Hubert Renaud-Roland
Bailleul Jonathan
Bazzaro Fabrice
Pujos Michaël
Degand Benoît

Vendredi 5 mai 2000

Glos

Manuel utilisateur

Maîtrise Informatique
Université Bordeaux I

Table des matières

Chapitre 1

Présentation

1.1 Fonctionnalités

Voici un court résumé des fonctionnalités de Glos :

- Tracé de points, lignes et triangles spécifiés en 3 dimensions.
- Remplissage de triangles avec couleur uniforme ou couleurs interpolées (gouraud).
- Gestion des transformations usuelles : rotations, translations, mise a l'echelle (scaling).
- Projection perspective avec volume de vue (frustum).
- Gestion du zbuffer pour élimination des faces cachées.
- Jusqu'a 8 sources de lumières directionnelles et gestion des matériaux associés aux faces avec composantes ambiante, diffuse et spéculaire.
- Gestion du Viewport pour visualisation de la scène dans une partie de la fenêtre d'affichage.
- Elimination des faces non visible par la méthode de l'orientation des faces (sens non trigonométrique).

1.2 Concepts généraux

Le fonctionnement de la bibliothèque Glos est celui d'un automate, en effet à tout moment, un environnement est défini. Cette particularité va permettre d'utiliser différentes commandes sans avoir le même résultat suivant le contexte. Les commandes se découpent en deux grands groupes :

- la création de l'environnement

  création des objets et des lumières

- les transformations

  elles permettent de placer les objets dans une scène, et de les déformer.

Pour la création ou définition de l'environnement, il est nécessaire d'utiliser les blocs `Begin`, en effet, c'est la seule commande permettant cette opération. L'utilisation d'un bloc `Begin` peut être directe ou détournée. Les commandes `DrawArray` et `DrawElements` sont par exemple les deux moyens détournés d'utiliser les blocs étant donné que ce sont des macros, le `Begin` est créé mais de manière sous-jacente. Le `Begin` sert en fait à fixer un contexte dans lequel les commandes de définition seront valables. Les commandes essantielles à la définition d'objets sont `Vertex`, `Normal` et `Color`. On peut déjà à partir de ces commandes construire des modèles assez complexes. Pour pouvoir les voir de manière plus réaliste, il faudra néammoins créer des lumières et des matériaux, grâce aux commandes `Light` et `Material`.

Les transformations sont de deux types, comme on peut le voir dans le chapitre qui y est consacré, il y a les transformations appliquées aux objets et celles qui sont appliquées à l'observateur. Dans les deux cas, les commandes sont appelées de la même façon mais dans des environnemts différents, notamment au niveau de celui fixé par `MatrixMode`.

Dans tous ces cas on peut donc voir l'importance de l'environnement dans lequel les commandes sont appelées, il faut donc bien connaître les effets de celles-ci suivant le contexte qui les concerne.

Chapitre 2

Installation

2.1 Prérequis à l'utilisation de Glos

- Une installation de CMU Common Lisp fonctionnelle.
- Le package Defsystem (pour la compilation uniquement).
- Le package CLX installé.

2.2 Décompactage des sources à partir de la distribution Glos

1. Se placer dans le répertoire dans lequel la distribution Glos sera installée (par exemple `/usr/local/src`) :
   `$cd /usr/local/src`
2. décompacter la distribution Glos :
   `$tar zxvf glos-src.tgz`

2.3 Compilation

1. Se placer dans le répertoire des sources :
   `$cd glos-src`
2. lancer la compilation :
   `$make clean ; make`
   Après la compilation, le code binaire compilé de la bibliothèque se trouve dans le fichier `glos-library.x86f`.

2.4 Installation

Copier manuellement le fichier `glos-library.x86f` dans un répertoire où il sera accessible aux applications utilisant la bibliothèque (par exemple dans `/usr/lib/cmucl/glos/`).

2.5 Utilisation

Rajouter les lignes suivantes dans le fichier `~/init.lisp` qui sera chargé automatiquement au démarrage de CMU Common Lisp (ces lignes résolvent des conflits de noms de symboles entre CLX et Common Lisp) :

```
(setq sym (intern "DEFINE-KEYSYM"))
(shadowing-import sym 'xlib)
(setq sym (intern "ANGLE"))
(shadowing-import sym 'xlib)
```

Dans le programme utilisant Glos, il faut charger le code binaire de la bibliothèque avec la commande :
`(load "/usr/lib/cmucl/glos/glos-library.x86f")`

La commande `(use-package :glos)` permet d'utiliser directement les fonctions du package Glos.

Enfin, avant toute utilisation des primitives de Glos, il faut initialiser la bibliothèque avec un appel à la fonction `(init-glos)`.

