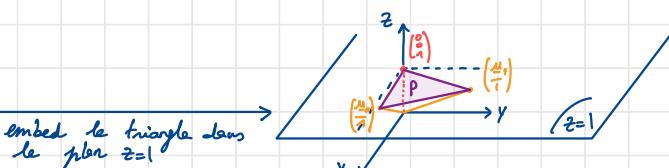
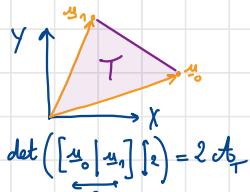
On peut simplifier ce déterminant car  $\det(\underline{u}, \underline{v}) = \det(\underline{u}, \underline{v} + \lambda \underline{u} - \lambda \underline{u}) = \det(\underline{u}, \underline{v} + \lambda \underline{u}) + \underbrace{\det(\underline{u}, -\lambda \underline{u})}_{\text{équivaut à 1ère colonne}} \quad \text{"0 car colonne nulle"}$   
 $= \det(\underline{u}, \underline{v} + \lambda \underline{u})$

Ainsi pour retirer la 1<sup>e</sup> colonne à la 2<sup>e</sup> et 3<sup>e</sup> sans modifier le det:  $\langle -\underline{v}_0 | \underline{n} \rangle = \det([-\underline{v}_0 | \underline{v}_1 | \underline{v}_2]) = -\det([\underline{v}_0 | \underline{v}_1 | \underline{v}_2]) = -d \quad \text{l'aire moins}$

Pour pouvoir appliquer P (la matrice de project°), on passe en coord homogène.

Posons  $M = \left[ \begin{array}{c|ccc} \underline{x}_0 & \underline{y}_1 & \underline{z}_1 & 0 \\ \hline 1 & 1 & 1 & 1 \end{array} \right] = \left[ \begin{array}{cccc} \underline{x}_0 & \underline{y}_1 & \underline{z}_1 & 0 \\ \underline{y}_0 & \underline{y}_1 & \underline{z}_1 & 0 \\ \underline{z}_0 & \underline{y}_1 & \underline{z}_1 & 0 \\ 1 & 1 & 1 & 1 \end{array} \right]$  (en eye space), on va alors montrer que M a pour déterminant d!

↪ Par analogie avec la 2D:



$$\det \left( \begin{bmatrix} \underline{u}_0 & \underline{y}_1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \right)_3 = 6 \frac{1}{\rho} = 6 \cdot \frac{1}{3} \text{ hauteur} \cdot \frac{1}{\rho} \text{ base} \\ = 2 \cdot \det_{\text{triangle}}(0, \underline{u}_0, \underline{u}_1) \\ = \det_T = \det \left( \begin{bmatrix} \underline{u}_0 & \underline{u}_1 & 1 \end{bmatrix} \right)$$

↪ Par opérat° sur le déterminant:

$$\text{En développant l'à la dernière colonne de } M: \det(M) = \left[ \begin{array}{cccc} \underline{x}_0^+ & \underline{y}_1^- & \underline{z}_1^+ & 0^- \\ \underline{y}_0^- & \underline{y}_1^+ & \underline{z}_1^- & 0^+ \\ \underline{z}_0^+ & \underline{y}_1^- & \underline{z}_1^+ & 0^- \\ 1^- & 1^+ & 1^- & 1^+ \end{array} \right] = -0 \det(*) + \bullet \det(*) - 0 \det(*) \\ + \bullet \det \left( \begin{bmatrix} \underline{x}_0 & \underline{y}_1 & \underline{z}_1 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix} \right) \\ = \det([\underline{v}_0, \underline{v}_1, \underline{v}_2]) = d$$

On peut alors appliquer la matrice de project° à M pour passer en clip space:

$$PM = P \left[ \begin{array}{c|ccc} \underline{x}_0^c & \underline{y}_1^c & \underline{z}_1^c & 0 \\ \hline w_0^c & w_1^c & w_2^c & 0 \end{array} \right] = \left[ \begin{array}{c|c|c|c} P\underline{x}_0^c & P\underline{y}_1^c & P\underline{z}_1^c & 0 \\ \hline 0 & 0 & 0 & 0 \end{array} \right] = \left[ \begin{array}{c|c|c|c} \underline{x}_0^c & \underline{y}_1^c & \underline{z}_1^c & (-2fn/(f-n)) \\ \hline w_0^c & w_1^c & w_2^c & 0 \end{array} \right] \text{ avec } \underline{v}_i^c = \begin{pmatrix} \underline{v}_i \\ w_i^c \end{pmatrix} \quad (\text{clip space})$$

