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

Vendredi 5 mai 2000

Glos

Manuel de maintenance

Maîtrise Informatique
Université Bordeaux I

Table des matières

Ce manuel est destiné à toute personne voulant modifier ou augmenter les fonctionnalités de la bibliothèque Glos. On suppose que le lecteur a une bonne connaissance du langage Common Lisp et qu'il possède une bonne compréhension des concepts d'infographie liés à la 3D. Une description détaillée des primitives de Glos se trouve dans le *manuel de l'utilisateur de Glos*.

Ce manuel présente l'architecture interne de la bibliothèque et un exemple concret d'ajout de fonctionnalités.

1 Compilation et Installation

Suivre les instructions du *manuel de l'utilisateur de Glos*.

2 Structure générale du code

Le code est découpé en plusieurs fichiers sources. Chaque fichier regroupe les fonctions, macros et variables spécifiques à un aspect de la bibliothèque.

2.1 Description des fichiers

Le code comprend :

- la bibliothèque graphique Glos.
- un module de lecture et stockage d'objets 3D au format ASC.
- un programme d'exemple et de test utilisant le module ASC et la bibliothèque Glos pour le rendu et l'animation d'objets ASC.

Les fichiers suivants constituent la bibliothèque Glos :

- **Glos.lisp** : variables globales partagées entres tous les fichiers, variables d'état, fonctions de manipulation de l'état.
- **Math3D.lisp** : définition et opérations mathématiques sur les vecteurs à 3 et 4 éléments et les matrices 4x4.
- **Transforms.lisp** : primitives de manipulation des matrices de transformations de type MODELVIEW et PROJECTION.
- **Render.lisp** : implémentation des algorithmes de tracé (rendu) des primitives graphiques.
- **VertexBuffer.lisp** : définition du bloc `Begin` et pipeline de transformation des vertices. Le vertex buffer est le buffer contenant les vertices à transformer.
- **Tools.lisp** : divers utilitaires utilisés par les autres modules y compris toutes les readers macro et les fonctions (ou macros) utilisées par d'autres macros pour la macroexpansion de la macro (car elles doivent être compilées *"avant"* la macro qui les utilise).
- **Lighting.lisp** : manipulation des attributs des lumières et matériaux.
- **glos.system** : fichier defsystem pour la bibliothèque Glos.

Les fichiers suivants constituent le module ASC :

- **ASCParser.lisp** : lecture et stockage du fichier ASC

- **Object.lisp** : structures de données pour le stockage d'objets ASC et calcul des normales pour chaque vertex de l'objet.

Le fichier suivant contient le programme de test :

- **DemoASC.lisp** : Affichage avec rotation d'un objet ASC

2.2 Conventions de nommage des fonctions et variables

- Les variables globales sont entourées du symbole `*`
  ex : `(defparameter *screen-width* 320)`
- Les constantes sont entourées du symbole `+`

  ex :
  ```
  (defconstant +state-keys+ '(   :LIGHTING :DEPTH-BUFFER
                                 :NORMALIZE))
  ```
- Le nom des fonctions de l'API ont le même nom que leur équivalent OpenGL dont elle sont inspirées avec le préfixe `gl` en moins (ex : `glEnable` devient `Enable`).
- Le nom des fonctions internes ne faisant pas partie de l'API respectent l'idiome du langage Lisp (en minuscule, avec des tirets pour séparer les mots).
- Si *type* est un type de données, ses primitives commencent par "*type-*" (ex : `vec4-make` pour le type `vec4`).
- Un ensemble de variables globales dont l'objet est relaté doivent commencer par le même préfixe (exemple les variables liées aux vertex buffer sont préfixées par `vb-`).
- Le type `single-float` est utilisé pour toute fonction manipulant des flottants.