Un exemple complet de démonstration d'utilisation de Glos se trouve dans `DemoASC.lisp`. Pour le lancer, taper :
`$lisp -load RunDemoASC.lisp`
(il peut être nécessaire au préalable de taper `xhost +` pour que CLX puisse créer la fenêtre).

Chapitre 3

Liste des commandes principales

Toutes les commandes définies par la suite respecterons le modèle suivant :

Nom de la commande

Ligne descriptive de ses fonctionnalités.

Spécification :

Type de commande (Macro/Fonction) prototype de la fonction avec ses paramètres.

Paramètres :

Pour chaque paramètre, une explication est donnée sur le rôle qu'il joue dans la fonction ainsi que les valeurs qu'il peut prendre.

Description :

Toutes les fonctionnalités de la commande sont expliquées suivant le contexte dans lequel elle est appelée.

Notes :

Dans cette partie figure les remarques concernant la commande, notamment les cas particulier d'utilisation.

Erreurs :

Lorsque la commande est susceptible de renvoyer des erreurs, elles sont spécifiées, de plus il est précisé si l'erreur est renvoyée à la compilation ou à l'exécution.

Voir aussi :

Toutes les commandes liées sont citées dans cette partie.

3.1 Création et Définition d'objets

Les commandes suivantes sont utilisées pour la définition d'objets grâce aux blocs `Begin`. En effet ils permettent de donner en séquence une liste de points qui vont suivant l'environnement définir un objet dans un mode particulier (paramètres du `Begin`, par exemple pour tracer en fil de fer). Une partie de l'environnement peut être modifiée à tout moment, car certaines parties vont être stockées pour chaques points et ce grâce aux commandes suivantes :

```
Begin
Color
Material
Normal
Vertex
```

3.1.1 Begin

Délimite la définition de primitives à partir de points.

Spécification :

Macro (Begin mode &rest body)

Paramètres :

- *mode*

  Défini les primitives qui vont être générées à partir de la définition des points effectuées dans le bloc. Les modes acceptés sont les suivants : :POINTS :LINES :LINE-STRIP :LINE-LOOP :TRIANGLES :TRIANGLE-STRIP :TRIANGLE-FAN

Description :

`Begin` délimite les définitions de primitives à partir de celles des points. `Begin` accepte un seul mode qui va déterminer de quelle façon les définitions comprises dans le bloc vont être interprétées. Les résultats suivant les modes sont donnés en prenant en compte un nombre N de points définis dans le bloc `Begin`, avec un index n variant de 1 à N. Les différents modes sont les suivants :

- :POINTS

  Traite chaque point comme une unique point n'étant pas lié à la définition de polygone ou autre.
- :LINES

  Traite chaque paire de points comme un segment. Les points 2n-1 et 2n définissent un segment. N/2 segments seront affichés.
- :LINE-STRIP

  Dessine un ensemble de segments connectés, du premier point jusqu'au dernier. Les points n et n+1 définissent le segment n. N-1 segments seront affichés.
- :LINE-LOOP

  Dessine un ensemble de segments connectés du premier point jusqu'au dernier, puis un autre liant le dernier au premier. N segments seront affichés.
- :TRIANGLES

  Traite chaque triplet de points comme définissant des triangles indépendants. Le triplet 3n-2, 3n-1 et 3n défini le triangle n. N/3 triangles seront affichés.

– :TRIANGLE-STRIP

  Traite une suite de triangles connectés. Un triangle est défini pour chaque points situés après les deux premiers. Pour un index pair, les points n, n+1 et n+2 définissent le triangle n. Pour un index impair, les points n+1, n et n+2 définissent le triangle n. N-2 triangles seront affichés.

– :TRIANGLE-FAN

  Défini un groupe connecté de triangles. Un triangle est défini pour chaque point défini après la définition des deux premier points. Les points 1, n+1 et n+2 définissent le triangle n. N-2 triangles seront affichés.

Seulement un sous-ensemble des commandes peut être utilisé dans un bloc Begin. Les commandes sont les suivantes : Vertex, Color, Normal et ArrayElement. Si une autre commande est présente dans un bloc Begin, une erreur est levée durant la compilation.

Sans tenir compte du mode choisi, il n'y a pas de limite au nombre de points présent dans un bloc Begin. Les lignes, ou triangles qui ne sont pas complètement définis, ne sont pas affichés.

Les minimums de points définis pour chaque primitive sont : 1 pour un point, 2 pour les lignes, 3 pour les triangles. Les modes qui requièrent un certain multiple de points en entrée sont LINES(2) et TRIANGLES(3).

Voir aussi :

`ArrayElement, Color, Normal, Vertex.`

Erreurs :

Une erreur est levée à la compilation si une commande qui ne doit pas figurer dans un bloc `Begin` est présente ou qu'un mode donné en paramètre est incorrect.

3.1.2 Color

Défini la couleur courante.