$$\text{Alors on a en développant l'à la dernière colonne } \det(PM) = - \frac{-2fn}{f-n} \det \left[ \begin{array}{ccc|c} \underline{x}_0^c & \underline{x}_1^c & \underline{x}_2^c & 0 \\ \underline{y}_0^c & \underline{y}_1^c & \underline{y}_2^c & 0 \\ \underline{w}_0^c & \underline{w}_1^c & \underline{w}_2^c & 0 \end{array} \right]$$

$$= \frac{2fn}{f-n} \det \left[ \begin{array}{ccc|c} \underline{x}_0^c & \underline{x}_1^c & \underline{x}_2^c & 0 \\ \underline{y}_0^c & \underline{y}_1^c & \underline{y}_2^c & 0 \\ \underline{w}_0^c & \underline{w}_1^c & \underline{w}_2^c & 0 \end{array} \right]$$

De plus  $\det(PM) = \det(P) \det(M) = \det(P) d$

$$\text{Donc } \det(P) d = \frac{2fn}{f-n} \det \left[ \begin{array}{ccc|c} \underline{x}_0^c & \underline{x}_1^c & \underline{x}_2^c & 0 \\ \underline{y}_0^c & \underline{y}_1^c & \underline{y}_2^c & 0 \\ \underline{w}_0^c & \underline{w}_1^c & \underline{w}_2^c & 0 \end{array} \right] \text{ et } d = \frac{2fn}{(f-n)\det(P)} \det \left[ \begin{array}{ccc|c} \underline{x}_0^c & \underline{x}_1^c & \underline{x}_2^c & 0 \\ \underline{y}_0^c & \underline{y}_1^c & \underline{y}_2^c & 0 \\ \underline{w}_0^c & \underline{w}_1^c & \underline{w}_2^c & 0 \end{array} \right]$$

On ne s'intéresse qu'au signe de  $d$ , on a donc besoin de connaître le signe de  $\det(P)$ :

$$\det(P) = \det \begin{bmatrix} \frac{2n}{r-e} & -0 & \frac{r+l}{r-e} & -0 \\ -0 & \frac{2n}{t-b} & -\frac{t+b}{t-b} & +0 \\ +0 & -0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ -0 & +0 & -1 & 0 \end{bmatrix} \quad \text{dans la dernière ligne}$$

$$= -(-1) \det \begin{bmatrix} \frac{2n}{r-e} & 0 & 0 \\ 0 & \frac{2n}{t-b} & 0 \\ 0 & 0 & -\frac{2fn}{f-n} \end{bmatrix}$$

$$= \left( \frac{2n}{r-e} \right) \left( \frac{2n}{t-b} \right) \left( \frac{-2fn}{f-n} \right) \quad \text{comme } r>e, t>b \text{ et } f>n \text{ le signe de } \det(P) \text{ est } 0 !$$

$$\text{D'où } \text{signe}(d) = \text{signe} \left( \frac{2fn}{(f-n)\det(P)} \right) = \underbrace{\text{signe} \left( \frac{2fn}{f-n} \right)}_{\oplus} \underbrace{\text{signe}(\det(P))}_{0} \text{ signe} \left( \det \begin{bmatrix} x_0^c & x_1^c & x_2^c \\ y_0^c & y_1^c & y_2^c \\ w_0^c & w_1^c & w_2^c \end{bmatrix} \right)$$

$$\text{signe}(d) = - \text{signe} \left( \det \begin{bmatrix} x_0^c & x_1^c & x_2^c \\ y_0^c & y_1^c & y_2^c \\ w_0^c & w_1^c & w_2^c \end{bmatrix} \right)$$

$$\text{Finalement : } \text{signe}(\langle v_0^c | n \rangle) = -d = \text{signe} \left( \det \begin{bmatrix} x_0^c & x_1^c & x_2^c \\ y_0^c & y_1^c & y_2^c \\ w_0^c & w_1^c & w_2^c \end{bmatrix} \right)$$

