RAPPORT DE PROJET

Synthèse D'image

CLUZEL Ophélie | VAUBAN Samuel

Introduction

Le but de ce projet était de modéliser un chat, de l'habiller et de l'animer. Nous avons réalisé ceci en utilisant le langage de programmation C++ ainsi que le logiciel Code::Blocks et les différentes librairies proposées par OpenGL.

Pour commencer nous avons représenté de manière schématique notre chat en le décomposant en plusieurs parties et formes. Vous pouvez retrouver ce schéma en annexe.

Nous avons également modélisé un lampadaire et un sol en plus du chat. Vous pourrez également retrouver leurs schémas en annexe.

Analyse Fonctionnelle

Nous allons maintenant expliquer les différentes parties dont est composé notre programme.

- Partie 1 : Inclusion des librairies

  En effet pour faire de la modélisation avec code::blocks nous allons avoir besoin d'importer plusieurs librairies. Pour cela, nous avons du faire un clic droit sur notre projet puis aller dans build options et enfin dans linker settings où nous avons ajouté les librairies suivantes : GL,GLU, glut et jpeg.

  Nous avons également importé d'autres librairies directement dans le programme via #include. Les voici :

  -#include <cstdio> : cette bibliothèque comprends des fonctions utilisées dans les opérations d'entrée/sortie (exemple : utilisation du clavier,souris...).

  -#include <cstdlib> : cette bibliothèque est la bibliothèque standard du C++.

  -#include <GL/glut.h> : cela permet d'importer GLUT qui est une librairie pour OpenGL (signifiant GL utility toolkit) qui nous permet de gérer le graphisme mais aussi la souris, le clavier...

  -#include <GL/freeglut.h> : freeglut est comme glut une librairie pour OpenGL qui contient des fonctionnalités supplémentaires (notamment l'objet cylindre).

  - #include <jpeglib.h> : cette librairie fournit du code C pour lire et écrire des fichiers image compressés au format JPEG, cela va nous être très utile pour utiliser des images importées comme textures.

- #include <jerror.h> : Cette librairie définit les codes d'erreur et les messages pour la bibliothèque JPEG.

- #include <math.h> : cette librairie contient des fonctions et constantes mathématiques notamment la constante Pi dont nous allons avoir besoin.

- Partie 2 : Déclarations

  On commence par déclarer une classe point qui va nous permettre de créer un objet point avec des coordonnées à 3 dimensions (x, y, z). Elle nous permet aussi de d'associer une couleur (RVB) à ce point. Cette fonction va nous être utile par la suite pour la construction de notre cylindre en primitives.
  Ensuite on déclare les différentes variables nécessaires à la construction du cylindre en primitives (nombre de faces, hauteur, rayon) ainsi que les tableaux qui vont contenir les points du cylindre et les faces du cylindre (4points pour chaque face).
  Ensuite on déclare un tableau image qui va contenir les différentes images qui nous allons importer. Nous avons fixé la taille de cette image à 256x256.
  Ensuite on déclare différentes variables permettant de contrôler l'angle de la caméra et de contenir les coordonnées de la souris.
  On déclare également les différentes variables que nous allons utiliser lors des animations
  Pour finir on déclare les entêtes des différentes fonctions qui seront contenues dans notre programme

- Partie 3 : Main

  La première étape est d'initialiser glut, pour cela on fait appel aux fonctions *glutInit(&argc,argv)* à laquelle on passe en paramètre les arguments de ligne de commande et la fonction *glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH)* qui permet de spécifier le mode d'affichage de la fenêtre, ici on a choisi le double buffer, le mode RGB et le mode avec gestion de la profondeur (parties cachées).Ensuite on va créer une nouvelle fenêtre avec *glutCreateWindow()* après avoir définit sa taille et sa position respectivement grâce aux fonctions *glutInitWindowSize()* et *glutInitWindowPosition()* .

  Ensuite on passe à l'initialisation d'OpenGL, on commence par supprimer la couleur de fond avec *glClearColor(0.0,0.0,0.0,0.0)* et on définit la couleur courante à blanc *glColor3f(1.0,1.0,1.0).* On va ensuite activer différentes fonctionnalités : *glEnable(GL_DEPTH_TEST)* qui va activer le test de profondeur, c'est-à-dire que si un point d'une face se situe derrière une autre face, il n'est pas dessiné et *glEnable(GL_COLOR_MATERIAL)* va quant à elle activer la coloration du matériau de l'objet.