Spécification :

Fonction (Color r g b)

Paramètres :

- *r, g, b*
  Déterminent les nouveaux niveaux de rouge, vert et bleu présents dans la couleur courante. Les valeurs sont comprises entre 0 et 255.

Description :

`Color` détermine la couleur courante qui va être utilisé pour la définition des points qui suivent. La couleur utilise le mode RGB, avec des valeurs comprises entre 0 et 255. La commande `Color` peut être utilisé dans un bloc `Begin`.

Notes :

La valeur initiale de la couleur courante est le triplet (255, 255, 255).

3.1.3 Material

Spécifie le matériau utilisé pour l'utilisation des lumières.

Spécification :

Macro (Material slot-type &rest slot-values)

Paramètres :

- *slot-type*
  Défini la composante du matériau courant qui est en cours de modification. Les valeurs possibles sont : :AMBIENT, :DIFFUSE, :SPECULAR, :SHININESS et :EMISSION.
- *slot-values*
  Donne les nouvelles valeurs pour la composante du matériau courant.

Description :

`Material` spécifie le matériau utilisé pour l'utilisation des lumières. Les matériaux sont utilisés dans les équations pour les calculs de lumières, optionnellement pour chaque point.

- :AMBIENT
  Permet de définir les paramètres de la réflexion pour la composante ambiante de la lumière (voir `Light`). Pour le définir, il faut donner quatre réels en paramètre, pour le degré de réflexion au niveau du rouge, du vert, du bleu et de l'alpha. La valeur initiale est (0.2, 0.2, 0.2, 1.0).
- :DIFFUSE
  Permet de définir les paramètres de la réflexion pour la composante diffuse de la lumière (voir `Light`). Pour le définir, il faut donner quatre réels en paramètre, pour le degré de réflexion au niveau du rouge, du vert, du bleu et de l'alpha. La valeur initiale est (0.8, 0.8, 0.8, 1.0).
- :SPECULAR
  Permet de définir les paramètres de la réflexion pour la composante speculaire de la lumière (voir `Light`). Pour le définir, il faut donner quatre réels en paramètre, pour le degré de réflexion au niveau du rouge, du vert, du bleu et de l'alpha. La valeur initiale est (0.0, 0.0, 0.0, 1.0).

– :SHININESS

  Permet de définir un second paramètre pour la réflexion de la composante speculaire de la lumière (voir `Light`). Pour le définir, il faut donner un réel qui est l'exposant utilisé. Sa valeur est comprise entre 0 et 128. La valeur par défaut est 0.

– :EMISSION

  Défini les paramètres pour l'émission de lumière émanant de l'objet (voir `Light`). Pour le définir, il faut donner quatre réels en paramètre, pour le degré d'émission au niveau du rouge, du vert, du bleu et de l'alpha. La valeur initiale est (0.0, 0.0, 0.0, 1.0).

Notes :

Les paramètres concernant les matériaux peuvent être modifiés à tout moment. En particulier, la commande `Material` peut être utilisée dans un bloc `Begin`.

Voir aussi :

`Light.`

3.1.4 Normal

Défini le vecteur normal courant.

Spécification :

Fonction (Normal nx ny nz)

Paramètres :

- *nx, ny, nz*
  Spécifie les coordonnées x, y, z de la nouvelle normale courante. La valeur par défaut de la normale courante est (0, 0, 1).

Description :

La normale courante est définie par les coordonnées passée en paramètre. Il n'est pas nécessaire que la normale soit normalisée, il faut alors activer en utilisant Enable le mode :NORMALIZE.

Notes

La normale courante peut être modifiée à tout moment, en particulier dans les blocs Begin.

Voir aussi :

`Begin, Color, Vertex.`

3.1.5 Vertex

Permet de définir un point dans l'espace.

Spécification :

Fonction (Vertex x y z)

Paramètres :

- x, y, z
  Spécifient les coordonnées du point à définir.

Description :

La commande `Vertex` permet de définir un point à l'aide de ses coordonnées. Cette commande est utilisée à l'intérieur des blocs `Begin` pour créer des points, des lignes ou des polygones. La couleur courante, et la normale courante sont associé à un point lors de sa définition par l'appel à `Vertex`.

Notes :

Un appel à `Vertex` à l'extérieur d'un bloc `Begin` est une erreur.

Voir aussi :

`Begin`, `Color`, `Normal`, `VertexPointer`.

3.2 Définition d'objets grâce à des tableaux

Les commandes suivantes permettent de stocker les points dans des tableaux, et de générer les primitives automatiquement suivant des tableaux d'index, ou séquentiellement. De cette façon, on évite de donner tous les points séquentiellement.

```
ArrayElement
DrawArrays
DrawElements
VertexPointer
```

3.2.1 ArrayElement

Effectue le rendu d'un point spécifié à l'intérieur du tableau.