2.3 Description du pipeline de transformation

Le corps du bloc `Begin` contient la définition (coordonnées, normales, couleurs) des vertices spécifiant les primitives graphiques à tracer. Les vertices sont stockés dans le tableau `*vb*` et sont prêts à subir les transformations qui vont les conduire de l'état de points en 3 dimensions à celui de points en 2 dimensions définissant un pixel. Ces transformations ont lieu dans la fonction `do-transforms` La série de transformation appliquée à chaque vertex est directement inspirée de la section *"Coordinate transformation"* pp 28-29 de la spécification d'OpenGL 1.2.1. Le schéma fig. 1 synthétise la succession d'opérations pour chaque vertex :

Tout d'abord chaque vertex v_0 en coordonnées de l'objet (object coordinates), c'est à dire par rapport au repère local attaché à l'objet, est transformé par la matrice MODELVIEW M (stockée dans la variable `*modelview-matrix*`) en un vertex v_e pour donner les coordonnées de l'oeil (eye coordinates) :

$$v_e = M \cdot v_0$$

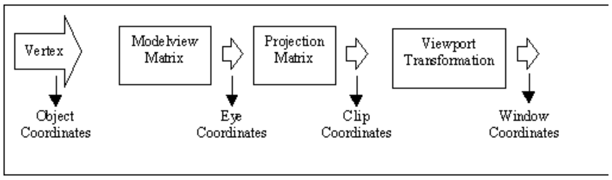

FIG. 1 – Pipeline de transformation

Pour pouvoir effectuer les calculs relatifs à l'éclairage (décrits dans la section suivante), le vecteur normal n_0 associé au vertex v_0 est transformé par la matrice 3x3 supérieure de M appelée M_{33}, pour donner le vecteur normal n_e :

$$n_e = M_{33} \cdot n_0$$

Ensuite, le vertex v_e est transformé en coordonnée de clip (clip coordinates) par la matrice de projection P (`*projection-matrix*`) pour donner le vertex v_c :

$$v_c = P \cdot v_e$$

La division perspective est appliquée à v_c (division par la composante w_c du vecteur w_c) pour donner les coordonnées normalisées d'écran (normalized device coordinates) v_d :

$$v_d = \begin{bmatrix} x_d \\ y_d \\ z_d \end{bmatrix} = \begin{bmatrix} x_c/w_c \\ y_c/w_c \\ z_c/w_c \end{bmatrix}$$

Enfin le vertex v_d est transformé en coordonnées de fenêtrage v_w) (window coordinates) correspondant au coordonnées finale 2D du pixel qui doit être affiché dans la fenêtre. p_x) et p_y) sont respectivement la longueur et la hauteur de la fenêtre (viewport) en pixels et (o_x, o_y) sont centre :

$$v_w = \begin{bmatrix} x_w \\ y_w \end{bmatrix} = \begin{bmatrix} (p_x/2)x_d + o_x \\ (p_y/2)y_d + o_y \end{bmatrix}$$

2.4 Description de l'éclairage

Quand une image en trois dimensions est calculée, le modèle doit paraître correct au niveau de la géométrie, mais il doit également avoir une apparence réaliste. C'est généralement réalisé en utilisant la combinaison des matériaux et des lumières. Ici, seul le cas des lumières directionnelles sera développé, mais les paramètres des structures `Material` et `Light` permettent beaucoup d'autres options. La fonction `do-lighting` est responsable du calcul de l'éclairage pour chaque vertex.

2.4.1 Les composantes de la lumière et des matériaux.

Les composantes définissant la couleur des composantes `ambient`, `diffuse` et `specular` de la lumière et des matériaux se composent de quatre éléments : rouge, vert, bleu et transparence. Chaque élément est une valeur comprise entre 0 et 1, ce qui facilitera les calculs par la suite.

- La composante `emission` des matériaux.
  Cette composante permet de faire de objets auto-éclairant, elle ne dépend donc d'aucune lumière et correspond à une quantité de lumière émise par l'objet lui même.

2.4.2 Lumière directionnelle.

Une lumière directionnelle est considérée comme une lumière positionnée à une distance infinie de l'objet éclairé et qui ne subit aucune atténuation en fonction de la distance. On peut par exemple comparer une lumière directionnelle avec le soleil.