  Ensuite nous allons passer à l'initialisation de nos différentes lumières. On commence par créer une lumière ponctuelle que l'on va placer dans *GL_LIGHT_1.* Pour cela on

définit tous ses paramètres : sa valeur ambiante, diffuse et spéculaire, sa position et sa valeur constante, linéaire et quadratique d'atténuation. Puis on définit son *GL_SPOT_CUTOFF* à 180°, cela représente l'angle sur lequel le spot va éclairer. Ici comme on essaye de recréer une ampoule, on veut qu'elle éclaire tout autour. Du coup on ne lui définit pas de valeur *GL_SPOT_DIRECTION* car elle va aller dans toutes les directions. Pour finir on définit *GL_SPOT_EXPONENT*, la valeur de la qualité de l'affichage de l'image.
Passons ensuite à la lumière 2, qui cette fois sera une lumière directionnelle de couleur rouge. Pour cela on définit tous ses paramètres : sa valeur ambiante, diffuse et spéculaire, sa position et sa valeur constante, linéaire et quadratique d'atténuation.

Ensuite on passe à l'enregistrement des différentes fonctions de rappel.
glutDisplayFunc() prends en paramètre la fonction qui s'occupe de l'affichage,
glutReshapeFunc() prends en paramètre la fonction qui s'occupe du re fenêtrage,
glutKeyboardFunc() prends en paramètre la fonction qui s'occupe des action déclenchées sur une touche type caractère ASCII du clavier,
glutSpecialFunc() prends en paramètre la fonction qui s'occupe des actions délenchées sur une touche spéciale du clavier(flèches,F1,F2…),
glutMouseFunc() prends en paramètre la fonction qui s'occupe des actions déclenchées sur un clic souris,
glutMotionFunc() prends en paramètre la fonction qui s'occupe des actions liées à un déplacement souris bouton enfoncé,
enfin *glutTimerFunc()* qui prends en paramètre une fonction qui s'occupe de l'actualisation de la scène selon un temps imparti. Nous aurions pu utiliser à la place la fonction *glutIdleFunc()* mais nous avons préféré utiliser un timer afin de pouvoir contrôler la vitesse de l'animation.

Pour finir on va rentrer dans la boucle principale glut avec *glutMainLoop()* et on retourne 0 car c'est la fin du main.

- Partie 4 : Fonction Affichage

On commence par initialiser la projection et la caméra. Pour cela on change le mode de la matrice courante avec *glMatrixMode(GL_PROJECTION)* pour spécifier que l'on va travailler sur la projection, puis on charge la matrice identité dans la matrice courante avec *glLoadIdentity()* et enfin on définit une nouvelle projection orthographique avec *glOrtho()* , cela multiplie la matrice courante avec notre matrice orthographique et place le résultat dans la matrice courante. Ensuite, on change à nouveau le mode de la matrice courante pour maintenant travailler sur la caméra avec *glMatrixMode(GL_MODELVIEW)* , on charge encore une fois la matrice identité puis on effectue différentes transformations sur celle-ci (translation, rotations…).

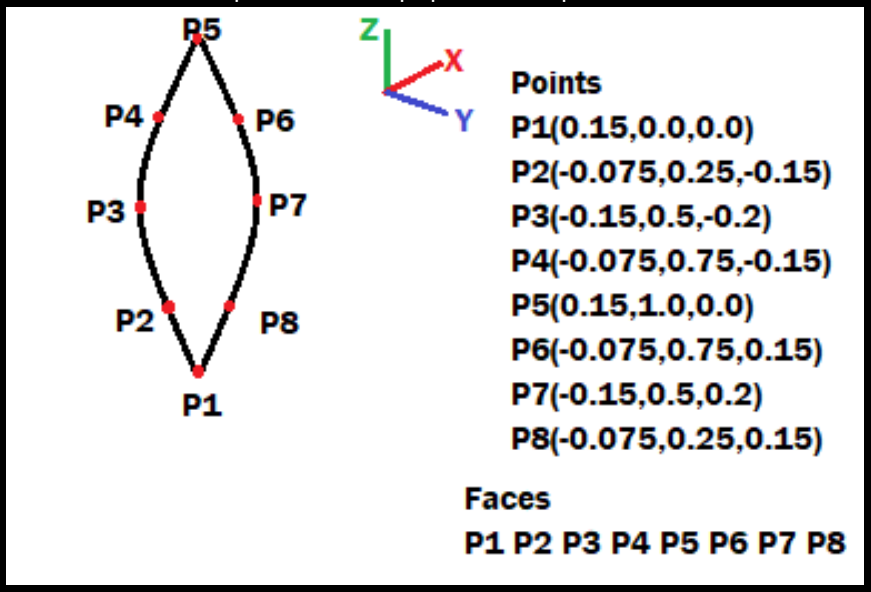
Ensuite on va réinitialiser les buffers avec *glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)*, on réinitialise également la couleur de fond avec *glClearColor(0.0,0.0,0.0,0.0)* (fond noir) .