Spécification :

Macro (ArrayElement i)

Paramètres :

- i

  Spécifie l'index utilisable dans le tableau de points.

Description :

`ArrayElement` est une commande utilisée dans un `Begin` afin de définir les attributs pour les données d'un point, une ligne ou un polygone. Utiliser `ArrayElement` pour construire des primitives en indexant les données sur le points plutôt que de donner tous les points du premier au dernier. Les changements effectués sur le tableau dans le bloc `Begin` peuvent affecter les appels à `ArrayElement` effectués dans le même bloc, et ce quelque soit la séquence. Un appel à `ArrayElement` effectué sur un tableau avant une modification peut accéder à la donnée après changement, et vice versa, un appel à `ArrayElement` après une modification peut accéder à la donnée originale.

Voir aussi :

`DrawArrays, VertexPointer`

3.2.2 DrawArrays

Effectue un affichage en utilisant un tableau de données.

Spécification :

Macro (DrawArrays mode first count)

Paramètres :

- *mode*
  Spécifie quel type de primitive va être utilisé pour le rendu. Les valeurs possibles sont : :POINTS, :LINES, :LINE-STRIP, :LINE-LOOP, :TRIANGLES, :TRIANGLE-STRIP et :TRIANGLE-FAN.
- *first*
  Spécifie à partir de quel index on doit partir dans le tableau.
- *count*
  Spécifie le nombre de points qui doivent être utilisés pour l'affichage.

Description :

`DrawArrays` permet de définir des primitives, ce qui évite d'appeler une par une les définition pour chaque point. On peut prédéfinir un tableau pour l'ensemble des points et les utiliser ensuite par un seul appel à `DrawArrays`.

Erreurs :

Une erreur est générée à la compilation si le mode donné en paramètre est inconnu.

Voir aussi :

`ArrayElement, DrawElements, VertexPointer.`

3.2.3 DrawElements

Effectue un affichage à partir de tableaux de données.

Spécification :

Macro (DrawElements mode count indices)

Paramètres :

- *mode*

  Spécifie quel type de primitive va être utilisé pour le rendu. Les valeurs possibles sont : :POINTS, :LINES, :LINE-STRIP, :LINE-LOOP, :TRIANGLES, :TRIANGLE-STRIP et :TRIANGLE-FAN.
- *count*

  Spécifie le nombre de points qui vont êtres utilisés pour l'affichage.
- *indices*

  C'est un tableau contenant les indices des points qui vont être traités.

Description :

`DrawElements` permet de définir des primitives, ce qui évite d'appeler une par une les définitions pour chaque points. L'avantage de cette commande par rapport à `DrawArrays`, est que l'on peut traités les points dans l'ordre que l'on veut (suivant le tableau *indices*) sans avoir modifier le tableau original.

Erreurs :

Une erreur est générée à la compilation si le mode donné en paramètre est inconnu.

Voir aussi :

`ArrayElement`, `DrawArrays`, `VertexPointer`.

3.2.4 VertexPointer

Défini un tableau de points.

Spécification :

Macro (VertexPointer array)

Paramètres :

- *array*

  Permet de spécifier le tableau de points.

Description :

Spécifie le tableau de points qui va être utilisé par la suite à l'aide des commandes DrawArrays, DrawElements et ArrayElement. Il permet de prendre lors de ces opérations les points stockés à l'intérieur du tableau array.

Notes :

Cette commande permettant de définir des points ne peut pas être appelée dans un bloc `Begin`.

Voir aussi :

`ArrayElement, DrawArrays, DrawElements.`

3.3 Transformations

Toutes les commandes suivantes permettent de gérer et créer des transformations sur des objets. Les transformations sont basées sur des rotations, des translations ou des changements d'échelle. Toutes les matrices créées à partir de ces transformations sont ensuites utilisées pour le rendu. il y a donc deux catégories de matrices, les matrices de transformation proprement dites et les matrices de projection, créées en fonction d'un champs de vision, de la position de l'observateur, son point de visée et la direction du haut pour lui. Il y a donc des opérations qui consistent seulement à changer de mode pour définir l'observateur ou les transformations sur les objets.

`Frustum` `LoadIdentity`
`LoadMatrix`
`MatrixMode`
`MultMatrix`
`PushMatrix`
`Rotate`
`Scale`
`Translate`

3.3.1 Frustum

Multiplie la matrice courante par une matrice de perspective.

Spécification :

Fonction (Frustum left right bottom top near far)

Paramètres :

- *left, right*
  Spécifient les coordonnées des plans de coupe verticaux de droite et de gauche.
- *bottom, top*
  Spécifient les coordonnées des plans de coupe horizontaux du bas et du haut.
- *near, far*
  Spécifient les distances pour les plans de coupe de profondeur, le plan proche et le plan éloigné. Les deux valeurs doivent être positives.