D'où pour déterminer si triangle  $(v_0^c, v_1^c, v_2^c)$  avec  $v_i^c = \begin{pmatrix} x_i^c \\ y_i^c \\ z_i^c \\ w_i^c \end{pmatrix} \in \mathbb{R}^4$  (clip space) fait face à la caméra il faut :

$$\bullet \text{ Calculer } \det \begin{bmatrix} x_0^c & x_1^c & x_2^c \\ y_0^c & y_1^c & y_2^c \\ w_0^c & w_1^c & w_2^c \end{bmatrix} = \begin{pmatrix} x_0^c \\ y_0^c \\ w_0^c \end{pmatrix} \times \begin{pmatrix} x_1^c \\ y_1^c \\ w_1^c \end{pmatrix} \cdot \begin{pmatrix} x_2^c \\ y_2^c \\ w_2^c \end{pmatrix}$$

$\bullet$  Si  $qf > 0$  fait face, sinon il faut cull le triangle!

A la fin de cet espace on obtient une liste de triangles faisant face à la caméra!

### \* Clipping :

Pour chaque triangle faisant face à la caméra, il faut encore discard ceux en dehors du champ de vision (viewing frustum), permet de réduire le nombre d'opérat à faire dans la suite (nécessite de traiter triangle non visible).

Une vertex  $v^c = \begin{pmatrix} x^c \\ y^c \\ z^c \\ w^c \end{pmatrix}$  du triangle est à l'intérieur du viewing frustum  $-w_c \leq x_c \leq w_c$  -

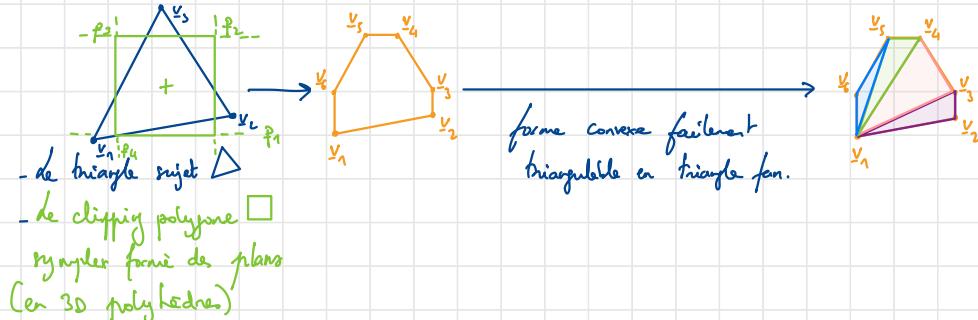
$$-w_c \leq y_c \leq w_c$$

$$-w_c \leq z_c \leq w_c$$

Ce qui correspond à être à l'intérieur du simplexe  $S = \{x \in \mathbb{R}^4 / H_i \leq 0, i \in \{1, 6\}, \vec{z}^T \vec{x} \geq 0\}$  défini par 6 plans :

- plan left:  $x \geq -w$  i.e.  $x + w \geq 0$  i.e.  $(1 \ 0 \ 0)^T \cdot \underline{x} \geq 0$  donc  $p_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$
- plan right:  $x \leq w$  i.e.  $-x + w \geq 0$  i.e.  $(-1 \ 0 \ 0)^T \cdot \underline{x} \geq 0$  donc  $p_2 = \begin{pmatrix} -1 \\ 0 \\ 0 \end{pmatrix}$
- Idem bottom:  $y \geq -w$  i.e.  $p_3 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$ ; top:  $y \leq w$  i.e.  $p_4 = \begin{pmatrix} 0 \\ -1 \\ 0 \end{pmatrix}$
- now:  $z \geq -w$  i.e.  $p_5 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ ; far:  $z \leq w$  i.e.  $p_6 = \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}$

Par conséquent un triangle peut être à cheval sur plusieurs plans. Dans ce cas, il faut clip le triangle i.e. calculer le polygone résultant du découpage du triangle par le plan et trianguler le polygone.  
 Etant donné que le polygone convexe (symples formé plan du triangle et du fronton) on peut facilement le trianguler en triangle fan.

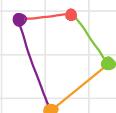


Pour réaliser cette opérat° on décrit le polygone sujet par une liste ordonnée de ses vertex et le clipping polyédre par un simplex donc collection de plans. On va appliquer l'algorithme de Sutherland - Hodgman :

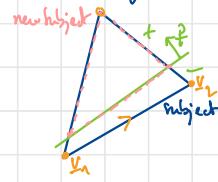
Input: de polygone sujet  $(v_1, \dots, v_n)$ . Ici un triangle  $(v_1, v_2, v_3)$  ordonné sans tripls (en coord homogène)  
 de clipping simplex décrit par une collect de plan  $(P_i = \{\underline{x} \in \mathbb{R}^4 / p_i^T \underline{x} = 0\})_{1 \leq i \leq \#P}$ .