Modélisation du chat

On commence par délimiter notre objet chat entre *glPushMatrix()* qui va mettre la matrice courante sur la pile et *glPopMatrix()* qui va retirer la matrice de la pile. A chaque partie que l'on voudra indépendante du reste on utilisera ces deux fonctions pour l'encadrer ainsi on pourra travailler sur certaines parties groupées notamment pour les animations.

On commence par modéliser la tête du chat, on ouvre un nouveau couple Push/PopMatrix on crée un objet sphère déjà existant grâce à *glutSolidSphere(rayon,long,lat)* de la bibliothèque glut, on a plus qu'à définir le rayon et le nombres de lignes de longitude et de latitude que l'on veut. Ensuite on la déplace avec un *glTranslatef()* et on lui applique une couleur avec *glColor3f()* . Il faut faire attention à l'ordre dans lequel on écrit les différentes transformations sur nos objets car elles s'effectuent à l'envers. Sur le même principe on lui ajoute un museau, un nez et des yeux. Pour les pupilles nous avons décidé d'utiliser des primitives, nous avons donc fait appel à *glBegin()* avec en paramètre GL_POLYGON pour construire une forme pleine. On crée nos différents points dans un ordre précis avec *glVertex3f()*.A la fin de notre construction on ne doit pas oublier d'appeler *glEnd()* .

Voici un schéma représentant la pupille et ses points :

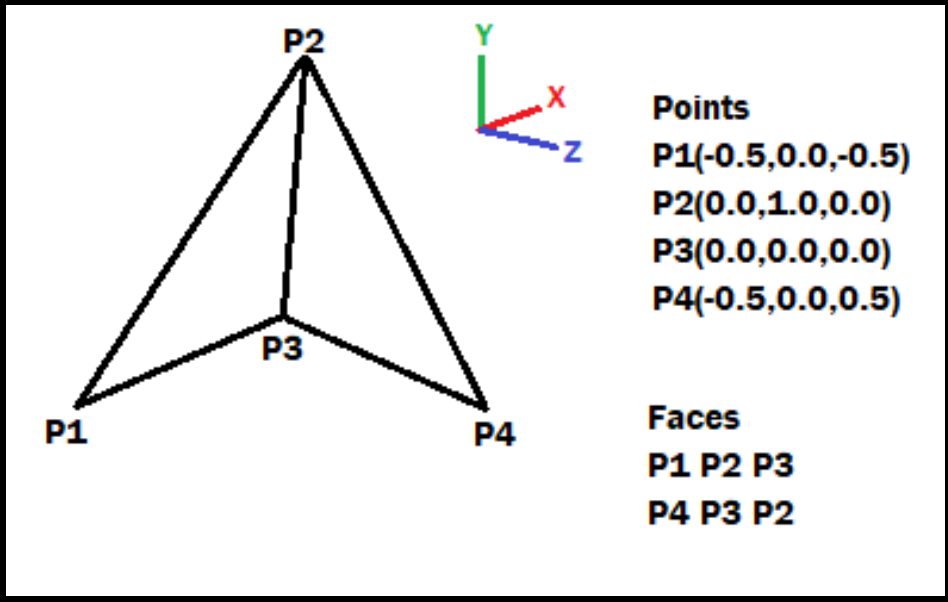


Pour finir on la redimensionne avec *glScalef()* et on la déplace avec *glTranslatef()*.

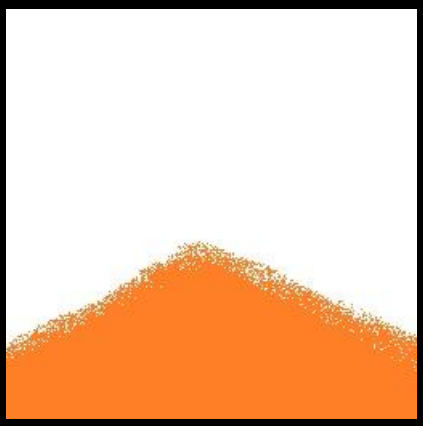
Pour les moustaches on utilise également les primitives mais cette fois-ci avec le paramètre GL_LINES car on veut juste construire des lignes entre deux points.

Pour les oreilles on va utiliser des primitives avec le paramètre GL_TRIANGLE_STRIP dans le *glBegin()* car on veut construire des triangles.