Description :

`Frustum` défini une matrice qui produit une projection perspective. La matrice courante (voir `MatrixMode`) est multipliée par cette matrice, et le résultat remplace la matrice courante, de la même façon que si on avait appelé `MultMatrix` avec la matrice suivante :

$$M = \begin{bmatrix} A & 0 & B & 0 \\ 0 & C & D & 0 \\ 0 & 0 & E & F \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

$$A = \frac{2 \times near}{right - left}$$

$$B = \frac{right + left}{right - left}$$

$$C = \frac{2 \times near}{top - bottom}$$

$$D = \frac{top + bottom}{top - bottom}$$

$$E = \frac{far + near}{far - near}$$

$$F = \frac{2 \times far \times near}{far - near}$$

Généralement, cette fonction est appelée lorsque le mode courant est "projection". Les points de coordonnées (*left*, *bottom*, -*near*) et (*right*, *top*, -*near*) définissent les points sur le plan de coupe proche qui vont être projetés dans le coin en bas à gauche et le coin en haut à droite d'une fenêtre, en tenant compte du fait que l'on considèrent que l'observateur est placé en (0, 0, 0).

Voir aussi :

`Ortho, MatrixMode, MultMatrix, PushMatrix, Viewport.`

3.3.2 LoadIdentity

Remplace la matrice courante par la matrice identité.

Spécification :

(LoadIdentity)

Description :

`LoadIdentity` remplace la matrice courante par la matrice identité. Sémantiquement, cette opération est la même que celle qui consiterait en un appel à `LoadMatrix` avec la matrice identité

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

mais dans la plupart des cas cela serait moins performant.

Voir aussi :

`LoadMatrix`, `MatrixMode`, `MultMatrix`, `PushMatrix`.

3.3.3 LoadMatrix

Remplace la matrice courante par la matrice passée en paramètre.

Spécification :

(LoadMatrix m)

Paramètres :

- *m*

  matrice de 4x4 contenant des valeurs réelles.

Description :

`LoadMatrix` remplace la matrice courante par la nouvelle matrice m. La matrice courante peut être la matrice "modelview" ou "projection" dépendant du mode courant (voir `MatrixMode`). La matrice courante spécifie une transformation de coordonnées. Par exemple, on suppose que m est la matrice "modelview". Si V=(x, y, z, w) représente la valeur des coordonnées d'un vecteur, la transformation M(V) va donc produire :

$$M(V) = \begin{bmatrix} m_{0,0} & m_{0,1} & m_{0,2} & m_{0,3} \\ m_{1,0} & m_{1,1} & m_{1,2} & m_{1,3} \\ m_{2,0} & m_{2,1} & m_{2,2} & m_{2,3} \\ m_{3,0} & m_{3,1} & m_{3,2} & m_{3,3} \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

La multiplication $\times$ dans ce cas présent est la multiplication de matrices. Pour les transformations de projection, les opérations sont similaires.

Voir aussi :

`LoadIdentity, MatrixMode, MultMatrix, PushMatrix.`

3.3.4 MatrixMode

Spécifie quelle matrice est la matrice courante.

Spécification :

Macro (MatrixMode mode)

Paramètres :

- *mode*

  Spécifie quelle pile de matrice sera la cible des prochaines opérations matricielles. Deux valeurs sont acceptées : :MODELVIEW et :PROJECTION.

Description :

`MatrixMode` fixe le mode de la matrice courante. mode peut avoir les deux valeurs suivantes : :MODELVIEW Les prochaines opérations sur les matrices seront appliquées à la pile de matrices "modelview". :PROJECTION Les prochaines opérations sur les matrices seront appliquées à la pile de matrices "projection".

Voir aussi :

`LoadMatrix, PushMatrix.`

3.3.5 MultMatrix

Multiplie la matrice courante par la matrice spécifiée.

Spécification :

Macro (MultMatrix m)

Paramètres :

- m

  Matrice de 4x4 contenant des valeurs réelles.

Description :

`MultMatrix` multiplie la matrice courante par celle passée en argument m, et remplace la matrice courante par le produit. La matrice courante est déterminée par le mode courant qui est soit "modelview" soit "projection".

