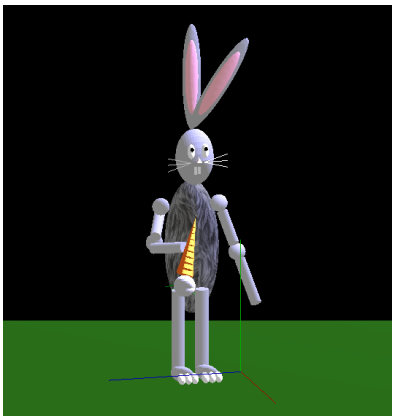
NICOLLE Thomas
BELASSEL Meryem

Synthèse d'images : Rapport du projet lapin



Introduction

L'objectif principal de ce projet est de créer une représentation en 3D d'un lapin en utilisant des techniques avancées de modélisation, d'habillage, et d'animation à l'aide de la technologie OpenGL et les différentes librairies GLU, glut et jpeg.

1. Modélisation du Lapin

Pour modéliser notre lapin, l'approche adoptée consiste à utiliser des primitives solid GLUT et paramétriques. Cela nous permet de créer des formes personnalisées par opposition à aux solid GLUT et sans recourir à des quadriques.

a. Représentation paramétrique

- Sphère et Ellipsoïde

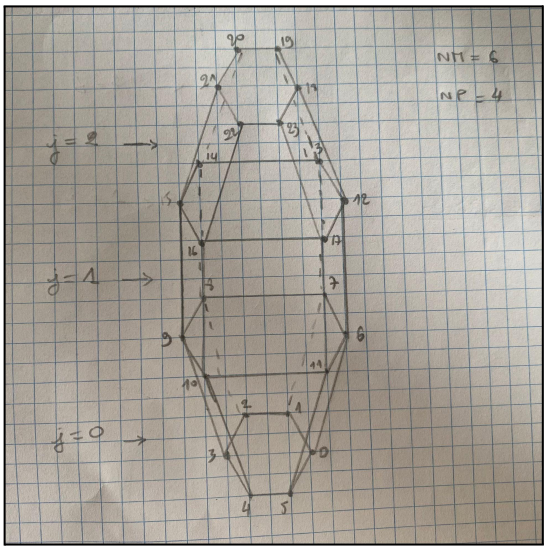
Pour représenter le corps du lapin et plusieurs de ses membres tels les oreilles, la tête, les yeux, les pattes et les orteils, nous avons employé une primitive créée à partir de la représentation paramétrique d'une ellipsoïde ; qui est d'ailleurs inspiré de notre sphère déjà réalisé dans nos tds et tps.
L'équation paramétrique d'une sphère de centre $O=[x_0,y_0,z_0]$ et de rayon R est :

$x=R.\cos(\theta).\cos(\varphi)$,
$y=R.\sin(\theta).\cos(\varphi)$,
$z=R.\sin(\varphi)$

Avec : $\theta = i.2.\frac{\pi}{NM}$ et $\varphi = -\frac{\pi}{2} + j.\frac{\pi}{NP-1}$

Nous avons donc créé et implémenté notre fonction maSphere() pour générer les sommets, les faces et dessiner l'objet.
Tout d'abord, une première série de deux boucles itératives s'occupe de calculer les coordonnées (x, y, z) de chaque sommet (des faces) en fonction des angles *theta* et *phi* à partir de l'équation paramétrique.
Un *point S* récupère ces coordonnées, et est ajouté dans le tableau *sommets[NM * (NP + 1)]*.
Ensuite, une deuxième série de deux boucles *for* est utilisée pour remplir le tableau *faces* des 4 indices des sommets constituant chaque face de la sphère.