Voici un schéma représentant les points et les faces des oreilles :

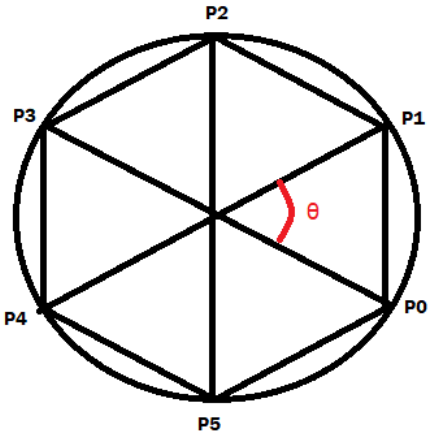


Mais nous allons également appliquer une texture sur les oreilles. Pour cela on fait appel à la fonction *loadJpegImage("./oreille.jpg")* que l'on explicitera plus tard mais qui nous permet d'importer une image jpeg dans notre programme. Ensuite on déclare une nouvelle texture 2D associée à cette image grâce à *glTexImage2D()* et enfin on oublie pas d'activer le plaquage de texture 2D avec *glEnable(GL_TEXTURE_2D)*. Ensuite dans le *glBegin()* on va associer à chaque point un *glTexCoord2f()* qui correspondra à la coordonnée de la texture associée à ce point.
Notre texture est la suivante :



Nous l'avons réalisé sur Paint. Une fois appliquée sur notre oreille cela nous donne :

Passons ensuite au cou du chat, nous avons décidé de le représenter par un cylindre en primitive. Pour cela on commence par remplir les tableaux de points et de faces associés à notre cylindre que nous avions déclaré plus haut. Pour cela on utilise des boucles for. La méthode que nous avons utilisée pour déterminer les coordonnées des points est expliquée dans les schémas ci-dessous :



Angles θ:

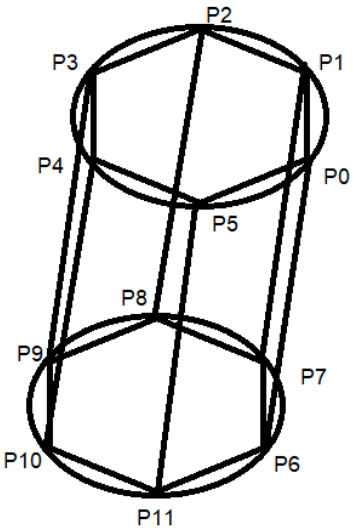
θ0= 0 = 0* 2π/6
θ1= π/3 = 1* 2π/6
θ2= 2π/3 = 2* 2π/6
θ3= π = 3* 2π/6
θ4= 4π/3 = 4* 2π/6
θ5= 5π/3 = 5* 2π/6

Comme ici on a n=6(nombre de faces), on peut écrire en règle générale que:

θ0= 0+ i2π/n

On peut donc en déduire les coordonnées des points i:

x[i]= r*cos(θ)= r*cos(i2π/n)
y[i]=r*sin(θ)= r*sin(i2π/n)



Pour les coordonnées des points c'est le même principe qu'avec le cercle mais avec une coordonnée z en plus. En partant du principe que le point 0,0,0 est au centre du cylindre on a donc:

Pour les points P0 à P5:

x[i]= r*cos(i2π/n)
y[i]= r* sin(i2π/n)
z[i]=h/2

Pour les points P6 à P11:

x[i]= r*cos(i2π/n)
y[i]= r* sin(i2π/n)
z[i]=-h/2

On peut donc écrire :

x[i]= r*cos(i2π/n)
y[i]= r* sin(i2π/n)
z[i]=h/2
x[i+n]= r*cos(i2π/n)
y[i+n]= r* sin(i2π/n)
z[i+n]=-h/2

On énumère les faces du cylindre, ici on a n=6 faces:

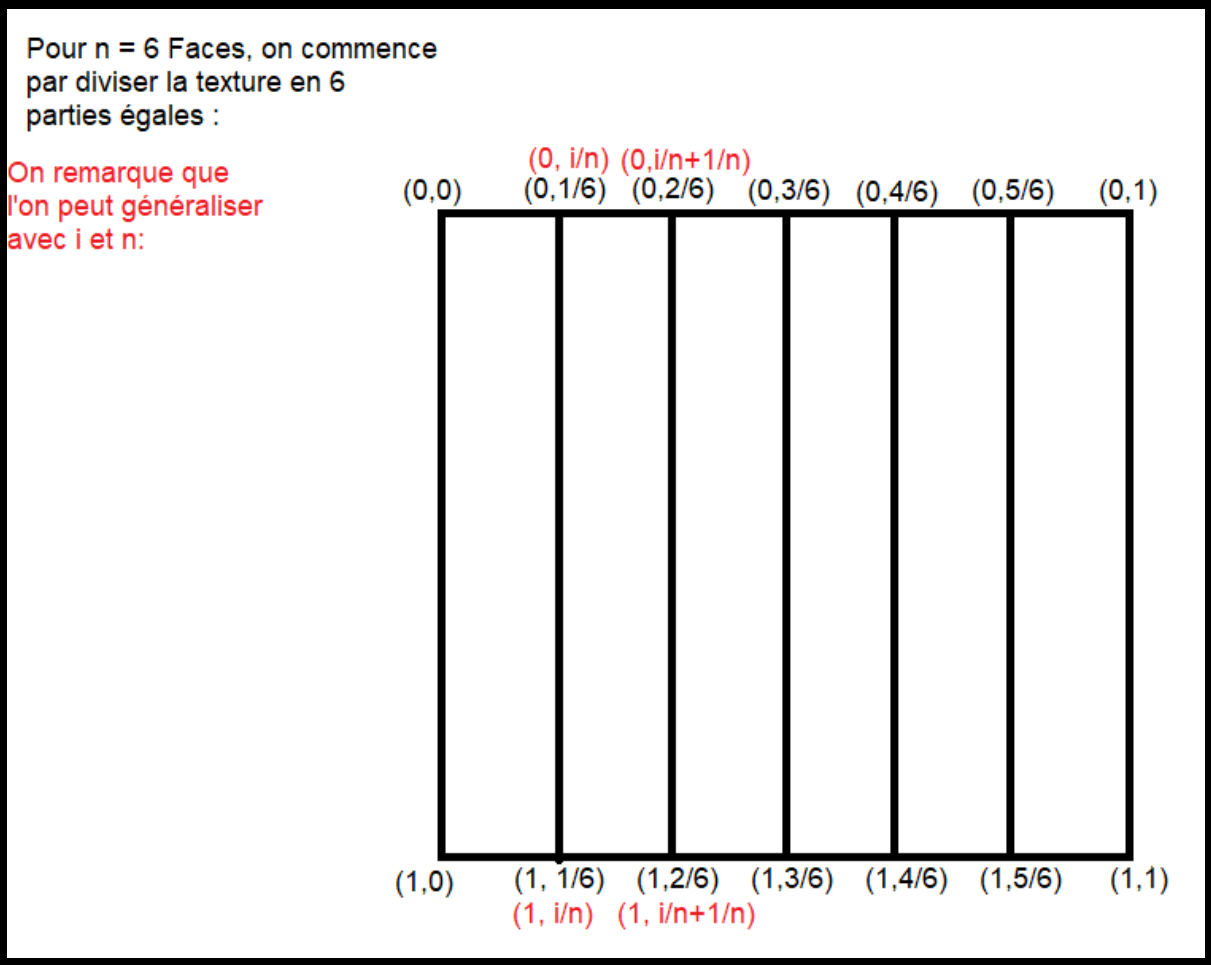
i	Faces			
0	P0	P6	P7	P1
1	P1	P7	P8	P2
2	P2	P8	P9	P3
3	P3	P9	P10	P4
4	P4	P10	P11	P5
5	P5	P11	P6	P0

i i+n (i+1)%n+n (i+1)%n

On remarque que l'on peur généraliser les faces pour n'importe quel n, on peut donc écrire :

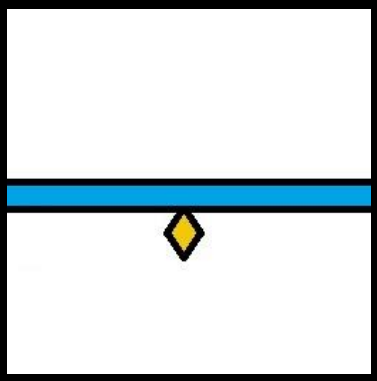
Face[i][0]=i
Face[i][1]=i+n
Face[i][2]=(i+1)%n+n
Face[i][3]=(i+1)%n

Une fois cela fait nous devons maintenant passer au dessin des points et au plaquage de la texture. Nous avons décidé d'enrouler la texture autour du cylindre mais nous avons également réalisé le plaquage de texture face par face que nous avons laissé en commentaire dans le code de notre programme. Pour cela on commence par charger la texture voulue avec *loadJpegImage()*, on déclare cette image comme texture avec *glTexImage2D()* et on active le plaquage de texture avec *glEnable(GL_TEXTURE_2D)* . Puis on lance une boucle for de i à n le nombre de faces choisit puis on utilise *glBegin()* avec le paramètre GL_POLYGON et on construit les 4 points de chaque face à partir des données rentrées dans nos tableaux que l'on entre dans *glVertex3f()*. On leur associe également à chacun un *glTexCoord2f()* qui contient les coordonnées de texture qui lui sont associé. Pour déterminer les coordonnées de textures pour faire une texture enroulée on a dû diviser la texture en n parties. Voir l'explication dans le schéma ci-dessous :



On est obligé de transtyper avec (float) devant les divisions pour éviter que le programme calcule des divisions entières et que le résultat soit faussé.