L'intérieur du simplex étant donné par  $S = \{\underline{x} \in \mathbb{R}^4 / \forall i \in \llbracket 1, \#P \rrbracket, p_i^T \underline{x} \geq 0\}$ , intérieur du plan  $\underline{x} \in \mathbb{R}^3 \neq 0$

Algorithme: d'algorithme consiste à itérer sur chaque clipping plane afin de découper le polygone sujet courant, qui est initialisé en pratique avec le triangle. de découpage du polygone sujet courant par un plan consiste à construire un nouveau polygone (qui sera le prochain polygone courant) appelle newSubject (aussi décrit par une liste de vertex). Pour cela on parcourt les edges du polygone sujet courant  $(v_i, v_{i+1})$  et on détermine la contribution de cet edge au newSubject (i.e. au polygone sujet courant découpé par le plan courant). Pour éviter d'avoir de vertex en double, on conserve que chaque vertex  $(v_i, v_{i+1})$  est responsable de  $v_i$  et de l'éventuelle intersection avec le plan:



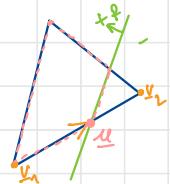
On distingue:



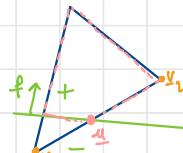
des 2 extrémités de l'edge sont en dehors de la région + du plan, il ne contribue donc pas au nouveau sujet. (Ni  $v_1$ , ni  $v_2$  dans newSubject).



des 2 extrémités sont à l'intérieur de la région + du plan. Ainsi chaque vertex contribue au nouveau sujet. On ajoute  $v_1$  à la liste newSubject car  $v_2$  sous responsabilité de l'edge ( $v_1, v_2$ )



d'edge sont de la région + du plan (i.e.  $v_i$  dedans mais pas  $v_{i+1}$ ). Ainsi  $v_1$  contribue au nouveau sujet et  $v_2$  (l'intersection de l'edge au plan) contribue aussi. Ils sont tous les 2 ajoutés à la liste.



d'edge entre dans la région + du plan (i.e.  $v_i$  en dehors mais  $v_{i+1}$  dedans). Ainsi  $v_1$  contribue au new subject et  $v_2$  aussi. On n'ajoute que  $v_1$  à la liste car  $v_2$  sous responsabilité de l'edge ( $v_2, v_1$ ).

Ainsi on a besoin de déterminer si  $v$  dans région + ou - délimitée par le plan. Il suffit de calculer le signe de  $p \cdot v$ . On a aussi besoin de calculer d'éventuelle intersection i.e. si  $p \cdot v_i < 0$  et  $p \cdot v_{i+1} > 0$  sont opposés (i.e. produit négatif). Alors on doit trouver  $t$  tel point de l'edge  $v_i(t) = v_i + t(v_{i+1} - v_i) \in$  au plan i.e.  $v_i(t) \cdot p = 0$ .

$$\text{D'où } \langle v_i(p) + t \langle v_{i+1} - v_i \rangle p \rangle = 0 \text{ et } t = -\frac{\langle v_i(p) \rangle}{\langle v_{i+1} - v_i \rangle p}.$$

Ainsi  $v_i = v_i - \frac{\langle v_i(p) \rangle}{\langle v_{i+1} - v_i \rangle p} (v_{i+1} - v_i)$  et de là on utilise  $t$  pour interpoler les attributs des 2 vertex pour les définir en  $v$ .

Finalement :

```

subject = [v_0, ..., v_n] # input
clipping Planes = [p_0, ..., p_m] # input
for p in clipping Planes:
    newSubject = []
    for i in [0, n-1]:
        # Cet hne edge
        startV = subject[i]
        endV = subject[(i+1)%n]
        # Telle que vertex in est plus out.
        startIn = <p | startV>
        endIn = <p | endV>
        # Calcul intersect si besoin
        intersectPoint = None

```

```

if startIn < 0:
    t = -<startV | p>
    t /<endV - startV | p>
    intersect = startV + t (endV - startV)
    # Interpoler attribut grâce à t (newSubject.append)

# Ajoute contribut à newSubject
if startIn:
    newSubject.append(startV)
    if not endIn:
        newSubject.append(intersect)
    elif endIn:
        newSubject.append(intersect)
subject = newSubject
return newSubject

```

On dispose alors d'une nouvelle liste de triangle et de vertex en coord clip space, seul triangle faisant face à l'œil et dans le viewing frustum s'y trouve. Formules de nouvelle transform sont appliquées aux vertex.