Faces				
j = 2 j = 2	17	18	18	23
	16	17	23	22
	15	16	22	21
	14	15	21	20
	13	14	20	19
	12	13	19	18
j = 1	11	12	12	17
	10	11	17	16
	9	10	16	15
	8	9	15	14
	7	8	14	13
	6	7	13	12
j = 0	5	0	6	11
	4	5	11	10
	3	4	10	9
	2	3	9	8
	1	2	8	7
	0	1	7	6
Généralisation de la syntaxe	i +j*6	(i+1) %NM + j* NM	(i+1)% NM + (j+1)* NM	i % 6 + (j+1)* NM



➔ i1: C'est l'indice du sommet en position (i, j), qui est exprimé comme suit: i + j* NM

➔ i2: C'est l'indice du sommet en position (i + 1, j), qui est exprimé comme suit : (i+1)% NM + j * NM

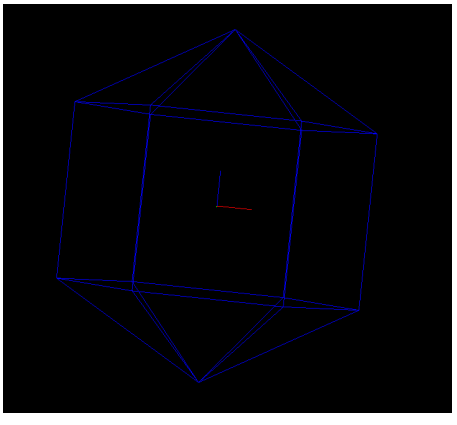
➔ j1: C'est l'indice du sommet en position (i + 1, j + 1), qui est exprimé comme suit :(i+1)% NM + (j+1) * NM

➔ j2: C'est l'indice du sommet en position (i, j + 1), qui est exprimé comme suit : i + (j+1) * NM

Les indices (i1, i2, j1, j2) sont calculés en fonction des indices i et j, formant ainsi un quadrilatère pour chaque paire de méridiens et parallèles. Ces indices sont ensuite stockés dans le tableau faces[NM*NP][4].

Une dernière série de boucles est utilisée pour itérer sur les méridiens et les parallèles, puis chaque sommet de la face est affiché avec glVertex3f() pour former le quadrilatère et enfin afficher notre sphère.

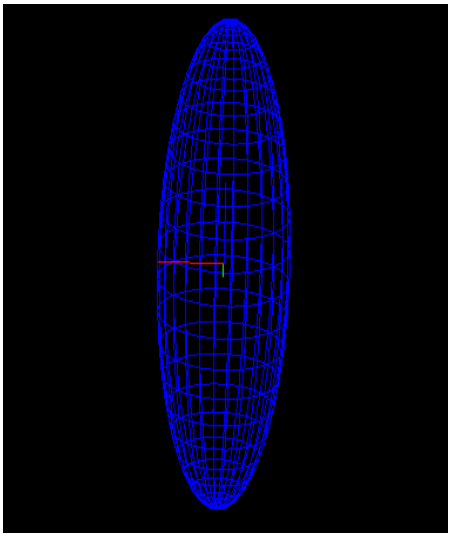
Affichage :

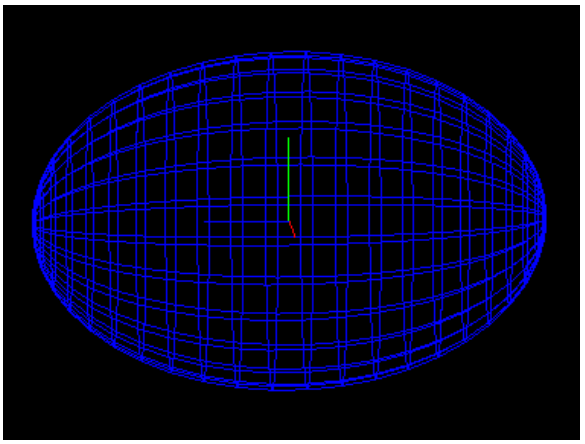


Nous avons plus tard utiliser cette fonction pour créer notre ellipsoïde qui est défini par trois rayons différents et par une équation paramétrique comme suit:

x=***RX***.cos(θ).cos(φ),
y=***RY***.sin(θ).cos(φ),
z=***RZ***.sin(φ)

Avec : $\theta = i.2.\frac{\pi}{NM}$ et $\varphi = -\frac{\pi}{2} + j.\frac{\pi}{NP-1}$