La texture que nous avons décidé d'enrouler est la suivante :



Nous l'avons réalisée sur paint.

Et une fois plaquée sur le cou du chat on obtient :

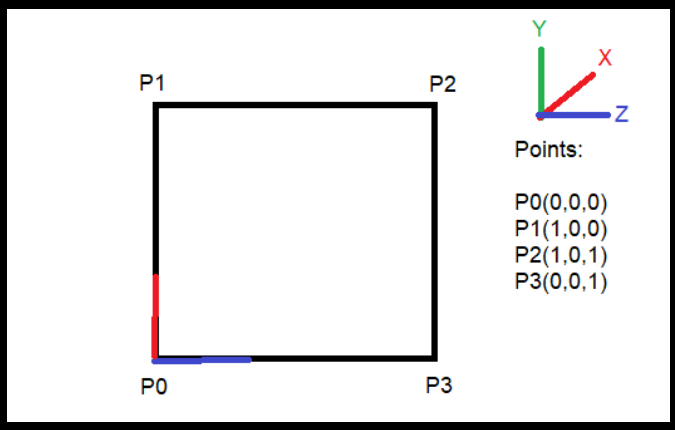
Passons ensuite au corps de notre chat, pour le réaliser nous avons utilisé *glutSolidSphere()* puis nous avons utilisé un *glScalef()* avec une plus grande valeur en x pour qu'elle ait un aspect allongé plus ovale. On la déplace à l'endroit voulu et on lui applique une couleur.

Pour construire la queue on réalise un enchainement de *glutSolidSphere()* et de *glutSolidCylinder()* qui représentent les différentes articulations et morceaux de la queue. On sépare une partie de la queue du reste avec *glPushMatrix()* et *glPopMatrix()* car on appliquera une animation sur celle-ci que nous verrons plus tard.

Pour les pattes on utilise une *glutSolidSphere()* pour faire la cuisse, un *glutSolidCylinder()* pour la jambe et encore une *glutSolidSphere()* pour le pied. On fait de même pour les 4 pattes on change un peu les formes avec des *glScalef()* et on les déplace à l'endroit voulu avec *glTranslatef()*.

Modélisation du sol

Pour le sol on va réaliser un carré plat simple avec *glBegin()* et le paramètre GL_POLYGON. Voici le schéma du sol :



On lui applique également une texture, pour que ce soit plus réaliste on rajoute comme paramètres

glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_REPEAT) et *glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_REPEAT)* qui vont faire en sorte que la texture se répète si on déborde de ces coordonnées.

On ajoute également *glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST)* et *glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST)*. Étant donné que les coordonnées de texture sont indépendantes de la résolution, elles ne correspondent pas toujours exactement à un pixel. Cela se produit lorsqu'une image de texture est étirée au-delà de sa taille d'origine ou lorsqu'elle est réduite. OpenGL propose différentes méthodes pour choisir la couleur échantillonnée lorsque cela se produit. Ce processus s'appelle le filtrage et on a choisit la méthode GL_NEAREST qui retourne le pixel le plus proche des coordonnées.

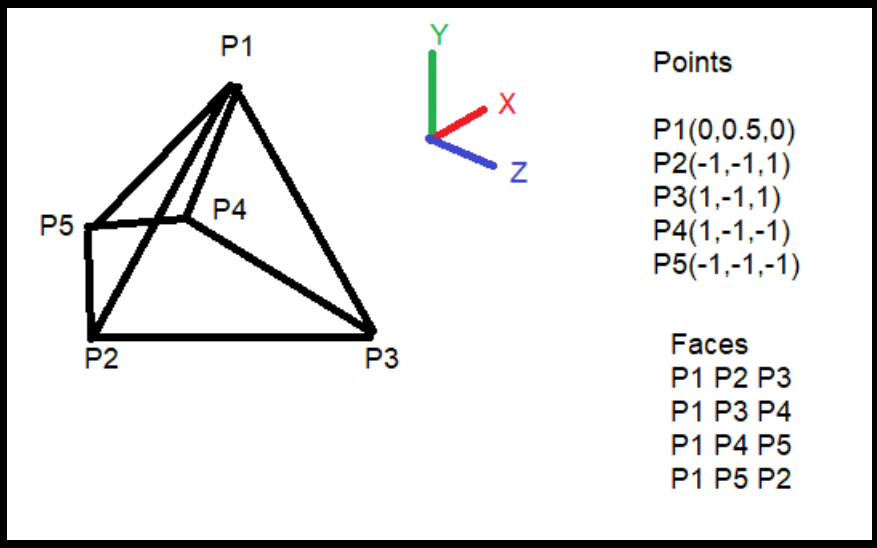
Notre texture est la suivante :