### \* Du clip space au Normalized Device Coordinates (NDC)

Cette étape correspond à passer les vertex en coord homogènes du clip space à des vertex en coord cartésiennes des NDC. On utilise donc que  $\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$  correspond au vecteur  $\frac{1}{w} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$ . On parle donc de perspective division ou w-division. Plus précisément NDC donnée par :

$$\begin{pmatrix} x_n \\ y_n \\ z_n \\ w_n \end{pmatrix} = \begin{pmatrix} x_c/w_c \\ y_c/w_c \\ z_c/w_c \\ 1/w_c \end{pmatrix} \in [0, 1]^3, \text{ on choisit } w_n = 1/w_c \text{ car besoin de cette info pour réclamer perspective correcte}$$

$= -1/z_c$  (z0) interpolé.

### \* Des NDC au window space :

Cette étape correspond à transformer le NDC en un espace dont les dimensions sont celle du rendu sur écran, on parle de viewport :  $[x, x+w] \times [y, y+h] \times [0, 1]$  (remarque 0, 1 peut être mis à une autre valeur), la 4<sup>e</sup> coord w reste inchangée. Ainsi en prenant la partie entière inférieure de  $x_w$  et  $y_w$ , on obtient les coord des pixels sur lesquels le vertex est projeté.

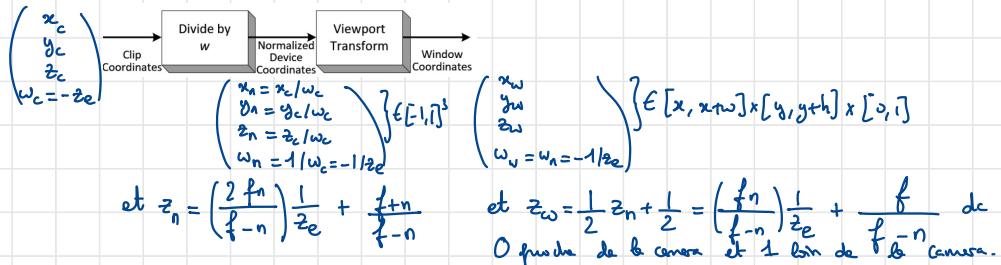
Par exemple pour x la transform est  $x_w = \frac{w}{1+1} (x_n + 1) + x$  car  $-1 \rightarrow x$  et  $1 \rightarrow x + w$ .

$$= \frac{w}{2} x_n + \left( \frac{w}{2} + x \right)$$

Et finalement :

$$\begin{pmatrix} x_w \\ y_w \\ z_w \\ w_w \end{pmatrix} = \begin{pmatrix} \frac{w}{2} x_n + \left( \frac{w}{2} + x \right) \\ \frac{h}{2} y_n + \left( \frac{h}{2} + y \right) \\ \frac{1}{2} z_n + \frac{1}{2} \\ w_n = 1/w_c = 1/z_c \end{pmatrix}$$

\* Recap : primitive assembly (back face culling), puis frustum clipping, puis perspective division, puis viewport transform.



On rappelle qu'en eye space  $z_c$  est négatif, c'est  $-z_c$  qui représente la profondeur à la camera.

## II - Planification

Cette étape consiste à transformer l'ensemble des triangles décrit vectoriellement dans le window space, en une collecte de fragments, un fragment correspondant à un pixel de l'écran qui se trouve à l'intérieur d'un triangle. Il contient comme informations ses coordonnées en window space et des attributs, obtenus par interpolation barycentrique (perspective correcte) des attributs des 3 vertex formant le triangle auquel le fragment appartient.

Par exemple :

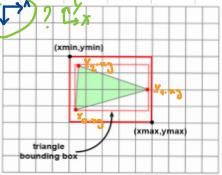


Pour chaque pixel on teste si son centre  $\in$  au triangle. On obtient alors dans cet exemple 11 fragments

### I.1- D'un triangle à ses pixels

Pour chaque triangle, on doit déterminer les pixels qu'il occupe. On itère pour chaque triangle  $(v_0, v_1, v_2)$  avec  $v_i \in \mathbb{R}^3$  en window coordinate.