Exemple :

Si la matrice courante est C, les coordonnées à transformer sont, $v = (v_0, v_1, v_2, v_3)$. Alors la transformation serait C $\times$ v, soit :

$$\begin{bmatrix} c_{0,0} & c_{0,1} & c_{0,2} & c_{0,3} \\ c_{1,0} & c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,0} & c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,0} & c_{3,1} & c_{3,2} & c_{3,3} \end{bmatrix} \times \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

Un appel à `MultMatrix` avec comme argument la matrice M, remplace la transformation par $(C \times M) \times v$ soit :

$$\begin{bmatrix} c_{0,0} & c_{0,1} & c_{0,2} & c_{0,3} \\ c_{1,0} & c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,0} & c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,0} & c_{3,1} & c_{3,2} & c_{3,3} \end{bmatrix} \times \begin{bmatrix} m_{0,0} & m_{0,1} & m_{0,2} & m_{0,3} \\ m_{1,0} & m_{1,1} & m_{1,2} & m_{1,3} \\ m_{2,0} & m_{2,1} & m_{2,2} & m_{2,3} \\ m_{3,0} & m_{3,1} & m_{3,2} & m_{3,3} \end{bmatrix} \times \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

Notes :

L'ordre des multiplications est important, si la transformation courante est une rotation, et que `MultMatrix` est appelée avec une matrice de translation en paramètre, la translation sera effectuée sur les coordonnées des points alors que la rotation sera effectuée sur les points après translation.

Voir aussi :

`LoadIdentity`, `LoadMatrix`, `MatrixMode`, `PushMatrix`.

3.3.6 PushMatrix

`PushMatrix/PopMatrix` empile et dépile les matrices de la pile courante.

Spécification :

Fonction (PushMatrix)
Fonction (PopMatrix)

Description :

Il y a une pile de matrices pour chacun des modes. Dans le mode "modelview", la profondeur maximale de la pile est de 32. Dans le mode "projection", la profondeur maximale de la pile est de 32.

`PushMatrix` empile une copie de la matrice courante dans la pile du mode courant. Donc de suite après un appel à la Fonction `PushMatrix`, la matrice en tête de pile est la même que la matrice courante.

`PopMatrix` dépile la matrice de la pile courante et remplace la matrice courante par celle qui est dépilée.

Initiallement, les deux piles de matrices sont vides. Des ereurs peuvent être dues au fait d'empiler dans une pile pleine ou de dépiler une pile vide.

Voir aussi :

`Frustum, LoadIdentity, LoadMatrix, MatrixMode, MultMatrix, Ortho, Rotate, Scale, Translate, Viewport.`

3.3.7 Rotate

Multiplie la matrice courante par la une matrice de rotation.

Spécification :

Fonction (Rotate angle x y z)

Paramètres :

- *angle*

  Défini l'angle de la rotation.
- *x, y, z*

  Définissent les coordonnées d'un vecteur.

Description :

`Rotate` produit une rotation de angle degrés autour du vecteur (x, y, z). La matrice courante (voir `MatrixMode`) est multipliée par la matrice de rotation et est ensuite remplacée par le résultat. Le même résultat peut être obtenu en appelant `MultMatrix` avec la matrice suivante :

$$\begin{bmatrix} xx(1-c)+c & xy(1-c)-zs & xz(1-c)+ys & 0 \\ yx(1-c)+zs & yy(1-c)+c & yz(1-c)-xs & 0 \\ xz(1-c)-ys & yz(1-c)-xs & zz(1-c)+c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Où $c = \cos angle$, $s = \sin angle$ et $||(x, y, z)|| = 1$. Si c n'est pas le cas la fonction normalise le vecteur.

Voir aussi :

`MatrixMode, MultMatrix, PushMatrix, Scale, Translate.`

3.3.8 Scale

Multiplie la matrice courante par la matrice de facteurs.

Spécification :

Fonction (Scale x y z)

Paramètres :

- x, y, z

  Définissent les facteurs de déformation suivant les trois axes.

Description :

`Scale` produit une déformation non uniforme suivant les axes des x, des y, et des z. Les trois paramètres donnent les facteurs appliqués sur chacun des axes. La matrice courante (voir `MatrixMode`) est multipliée par la matrice de facteur, et le produit la remplace comme si `MultMatrix` avait été appelé avec la matrice suivante en paramètre :

$$M = \begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Voir aussi :

`MatrixMode`, `MultMatrix`, `PushMatrix`, `Rotate`, `Translate`.

3.3.9 Translate

Multiplie la matrice courante par la matrice de translation.

Spécification :

Fonction (Translate x y z)

Paramètres :

- *x, y, z*
  Spécifient le vecteur de translation.

Description :

`Translate` permet une translation par le vecteur (x, y, z). La matrice courante (voir `MatrixMode`) est alors multipliée par la matrice de translation générée. Il s'agit de la même opération qu'un appel à `MultMatrix` avec la matrice suivante en paramètre :

$$M = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Voir aussi :

`MatrixMode`, `MultMatrix`, `PushMatrix`, `Rotate`, `Scale`.

3.4 Modification des buffers et de l'affichage

Des commandes permettent de modifier directement les buffers : le buffer d'affichage (un double buffer est utilisé), et le Z-buffer. Ces deux buffers peuvent être réinitialisé à l'aide de la commande `Clear`. Pour le buffer d'affichage, la couleur servant pour l'initialisation est fixée par la commande `ClearColor`. Au niveau de l'affichage, on peut le faire seulement dans une partie de la fenêtre courante. De cette façon, on va pouvoir avoir plusieurs vues d'un même objets dans la même fenêtre d'affichage grâce à la commande `Viewport`.

```
Clear
ClearColor
Viewport
```

3.4.1 Clear

Efface une zone mémoire suivant des valeurs prédéfinies.