Une fois appliquée au sol cela nous donne :

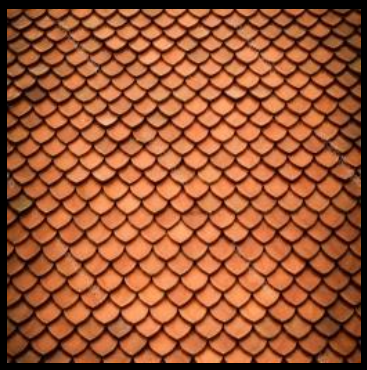
Modélisation du lampadaire

Pour finir nous avons modélisé un lampadaire. Il se compose tout d'abord d'un poteau réalisé à l'aide s'un simple *glutSolidCylinder().* Puis d'une partie lampe réalisée en deux parties avec des primitives.

La partie haute utilise *glBegin()* avec le mode GL_POLYGON. On construit ses faces puis on leur plaque une texture.

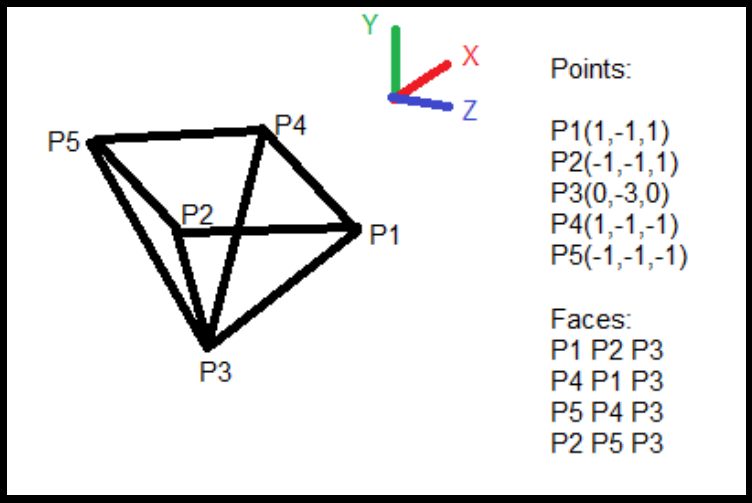


Notre texture est la suivante :



Une fois appliquée on obtient :

Ensuite on construit la partie basse de la lampe avec *glBegin()* mais cette fois avec le mode GL_LINE_STRIP. On change l'épaisseur des lignes pour qu'elles soient plus épaisses avec *glLineWidth().*



Pour finir on ajoute une ampoule qui est une *glutSolidSphere()*.

Comme on a fini de modéliser on ajoute à la fin de notre fonction affichage *glFlush()* qui garantit que les commandes OpenGL précédentes soient terminées dans un temps fini. Puis comme nous sommes dans une fenêtre en double buffer on appelle *glutSwapBuffers()* qui échange les buffers, cela fait en sorte que le contenu du buffer arrière de la couche utilisée par la fenêtre en cours devienne le contenu du buffer avant. Le contenu du tampon arrière devient alors indéfini.

- Partie 5 : Fonction Clavier

void clavier(unsigned char touche,int x,int y) Cette fonction prends en paramètre le caractère ASCII correspondant à la touche qui vient d'être tapée et x et y les coordonnées de la souris au moment où la touche est tapée. On fait un *switch(touche)* pour programmer plusieurs actions en fonction des touches.

case 'p': Lorsque la touche p est tapée, on affiche notre scène en mode polygone plein. Pour cela on appelle la fonction *glPolygonMode(GL_FRONT_AND_BACK,GL_FILL)*, GL_FILL indiquant le mode plein.

case 'f': Lorsque la touche f est tapée, on affiche la scène en mode fil de fer. Pour cela c'est le même principe que précédemment mais avec le paramètre GL_LINE.

case 's' : Lorsque la touche s est tapée, on affiche la scène en mode sommets seuls. Pour cela c'est le même principe que précédemment mais avec le paramètre GL_POINT.

case 'd' et 'D ' : Lorsque la touche d est tapée, on va activer le test de profondeur, c'est-à-dire que si un point d'une face se situe derrière une autre face, il n'est pas dessiné. On utilise pour cela *glEnable(GL_DEPTH_TEST)*. Et lorsque la touche D est tapée, on le désactive avec *glDisable(GL_DEPTH_TEST)*.

case 'q' : Lorsque la touche q est tapée, on quitte le programme avec *exit(o)* .

case 'o' : Lorsque la touche o est enfoncée, cela va animer la queue de notre chat. Pour cela nous avions placé une partie de la queue de notre chat entre glPushMatrix() et gl PopMatrix() afin de pouvoir

appliquer la même transformation à toute une partie. On applique donc nos transformations : *glTranslatef(8.5,1.3,0.0);* Permet de remettre la queue à sa place initiale.

glRotatef(angle3,1.0,0.0,0.0); Rotation en x d'une valeur angle3.

glTranslatef(-8.5,-1.3,0.0); Retour à l'origine pour pouvoir effectuer la rotation correctement.