\* Déterminer la AABBB (2D) du triangle, cela permet de réduire le nb de pixel à tester leur appartenance au triangle :



Une telle BB est décrite par les coins opposés  $v_{\min}$  et  $v_{\max}$ , alors :

$$v_{\max} = (-\infty, -\infty)$$

$$v_{\min} = (+\infty, +\infty)$$

for  $(v \in \in [v_0, v_1, v_2])$ :

$$v_{\min \cdot x} = \min(v_{\min \cdot x}, \text{int}(v \cdot x)) \quad (\text{int en floor})$$

$$v_{\min \cdot y} = \min(v_{\min \cdot y}, \text{int}(v \cdot y))$$

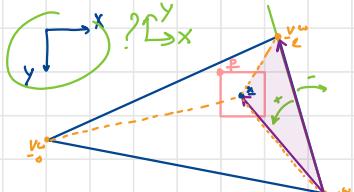
$$v_{\max \cdot x} = \max(v_{\max \cdot x}, \text{int}(v \cdot x))$$

$$v_{\max \cdot y} = \max(v_{\max \cdot y}, \text{int}(v \cdot y))$$

\* Ensuite on itère sur chaque pixel de la AABBB (clip sur l'écran, pour garantir que fragment  $\in$  à l'écran) et on teste leur appartenance au triangle, si il y appartient au triangle on intercale les attributs des vertex et on ajoute ce fragment à notre collecte.



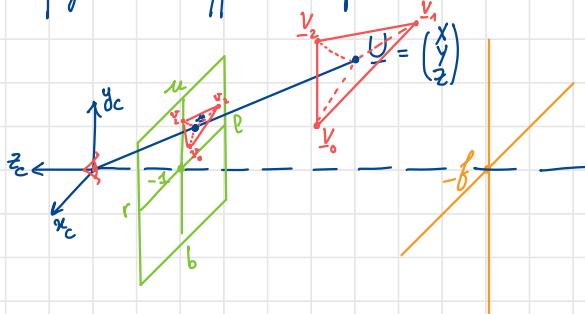
→ Tester l'appartenance d'un pixel au triangle : le pixel est décrit par le point se trouvant en son milieu i.e. de coordonnée 3D  $\underline{z} = \underline{p} + 0,5 \begin{pmatrix} 1 \\ 1 \end{pmatrix}$  avec  $\underline{p}$  le coin inférieur gauche du pixel (min, min).



Pour tester son appartenance, on teste que  $\underline{x}$  se trouve bien dans la région  $\oplus$  délimitée par chaque côté / droite qui forme le triangle. Par exemple, considérons le côté  $(v_1, v_2)$ . Pour savoir si  $\underline{z}$  du côté  $\oplus$ , on peut calculer l'air algébrique du parallélogramme engendré par  $(v_2 - v_0)$  et  $(\underline{z} - v_0)$ . En effet cet air est donné par  $\det \begin{pmatrix} v_2 - v_0 & \underline{z} - v_0 \end{pmatrix}$ ,  $\det \begin{pmatrix} v_2 - v_0 & \underline{z} - v_0 \end{pmatrix} > 0$  si  $\oplus$  est orienté sens direct (det d'une matrice 2x2 : ad - bc)

On remarque aussi que ce déterminant vaut 2 fois l'air du triangle  $(v_0, v_1, v_2)$  i.e. 2 fois le coefficient barycentrique associé à  $v_0$ . Finalement, le calcul de ce déterminant permet de tester si  $\underline{z}$  du bon  $\oplus$  du côté et permet aussi d'obtenir un coefficient barycentrique. Cependant, il reste à corriger ce coefficient car il ne prend pas en compte la perspective, car calculé en 2D...