Spécification :

Macro (Clear &rest buffers)

Paramètres :

– *buffers*

Indique quels tampons mémoires doivent être réinitialisés. Ce paramètre peut prendre pour valeur :COLOR-BUFFER ou DEPTH-BUFFER ou même les deux.

Description :

`Clear` permet de réinitialiser le ou les tampons mémoires spécifiés en paramètre suivant les valeurs prédéfinies par la commande `ClearColor` pour le tampon mémoire concernant l'affichage. Le paramètre :COLOR-BUFFER permet de réinitialiser le tampon d'affichage, et le paramètre DEPTH-BUFFER, permet de réinitialiser le Z-BUFFER.

Voir aussi :

`ClearColor`.

3.4.2 ClearColor

Spécifie avec quelle couleur va être réinitialiser le tampon mémoire d'affichage.

Spécification :

Fonction (ClearColor r g b)

Paramètres :

- *r, g, b*
  Ces trois paramètres définissent respectivement le niveau de rouge, de vert et de bleu pour la couleur servant à la réinitialisation. Les valeurs sont comprises entre 0 et 255.

Description :

`ClearColor` spécifie les degré de rouge, vert et bleu qui vont être utilisé lors de l'appel de `Clear` pour effacer le tampon de mémoire de l'affichage.

Voir aussi :

`Clear`.

3.4.3 Viewport

Défini une fenêtre d'affichage.

Spécification :

Macro (Viewport x y w h)

Paramètres :

- x, y

  Définissent le coin inférieur gauche de la fenêtre en pixels. La valeur initiale est (0, 0).
- w, h

  Définissent la largeur et la hauteur de la fenêtre. Quand une fenêtre est créée, largeur et hauteur sont fixés à la taille de celle-ci.

Description :

`Viewport` permet de spécifier les transformations qui vont permettre le calcul des coordonnées d'un pixel dans une partie de la fenêtre d'affichage.

Notes :

3.5 Gestion des lumières

Pour avoir un affichage, il est nécessaire d'avoir au moins une source de lumière. La commande `Light` permet de placer jusqu'à huit lumières dans le monde et de régler tous les paramètres nécessaires au niveau de la couleur, et surtout du type de lumière. La commande `ShadeModel` permet elle de fixer pour un objet comment la lumière doit être traitée, soit pour tous les points des faces, on calcule la couleur soit, on calcule une couleur pour toute la face.

```
Light
ShadeModel
```

3.5.1 Light

Défini une source de lumière.

Spécification :

Fonction (Light lightn pname param)

Paramètres :

- *lightn*

  Spécifie une lumière. Il y a 8 lumières disponibles, elles sont identifiées par un nom LIGHTi nombre i, $0 \leq i < 8$.
- *pname*

  Spécifie le type de lumière, ou la position, les valeurs suivantes sont acceptées : :AMBIENT, :DIFFUSE, :SPECULAR et :POSITION.
- *param*

  Ce paramètre dépend du paramètre *pname*, suivant sa valeur, on va devoir spécifier les composantes RGB, ou la position.
  - :AMBIENT

    param contient 4 valeurs réelles qui définissent les intensités de rouge, de vert, de bleu et d'alpha contenues dans la source de lumière. Ces quatre valeurs sont comprises entre 0 et 1.
  - :DIFFUSE

    param contient 4 valeurs réelles qui définissent les intensités de rouge, de vert, de bleu et d'alpha contenues dans la source de lumière. Ces quatre valeurs sont comprises entre 0 et 1.
  - :SPECULAR

    param contient 4 valeurs réelles qui définissent les intensités de rouge, de vert, de bleu et d'alpha contenues dans la source de lumière. Ces quatre valeurs sont comprises entre 0 et 1.
  - :POSITION

    param contient 4 valeurs réelles qui sont des coordonnées homogènes correspondant à la position de la source de lumière dans l'espace.

Description :

`Light` permet de définir pour chaque source de lumière sont type, sa position et ses caractéristiques aux niveau de la couleur. Pour activer ou désactiver une lumière, il faut utiliser `Enable/Disable` avec pour paramètre LIGHTi.

3.5.2 ShadeModel

Selectionne un modèle lissé ou pas.

Spécification :

Macro (ShadeModel model)

Paramètres :

- *model*

  Permet de spécifier le modèle à appliquer, les valeurs possibles sont : :FLAT ou :SMOOTH.