- La composante `ambient`.
  Elle détermine la quantité de lumière ambiante réfléchie par l'objet, quelle que soit la position de l'objet par rapport à la lumière.

$$ambientlight \times ambientmaterial$$

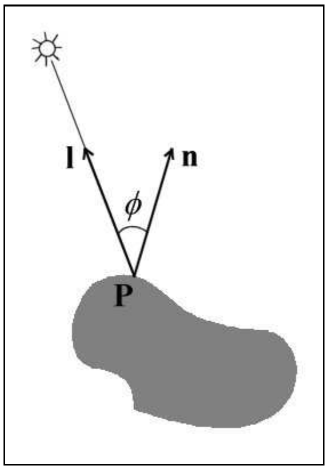

FIG. 2 – Eclairage diffuse

$$max((n \cdot l), 0) \times diffuselight \times diffusematerial$$

- La composante `diffuse`.
  Cette partie du modèle d'éclairage est celle qui se rapporte le plus à la réalité physique, avec l'interaction entre les photons et la surface. En effet, elle se base sur la loi de *Lambert*, qui s'applique aux surfaces mates. La quantité de lumière réfléchie est déterminée par le cosinus entre le vecteur normal **n** au point calculé et le vecteur directeur de la lumière **l**.

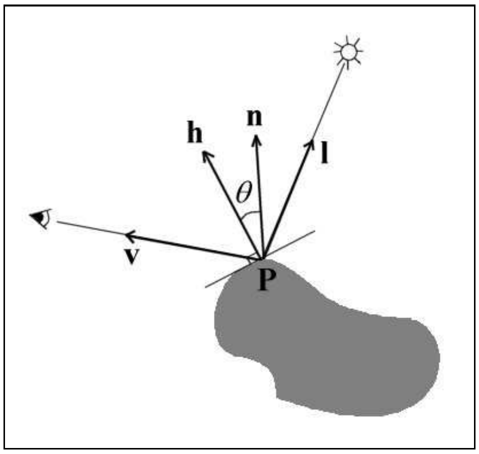

FIG. 3 – Eclairage spéculaire

$$max((n \cdot h), 0)^{shininessmaterial} \times specularlight \times specularmaterial$$

avec :

$$h = \frac{l + v}{||l + v||}$$

- La composante `specular` et `shininess`.
  Cette dernière composante permet de donner un aspect plus ou moins brillant à l'objet. Cette composante permet de déterminer visuellement la position de la lumière par rapport à l'objet. Cet effet est obtenu en utilisant l'équation d'illumination de *Phong* qui utilise deux vecteurs, **v** qui détermine la direction du la caméra vers le point calculé et **r** la direction de la réflection de la lumière. A ne pas confondre avec l'algorithme d'ombrage de *Phong* qui consiste à interpoler les normales et la lumière pour chaque pixel. Ici l'équation est légèrement modifiée mais aboutit à un résultat semblable. La quantité de lumière réfléchie est déterminée par le cosinus entre le vecteur normal **n** au point calculé et le vecteur **h** qui n'est que la somme normalisée du vecteur lumière **l** et celui de la direction de la caméra **v**. Ce cosinus est ensuite modifié par un coefficient `shininess`. Plus le coefficient `shininess` est élevé plus l'objet apparaît brillant. Ici comme le repère est ramené a celui de la caméra **v** est toujours égal a (0.0, 0.0, 1.0).

3 Extension de la bibliothèque

3.1 Recommendations

Une contrainte importante de Glos est de pouvoir animer des objets 3D en temps réel avec des performances acceptables. En particulier on désire obtenir une animation le plus rapide et fluide possible. Lors du rajout de fonctionnalités il faut :