→ Corriger le coefficient barycentrique : perspective correct interpolation. de test précédent a permis de calculer  $\lambda_0, \lambda_1$  et  $\lambda_2$  les coefficients barycentriques de  $\underline{z}$ . Reste à les corriger pour prendre compte la perspective. Pour cela rentrons à un espace où peut réaliser interpolation barycentrique 3D, comme le eye space, pour simplifier on suppose que plane à 1 :



Alors on a le triangle  $V_0, V_1, V_2 = \begin{pmatrix} x_i \\ z_i \end{pmatrix} \in \mathbb{R}^3$ , qui se projette sur la plan near en  $\underline{v}_i = \frac{1}{z_i} V_i = \begin{pmatrix} x_i \\ 1 \end{pmatrix} \in \mathbb{R}^3$ . Idem  $\underline{U} = \sum \mu_i \underline{V}_i = \begin{pmatrix} y \\ z \end{pmatrix}$  se projette en  $\underline{v} = \frac{1}{z} \underline{U} = \sum \lambda_i \underline{v}_i = \begin{pmatrix} x \\ 1 \end{pmatrix}$ . Ici les inconnues sont  $(\mu_i)$  et  $Z$ , pour le résoudre on va chercher exprimer  $\underline{U}$  comme CL des  $\underline{v}_i$  pour pouvoir identifier les  $\lambda_i$ .

$$\underline{U} = \sum \mu_i \underline{V}_i, \text{ or par relat de projet } \underline{v}_i = Z_i \underline{v}_i \text{ et } \underline{U} = Z \underline{v} \text{ d'où } Z \underline{v} = \sum \mu_i Z_i \underline{v}_i.$$

$$\text{Primi } \underline{v} = \sum \frac{\mu_i Z_i}{Z} \underline{v}_i = \sum \lambda_i \underline{v}_i. \text{ On a bien } \frac{\mu_i Z_i}{Z} \geq 0 \text{ et } \sum \frac{\mu_i Z_i}{Z} = \frac{1}{Z} \sum \mu_i Z_i = 1 \text{ donc } \left( \frac{\mu_i Z_i}{Z} \right) \text{ sont bien des coord barycentriques que l'on peut identifier au (2).}$$

$$\text{D'où } \lambda_i = \frac{\mu_i Z_i}{Z} \text{ et } \mu_i = Z \frac{\lambda_i}{Z_i}. \text{ Reste à trouver } Z, \text{ on sait que } \sum \mu_i = 1 \text{ donc } \sum Z \frac{\lambda_i}{Z_i} = 1 \text{ et } \frac{1}{Z} = \sum \lambda_i \frac{1}{Z_i}.$$

Cependant dans notre cas les " $\lambda_i$ " sont obtenus en window space. On remarque que  $\underline{z} = \sum \lambda_i \underline{v}_i$  donne  $(M_{\underline{z}}) = \sum \lambda_i (M_{\underline{v}_i})$  donc  $(\lambda_i)$  inchargé par transfo linéaire.

$(\lambda_i)$  inchargé pas

$$\underline{x} + \underline{s} = \sum \lambda_i \underline{v}_i + 1 \cdot \underline{s} = \sum \lambda_i \underline{v}_i + \sum \lambda_i \underline{s} = \sum \lambda_i (\underline{v}_i + \underline{s}) \text{ donc } (\lambda_i) \text{ inchargé par translat. } \text{ transfo affine.}$$

D'où  $(\lambda_i)$  inchargé par normalisation entre  $[0, 1]$  du NDC et inchargé par viewport transform.

Pour corriger les  $\lambda_i$  en  $\mu_i$ , donc besoin de connaître les  $\frac{1}{Z_i}$  i.e. inverse de la profondeur à la camera ce que on a stocké dans coord w du window space!

Ainsi pour interpoler correctement on doit:

- Calculer la coord barycentrique en window space  $(\lambda_1, \lambda_2, \lambda_3)$  du fragment  $x$  dans  $(x_1, x_2, x_3)$
- Interpoler l'inverse de la profondeur à la camera donne que inverse profondeur camera du fragment :

$$\frac{1}{z} = \lambda_1 \frac{1}{z_1} + \lambda_2 \frac{1}{z_2} + \lambda_3 \frac{1}{z_3} \text{ i.e. } W = \lambda_1 w_1^W + \lambda_2 w_2^W + \lambda_3 w_3^W \text{ avec } \underline{w}_i = \begin{pmatrix} x_i^W \\ y_i^W \\ z_i^W \end{pmatrix}$$

Idem pour la coord  $z$  en window space, et de la forme  $\frac{a}{z_e} + b$ , ainsi pour  $\underline{w}_i^W$  l'interpoler correctement il faut utiliser  $\lambda_i$  (et non perspective correct interpolation). En effet  $\sum \lambda_i \left( \frac{a}{z_e} + b \right) = a \sum \frac{\lambda_i}{z_e} + b = \frac{a+b}{z_e}$ . Alors les coord du fragment en window space sont données par :

$$\begin{pmatrix} x \in [x, x+W] \\ y \in [y, y+W] \\ z \in [z_e, z_e+H] \\ \sum \lambda_i z_i^W \in [0, 1] \\ \sum \lambda_i w_i^W \in [\frac{1}{f}, \frac{1}{t}] \end{pmatrix} \text{ avec } z_e \text{ la profondeur du pixel, fragment courant (i.e. centre du pixel).}$$