Description :

Le calcul des nuances par défaut est en mode :SMOOTH, dans ce cas, les couleurs de chaques pixels sont calculées par interpolation à partir des points définis. Le mode :FLAT lui calcule les couleurs pour les points définis, les traite et applique le résultat à la face. Dans tous les cas la couleur d'un point est calculée en fonction de la lumière si elle est activée et par la couleur courante dans l'autre cas.

Erreurs :

Une erreur est générée lors de la compilation si le paramètre passé comme model n'existe pas.

Voir aussi :

`Begin, Color, Light.`

3.6 Gestion des options

Une grande partie des commandes provoque des définition qui ne seront pas nécessairement utilisées pour le rendu. Pour activer ou désactiver des options, on peut utiliser les commandes `Enable` et `Disable`. Ces options permettent per exemple d'activer ou pas les lumières ou la normalisation des normales avant tout rendu.

`Enable`

3.6.1 Enable

`Enable/Disable` Permet d'activer ou de désactiver une option lors du rendu.

Spécification :

Macro (Enable mode)
Macro (Disable mode)

Paramètres :

- *mode*

  Permet de spécifier quel est le paramètre qui va être activé ou désactivé. Les valeurs de mode peuvent être : :LIGHTING, :DEPTH-BUFFER ou :NORMALIZE.

Description :

`Enable` active le mode passé en paramètre. `Disable` désactive le mode passé en paramètre.

- :LIGHTING

  Permet d'activer ou pas les lumières, elles seront alors ou pas utilisée pour les calculs de couleurs des points.
- :DEPTH-BUFFER

  Cette option est liée au Z-BUFFER qui peut être activé ou désactivé.
- :NORMALIZE

  Cette option permet de normaliser les normales définies dans les blocs Begin après les transformations. En effet, certaines transformations modifient la norme des vecteurs.

Voir aussi :

`Light, Normal.`

Chapitre 4

Commandes supplémentaires

Les commandes suivantes sont dans un chapitre séparé car elles utilisent les primitives précédentes. Elles ne sont donc pas essentielles mais permettent certaines facilités d'utilisation.

4.1 LookAt

Défini les transformation de vue.

Spécification :

Fonction (LookAt eyeX eyeY eyeZ centerX centerY centerZ upX upY upZ)

Paramètres :

- *eyeX, eyeY, eyeZ*
  Définissent la position du point de vue (position de l'observateur).
- *centerX, centerY, centerZ*
  Définissent la position du point de référence (le point cible de l'observateur).
- *upX, upY, upZ*
  Définissent le vecteur montrant la direction "haut" de la vue.

Description :

`LookAt` crée une matrice de transformation en utilisant un point de vue, un point de référence et un vecteur donnant la direction du haut pour l'observateur. On pourrait éviter d'utiliser `LookAt` en faisant les calculs suivants :

$$F = \begin{bmatrix} centerX - eyeX \\ centerY - eyeY \\ centerZ - eyeZ \end{bmatrix}$$

On normalise alors le vecteur : $f = \frac{F}{||F||}$
de même pour le vecteur UP : $UP' = \frac{UP}{||UP||}$
On pose ensuite $s = f \times UP'$, et $u = s \times f$

On construit alors la matrice M :

$$M = \begin{bmatrix} s_0 & s_1 & s_2 & 0 \\ u_0 & u_1 & u_2 & 0 \\ -f_0 & -f_1 & -f_2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

et LookAt est équivalent à :

(`MultMatrix` M)
(`Translate` -eyeX -eyeY -eyeZ)

Voir aussi :

`Frustum`, `Perspective`.

4.2 Perspective

Défini une matrice de projection.

Spécification :

Fonction (Perspective fovy aspect znear zfar)

Paramètres :

- *fovy*

  Définit le champs de vue en degrés dans la direction y.
- *aspect*

  Définit le ratio qui va permettre de donner le champs de vue en x. Ce ratio est celui entre les x (largeur) et les y (hauteur).
- *znear*

  Définit la distance entre l'observateur et le plan de coupe proche.
- *zfar*

  Définit la distance entre l'observateur et le plan de coupe éloigné.

Description :

`Perspective` définit un volume de vue dans les coordonnées du monde. En général le ratio déterminant l'aspect, est équivalent à celui qu'il y a entre la largeur et la hauteur de la fenêtre d'affichage associée. La matrice générée par la fonction `Perspective` est multipliée par la matrice courante, comme si la commande `MultMatrix` avait été appelée avec pour paramètre la matrice générée. Pour charger seulement la matrice de perspective à la place de la matrice courante, il suffit de précéder l'appel à `Perspective` par `LoadIdentity`.
Soit f défini comme suit :

$$f = cotangent(\frac{fovy}{2})$$

La matrice M générée est donc :

$$M = \begin{bmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{zfar+znear}{znear-zfar} & \frac{2\times zfar\times znear}{znear-zfar} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Voir aussi :

`Frustum`, `LoadIdentity`, `MultMatrix`.