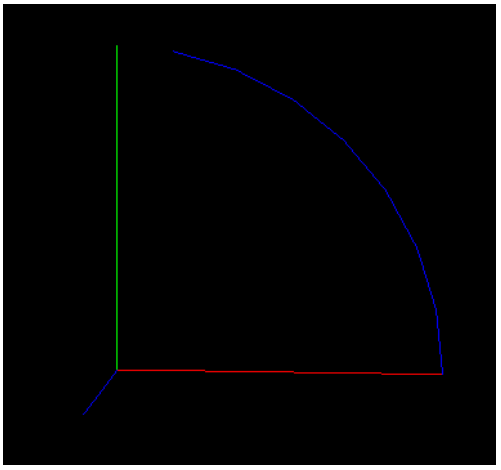
Le fait de pouvoir manipuler les différents rayons nous a permis de modéliser différentes formes associées à chaque partie du corps, par exemple, nous avons réalisé le corps du lapin en élargissant un peu plus le rayon suivant l'axe de Z et en gardant la même valeur pour les rayons suivants les axes X et Y, etc.

- Arc

Pour nous permettre de dessiner les fanes d'une carotte, nous avons eu besoin de créer une fonction pour une nouvelle forme géométrique : un arc. Nous avons utilisé l'équation paramétrique suivante:

$x = r\,.\,cos(\theta)$
$y = r\,.\,sin(\theta)$
$z = 0$

Avec $\theta = \{0, \frac{\pi}{2}\}$

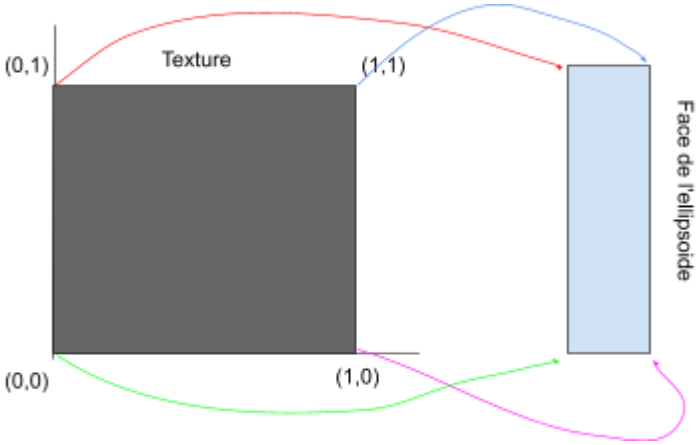


b. Représentation avec primitive

Dans cette section, nous avons utilisé diverses primitives de l'interface GLUT pour modéliser les membres du lapin. Pour représenter les bras, nous nous sommes inspirés de la structure d'un bras robotique, utilisant des rotules et des cylindres, et avons employé les fonctions *glutSolidCylinder* et *glutSolidSphere*. Le nez a été créé à l'aide de *glutSolidTetrahedron*, tandis que les dents ont été réalisées sous forme de cubes. Ces formes primitives ont subi des transformations telles que la mise à l'échelle, la rotation et la translation afin de répondre à nos exigences de modélisation.

2. Habillage du Lapin

a. Placage de texture sur une face



Le placage de texture consiste à placer une texture créée à partir d'une image. Pour procéder à cette opération, il faut suivre les étapes suivantes :

Génération des coordonnées de texture : À chaque sommet généré, il faut associer des coordonnées de texture, qui varient de 0 à 1 dans les deux directions.

Activation de la texture : Il faut activer l'utilisation de la texture avec glEnable(GL_TEXTURE_2D).

Génération la texture : Dans le main, nous avons chargé la texture à l'aide de la fonction loadJpegImage, qui a comme arguments le fichier (donc notre image) et le tableau qui contiendra la texture créée. Dans cette fonction, nous allons créer la texture que l'on veut en faisant appel à deux boucles itératives qui se suivent, qui vont

- La boucle externe parcourt les lignes de l'image.
- La boucle interne parcourt les colonnes de l'image.
- À l'intérieur de la boucle, les lignes suivantes copient les composantes RGB du pixel à la position (i, j) dans l'image (image) vers la position correspondante (i, j) dans le tableau t.
    - t[i][j][0] prend la valeur du composant rouge du pixel à la position (i, j) dans l'image.
    - t[i][j][1] prend la valeur du composant vert.
    - t[i][j][2] prend la valeur du composant bleu.