- Eviter à tout prix les allocations de mémoire (appelé consing dans la terminologie Lisp) autant que possible pour les fonctions de l'API Glos et les fonctions internes auxquelles elles font appel. Le ramasse-miettes de CMU Common Lisp génére une pause sensible lors de la récupération de la mémoire allouée. Lors de l'animation d'objets 3D cela se traduit par une pause à intervalle régulier.
  Certaines fonctions de l'API sont souvent appelées dans des boucles par l'utilisateur final (ex : `Rotate`, `Begin`, `Vertex`,...). Si l'on prends l'exemple d'une fonction qui alloue 1Ko à chaque appel¸ après 1000 appels on se retrouve avec près d'1Mo de mémoire allouée. La suppression totale de consing est moins importante dans les fonctions appelées peu souvent comme par exemple `Perspective` typiquement invoquée lors de la création ou redimensionnement de la fenêtre d'affichage. Dans le code des macros participant à la macroexpansion il n'est pas utile de s'occuper du consing, puisque ce consing est généré à la compilation. Il faut veiller toutefois à ce que le code macroexpansé ne génère pas de consing !
- Toujours déclarer le type des variables globales et locales ainsi que des arguments des fonctions¸pour que le compilateur puisse optimiser le code.
- Si la dernière instruction d'une fonction renvoie un flottant et que ce n'est pas désiré, renvoyer `t` à la fin de la fonction pour éviter les notes *"float to pointer coercion"* de la part du compilateur dans certains cas.
- Être attentif aux notes d'optimisation générées par le compilateur et les résoudre lorsque nécessaire.
- Utiliser l'opérateur spécial `the` dans certains cas d'expressions arithmétiques pour indiquer au compilateur le bon type.
- Ne pas utiliser l'opérateur `/` sur les entiers (qui peut renvoyer un ratio) mais plutôt `truncate`.
- Utiliser le profiler intégré a CMU Common Lisp pour isoler les fonctions utilisant beaucoup de temps CPU et les optimiser en priorité. Le profiler permet aussi de trouver les fonctions générant du consing. Une attention toute particulière doit être portée aux algorithmes de transformation/tracé/éclairage et à la bibliothèque mathématique.
- Éventuellement créer un nouveau fichier si l'ensemble de fonctionnalité rajouté le justifie (par exemple pour la gestion des textures).
- Bannir l'usage des listes lorsque la performance est en jeu et utiliser un tableau de type `simple-array` lorsque les éléments du tableau doivent être accédés très souvent. Les structures sont aussi très performantes.

3.2 Exemple : ajout de tracé avec remplissage de polygones à 4 côtés (quadrilatère)

Il faut :

- Dans Render.lisp, écrire la famille primitives qui implémentent l'algorithme de tracé avec remplissage d'un quadrilatère (quad). Le quad pourra éventuellement être divisé en 2 triangles afin de réutiliser les primitive de remplissage de triangle existantes. Pour rester homogène avec le nommage des primitive de remplissage de triangles, on pourra appeler la famille de primitives de tracé `fillpoly4` (remplissage avec une couleur uniforme)¸ `fillpoly4z` (remplissage uniforme avec zbuffer), `fillpoly4gour` (remplissage avec ombrage de gouraud) et
`fillpoly4zgour` (remplissage avec ombrage gouraud et zbuffer). Dans un premier temps il n'est pas indispensable d'écrire les 4 fonctions simultanément.
- Dans VertexBuffer.lisp, écrire la fonction `draw-quads` pour dessiner la succession de quads spécifiée dans le corps du `Begin`.
- Autoriser le nouveau mot-clé `:QUADS` comme mode de tracé valide en premier argument de la macro Begin. Pour cela il faut ajouter le mot-clé `:QUADS` à la fin de la liste des modes valides pour `Begin` dans la liste `+begin-drawing-keys+` :

```
(defconstant  +begin-drawing-keys+
              '( :POINTS :LINES :LINE-STRIP :LINE-LOOP
               :TRIANGLES :TRIANGLE-STRIP :TRIANGLE-FAN
               :QUADS))
```

- Ajouter la fonction `draw-quads` à la fin de la liste `+begin-drawing-funcs+` :

```
(defconstant  +begin-drawing-funcs+
              '(draw-points draw-lines draw-line-strip
              draw-line-loop draw-triangles draw-triangle-strip
              draw-triangle-fan draw-quads))
```

- Tester la nouvelle primitive de tracé, par exemple :

```
(Begin   :QUADS
        (Vertex 0.0 0.0 0.0 1.0)
        (Vertex 1.0 0.0 0.0 1.0)
        (Vertex 1.0 1.0 0.0 1.0)
        (Vertex 0.0 1.0 0.0 1.0))
```