- Interpoler les attributs par :  $A = \sum \mu_i A_i = \sum \lambda_i A_i$  i.e.  $A = \frac{1}{W} (\lambda_1 w_1^W A_1 + \lambda_2 w_2^W A_2 + \lambda_3 w_3^W A_3)$

### III - Fragment Shader

À la suite de la rasterisation on dispose d'une collecte de fragment qui sont décrits par leur point en window space et des attributs des vertex interpolés (e.g. normale, couleur, uv coord). À partir de ces inputs (et d'autres par exemple texture), le rôle du fragment shader est de calculer / output la couleur du fragment.

Ainsi, on obtient une collecte de fragment mis à jour par le calcul de la couleur.

Rq : le fragment shader peut output différentes "couleurs" par fragment, chacune seront écrite dans des buffers différents.

Le fragment shader output aussi la profondeur du fragment. Par défaut cette valeur est initialisée par la coord  $z$  en window space du fragment i.e.  $\frac{f}{f-n} \frac{1}{z_e} + f$  avec  $z_e$  la profondeur (<0) à la camera du pt 3D correspondant au fragment (obtenue par interpolation). Mais l'interpolation peut être modifiée pour éviter Z-fighting lors du depth-test. En effet avec cette valeur de profondeur  $z \approx \frac{1}{z_e}$ , on a que la précision est en  $\frac{dz}{dz_e} \approx \frac{1}{z_e^2}$  ce qui donne une très basse précision proche de la camera et très peu loin de celle-ci (les valeurs sont très proches). Un meilleur comportement serait  $\frac{dz}{dz_e} \approx \frac{1}{z}$  pour avoir haute précision proche camera là où Z-fighting peut être particulièrement visible et garder de la précision même pour  $z$  éloigné. Cela correspond à avoir valeur profondeur  $z \approx \log(z)$ .

### IV - Per Sample Processing

Cette étape consiste à des traitements des fragments pour enfin écrire leur couleur sur un buffer 2D représentant l'écran. Ces traitements, opérations en IL du fragment shader peuvent aussi être réalisés avant le fragment shader. Elles sont :

- Pixel ownership test (automatique).
- Sissor test: On peut définir un rectangle sur l'écran, tel que fragment en dehors ne soit pas affiché. I.e. défini rectangle en XetY du window space et ne rend pas fragment dont coord en X,Y en dehors.

Stencil test: Permet d'appliquer un mask / stencil à l'écran. Stencil est un dimension que la window / buffer composé de valeur 0 et 1 des fragments dont coord X et Y (en window space) qui correspondent à une valeur de 0 du stencil ne sont pas rendus.

Depth test: Permet de cacher les fragments qui se trouve derrière d'autres. lors de l'écriture de la couleur du fragment sur le buffer, on stocke également la "profondeur" du fragment dans le depth buffer (buffer en dimension que la window). Ainsi si un autre fragment à la m<sup>e</sup> coord (x,y) en window space, il override la couleur du pixel (et du depth buffer) que si sa profondeur est inférieure à celle stockée dans depth buffer. Cette stratégie est appelé Z-buffering, car compare la coord Z des fragments.

Réf: On peut changer le critère de comparaison.

On rappelle que la profondeur du fragment est output par fragment shader et est par défaut  $\frac{1}{Z}$ .

Blending: Permet de combiner la couleur du fragment à écrire sur le buffer avec la valeur si trouvront déjà, par exemple provenant d'une première passe de rendu (terrain, puis l'eau). Rend possible la transparence. Par exemple, supposons que le buffer (destinat) est une couleur provenant du terrain, et que fragment (/source) représente l'eau, alors la nouvelle couleur de la destinat est  $d_s C_s + (1-d_s) C_d$  ouuu...  $d_s$ : transparence <sup>source</sup> eau  
-  $C_s$ : couleur <sup>source</sup> eau  
-  $C_d$ : buffer.

Réf: Blending des couleurs et séparé du blending de  $\alpha$ .

Logic Operations

Write mask