Ensuite, on génère cette texture avec la fonction glGenTextures() qui prend comme arguments le nombre de noms de textures à générer et un tableau où les noms générés seront stockés, puis on fait appel à la fonction glBindTexture() qui s'occuper de lier une texture spécifique à un type de texture cible. Ella a donc comme paramètres: GLenum cible (GL_TEXTURE_1D, GL_TEXTURE_2D, GL_TEXTURE_3D, etc.), GLuint texture(le nom de la texture qu'on souhaite lier, et elle est normalement générée par glGenTextures)

Définition de la texture : La configuration des paramètres de texture, en la chargeant et en l'associant à notre primitive paramétrée :
Les appels à glTexParameteri sont utilisés pour définir les paramètres de filtrage de la texture. Ces paramètres affectent la manière dont OpenGL traite les pixels de texture lorsqu'ils sont agrandis ou réduits.

- *glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);*
  Cet appel configure le mode de filtrage, c'est-à-dire lorsque les pixels de texture sont agrandis. Les paramètres possibles pour *GL_TEXTURE_MAG_FILTER* sont:
  - GL_NEAREST : le pixel le plus proche dans la texture est utilisé pour chaque pixel agrandi. C'est une méthode rapide mais peut donner des résultats pixelisés.
  - GL_LINEAR : donne une apparence plus lisse mais peut être plus lente.
- glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
  Cet appel configure le mode de filtrage, c'est-à-dire lorsque les pixels de texture sont réduits. Les paramètres possibles pour *GL_TEXTURE_MIN_FILTER* sont les mêmes que pour *GL_TEXTURE_MAG_FILTER*.

Chargement des données de l'image dans la texture : Après la configuration de la texture, on va charger l'image par l'intermédiaire de la fonction suivante :

glTexImage2D(target, level, components,
w, h, border, format, type, texels);
target: GL_TEXTURE_2D
level: nombre de résolutions utilisées pour le mipmapping//0
components: éléments par image GL_RGB
w, h: largeur et hauteur de l'image (256x256
border: largeur de la bordure, dans notre cas, 0
format and type: description de l'image (Gl_RGB)
texels: pointeur sur le tableau de la texture ***texture1*** (créée à partir de l'image)

Enfin, pour pouvoir implémenter les deux types de textures et bien les manipuler comme nous voudrions, nous avons associé une touche à ce type de texture (**'F'**).

b. Texture enroulée d’un objet

Concernant cette section, nous allons reprendre ce qui était déjà détaillé lors de la partie précédente, tout en ajoutant quelques détails et modifications.

Dans la partie où nous dessinons l'ellipsoïde, nous allons changer cette fois-ci les glTexCoord2f() pour bien envelopper l’objet avec la texture souhaitée.
Les valeurs de u0, u1, v0, et v1 dans ce code représentent les coordonnées de texture utilisées pour envelopper la texture sur les sommets d'un quadrilatère dans OpenGL.

- u0 : C'est la coordonnée de texture horizontale du coin en bas à gauche du quadrilatère.
- u1 : C'est la coordonnée de texture horizontale du coin en bas à droite du quadrilatère.
- v0 : C'est la coordonnée de texture verticale du coin en bas à gauche du quadrilatère.
- v1 : C'est la coordonnée de texture verticale du coin en haut à gauche du quadrilatère.

Ces coordonnées de texture sont définies en fonction des indices de boucle i et j ainsi que des dimensions NM et NP.

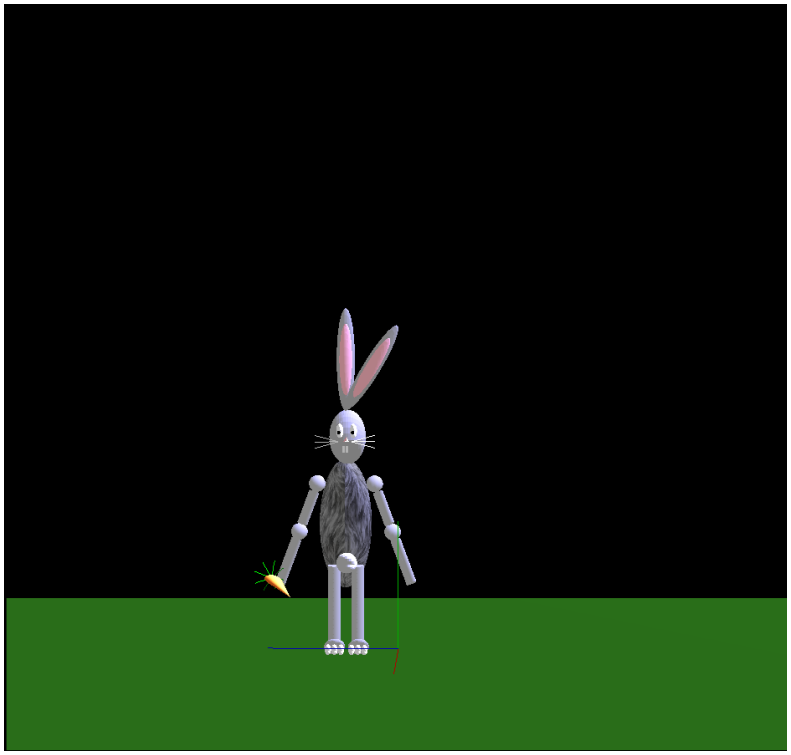
Par défaut, cette texture est appliquée sur le lapin. Au cas où, elle est désactivée ou l’autre texture est appliquée, nous pouvons l’activer en cliquant sur le bouton (**‘E’**).

3. Gestion des Lumières

a. Lumière ponctuelle simple

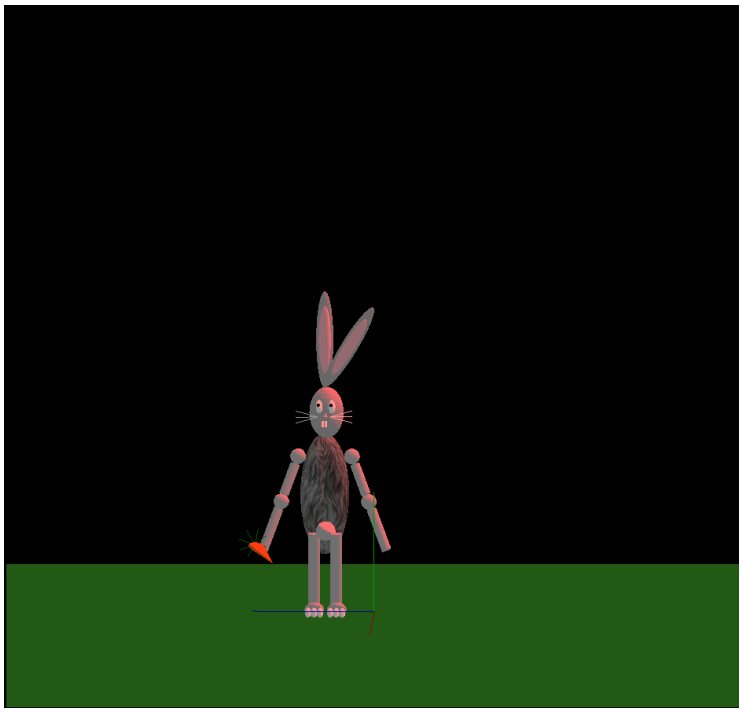
Pour créer une source lumineuse dans notre scène, nous avons utilisé la fonction *Lumiere()*. Dans cette dernière, nous avons défini plusieurs propriétés pour la source de lumière. Tout d'abord, nous avons spécifié les coordonnées homogènes de la position de la source lumineuse avec *position_source0[]*, où les trois premières valeurs définissent la position spatiale de la lumière et la dernière valeur (1.0) indique que la source lumineuse est une source ponctuelle. Ensuite, nous avons défini les composantes diffuse (dif_0[]), ambiante (amb_0[]), et spéculaire (spec_0[]) de la lumière. Ces valeurs déterminent respectivement la couleur de la lumière diffusée, ambiante et spéculaire.
En activant *GL_LIGHTING* et *GL_COLOR_MATERIAL*, nous avons activé le calcul d'éclairage et l'application des couleurs matérielles. Ensuite, nous avons utilisé la fonction glLightfv() pour spécifier les propriétés de la lumière, telles que la position, les

composantes ambiante, diffuse et spéculaire. La lumière que nous avons créée est la lumière GL_LIGHT0.



Cette configuration permet d'illuminer la scène avec une source lumineuse positionnée en (5.0, 8.5, 2.0) et émettant une lumière diffuse bleue (dif_0[]), une lumière ambiante grise (amb_0[]), et une lumière spéculaire blanche (spec_0[]). En appliquant ces propriétés, nous obtenons une illumination réaliste de notre scène et une présentation de notre lapin d'une façon esthétique.
Afin d'illuminer notre lapin correctement, nous sommes obligés de passer par la normalisation des vecteurs normaux de nos faces, car l'intensité de la lumière diffuse dépend de chaque face, ça fonctionne de la même façon pour la lumière spéculaire. Dans nos deux fonctions *maSphere()* et *monEllipsoide()*, nous avons tout d'abord calculer les deux vecteurs entre 2 sommets de nos faces puis nous avons calculé la normale de ces deux vecteurs à l'aide de la fonction *ProduitVectoriel(v1,v2)* et nous les ajoutons au tableau *faceNormales[]* qui va être ensuite sollicité dans l'affichage de la primitive paramétrée et donc qui va permettre de spécifier la normale à chaque face dessinée.
Cette lumière est activée lorsque l'on appuie sur le bouton **'1'** , et désactivée avec **'0'**.

b. Lumière de type SPOT:



Contrairement à la lumière précédente, qui elle illumine la scène en son entièreté, la lumière type spot permet de mettre en place une source lumineuse dans la scène, et qui va éclaire dans une seule direction.
Nous avons d'abord défini les paramètres spécifiques à la lumière spot, tels que sa position (`spot_position`), sa direction (`spot_direction`), et ses propriétés de diffusion (`dif_1`), ambiante (`amb_1`), et spéculaire (`spec_1`). Ces paramètres

influent sur l'apparence de la lumière spot dans la scène.

Ensuite, dans la fonction `Lumiere_Spot()`, nous avons activé l'éclairage *GL_LIGHTING* et la gestion automatique des couleurs *GL_COLOR_MATERIAL*. Les propriétés de la lumière spot ont été spécifiées en utilisant les appels de fonction *glLightfv* et *glLightf* pour définir respectivement les tableaux de valeurs et les valeurs uniques.

Les paramètres spécifiques à la lumière spot, tels que sa position (juste au dessus de notre lapin), sa direction (vers le bas), l'angle d'ouverture du cône '*spot_cutoff'* (dans notre cas, le spot av illuminer jusqu'à 60° autour de son axe) puis l'exposant de la lumière spot '*spot_exponent'*, qui lui correspond au coefficient de l'atténuation angulaire.
Pour activer cette lumière, on presse le bouton **'2'** et on la désactive avec **'.'**.

4. Contrôle de la vue

a. Zoom 2D

Pour effectuer la simulation d'un rapprochement ou éloignement, on utilise la fonction glOrtho() permettant de créer une matrice de projection parallèle pour définir le volume de visualisation. Il fallait faire attention à respecter le ratio largeur/hauteur du volume, et donc nous avons utilisé une unique variable globale pour calculer ce rapport.
Afin de simuler un éloignement, il suffit donc seulement d'incrémenter notre variable globale pour augmenter le volume de vision et donc automatiquement nous éloigner de l'origine où se trouve notre lapin. Le fonctionnement est le même pour se rapprocher, c'est-à-dire de diminuer la variable globale ce qui entraîne donc aussi la réduction du volume de vision.

b. Caméra

Pour simuler une vue au niveau de notre espace 3D, nous avons utilisé la méthode gluLookAt() qui nécessite neuf paramètres : (eyeX,eyeY,eyeZ) positionne la caméra dans l'espace, (centerX, centerY, centerZ) qui indique les coordonnées du point regardé et enfin (upX, upY, upZ) indique que l'axe y est orienté vers le haut. Afin de contrôler la vue, il nous faut utiliser la fonction clavierDirection() permettant la détection des pressions sur une touche directionnelles afin d'effectuer les modifications.

Au fur et à mesure que la touche "haut" est pressée, la coordonnée Y de l'œil est incrémentée; inversement si la touche "bas" est pressée, la coordonnée Y de l'œil est décrémenté.

En ce qui concerne les touches "gauche" et "droit", les coordonnées X et Y de la caméra sont re-calculées en fonction d'un angle. L'angle est incrémenté à chaque pression pour aller dans le sens trigonométrique(vers la droite) ou décrémenté pour l'autre sens.
Le point visé étant toujours l'origine, cela simule un déplacement de la vision.

5. Animations

a. Animation à l'aide des touches du clavier

Nous avons choisi pour l'animation déclenchée à l'aide de la touche 'p', de faire mouvoir le bras du lapin qui tient une carotte jusqu'à sa bouche ; la touche 'm' permet de le faire revenir à sa position initiale. Deux variables globales sont nécessaire pour faire varier l'angle de rotation du bras ainsi que de l'avant-bras.

Afin de détecter la pression d'une touche du clavier, nous avons créé une fonction clavierLettres(). Cette dernière, appelle notre fonction d'animation MouvementBras(int) en précisant un entier afin de choisir le sens de l'animation. Pour un sens positif (1) nous allons incrémenter l'angle de l'avant-bras ainsi que l'angle du bras jusqu'a atteindre un certain angle maximum. Inversement, pour un sens négatif (-1) nous faisont une décrémentation des deux angles jusqu'à atteindre l'angle nul initial. Il va de soit, qu'il est nécessaire d'utiliser pour le bras et l'avant-bras la fonction glRotatef() avec les deux constantes afin que les deux parties soit mobile.

b. Animation automatique

Notre animation automatique permet de faire tourner les pupilles à l'intérieur du globe oculaire. Pour réaliser cela, nous avons eu besoin d'une variable globale (pupileY) qui représente l'angle de rotation. Nous utilisons une fonction Pupile_Anim() qui incrémente cet angle jusqu'à atteindre la valeur 360 signifiant que nous avons fait un tour complet ; nous soustrayons ensuite 360 pour avoir une valeur d'angle comprise entre 0 et 360. L'animation est appelée constamment grâce la la fonction glutIdleFunc() qui l'a prend en paramètre, faisant varier en continu cet angle.

C'est à la construction de notre lapin que nous introduisons les deux pupilles, qui sont déplacer à l'aide d'une translation vers le centre des globes oculaires. Ensuite, nous utilisons la combinaison de la fonction glRotatef() avec notre variable globale pour les faire mouvoir. Cependant à ce moment-là, les pupilles tournaient en leur origine, une deuxième translation était nécessaire afin que ces dernières tournent autour de leur origine.

6. Décors

a. Carotte

Tout d'abord, il a fallu modéliser une primitive de l'arc puis une seconde inversé pour construire une double fanes et dupliquer ce modèle 4 fois. Ensuite grâce à un glutSolidCone dessiner le corps de la carotte.

b. Gazon

Afin de mettre en scène notre lapin, nous avons ajouter un terrain de gazon. Pour le construire, il suffit de définir la longueur, largeur et hauteur de la parcelle et de construire les sommet à l'aide de glVertex3f() en spécifiant le type GL_QUADS pour spécifier un quadrilatère. La hauteur est utilisé pour le positionner sur l'axe Y, la longueur sur l'axe Z et enfin la largeur sur l'axe X.

Conclusion

Pour améliorer le rendu visuelle du lapin, il aurrait fallut resoudre le problème de transparence des différentes primitives, ajouter une texture différentes aux autres parties du corps. Pour enrichir davantage l'animation du bras, il aurait été intéressant d'ajouter une animation aux dents, simulant ainsi l'interaction avec la carotte.