Maintenant pour animer la queue on modifie la valeur de cet angle3 : Si l'angle est supérieur à 40° on le met à -40° (cela nous sert à éviter que la queue fasse un tour complet) sinon on lui ajoute 10°.

case 'z' et 'Z' : Lorsque la touche z est enfoncée, cela va nous permettre de dézoomer la scène. Pour cela lors de la déclaration de notre projection orthographique on avait défini une variable zoom :

glOrtho(-zoom,zoom,-zoom,zoom,-zoom,zoom)

Ainsi pour dézoomer on incrémente zoom. On fixe une valeur d'arrêt à 20 pour éviter de pouvoir dézoomer à l'infini. Lorsque la touche Z est enfoncée, cela va nous permettre cette fois-ci de zoomer sur la scène, pour cela on décrémente zoom et on fixe une valeur d'arrêt à 5 pour empêcher de zoomer à l'infini.

case 'a' et 'A' : Lorsque la touche a est tapée, cela va activer l'éclairage avec *glEnable(GL_LIGHTING)* puis activer la première lumière 1 avec *glEnable(GL_LIGHT1).* Lorsque la touche A est tapée, on désactive la lumière1.

case 'b' et 'B' : Lorsque la touche b est tapée, cela va activer l'éclairage avec *glEnable(GL_LIGHTING)* puis activer la lumière 2 avec *glEnable(GL_LIGHT2)*. Lorsque la touche B est tapée, on désactive la lumière 2.

case 'c' et 'C' : Lorsque la touche c est tapée, cela va activer l'éclairage avec *glEnable(GL_LIGHTING)* puis activer la lumière 0 avec *glEnable(GL_LIGHT0)*. Comme nous n'avons pas redéfini la lumière 0, elle correspond à l'éclairage par défaut de OpenGL. Lorsque la touche C est tapée, on désactive la lumière 0.

case 'e': Lorsque la touche e est tapée, on désactive l'éclairage avec *glDisable(GL_LIGHTING)* .

A la fin de chaque cas on oublie pas de mettre *glutPostRedisplay()* qui permet de mettre à jour la scène après nos modifications puis un *break* pour indiquer que le cas est fini.

- Partie 6 : Fonction SpecialClavier

void SpecialClavier(int touche, int x, int y) Cette fonction prends en paramètre un entier correspondant à la touche spéciale qui vient d'être tapée et x et y les coordonnées de la souris au moment où la touche est tapée. On fait un *switch(touche)* pour programmer plusieurs actions en fonction des touches spéciales.

case GLUT_KEY_UP : GLUT_KEY_UP est une constante prédéfinie dans OpenGL qui correspond à la valeur de la touche flèche vers le haut. Lorsque cette touche est enfoncée, la caméra tourne autour

du personnage par le bas, ainsi on décrémente de 10 la valeur de l'anglex. Cet anglex intervient dans la rotation de notre caméra définie plus haut *glRotatef(anglex,1.0,0.0,0.0)* .

case GLUT_KEY_UP : GLUT_KEY_DOWN est une constante prédéfinie dans OpenGL qui correspond à la valeur de la touche flèche vers le bas. Lorsque cette touche est enfoncée, la caméra tourne autour du personnage par le haut, ainsi on incrémente de 10 la valeur de l'anglex. Cet anglex intervient dans la rotation de notre caméra définie plus haut *glRotatef(anglex,1.0,0.0,0.0)* .

case GLUT_KEY_LEFT : GLUT_KEY_LEFT est une constante prédéfinie dans OpenGL qui correspond à la valeur de la touche flèche vers la gauche. Lorsque cette touche est enfoncée, la caméra tourne autour du personnage par la droite, ainsi on décrémente de 10 la valeur de l'angley. Cet angley intervient dans la rotation de notre caméra définie plus haut *glRotatef(angley,0.0,1.0,0.0)* .

case GLUT_KEY_RIGHT : GLUT_KEY_RIGHT est une constante prédéfinie dans OpenGL qui correspond à la valeur de la touche flèche vers la droite. Lorsque cette touche est enfoncée, la caméra tourne autour du personnage par la droite, ainsi on incrémente de 10 la valeur de l'angley. Cet angley intervient dans la rotation de notre caméra définie plus haut *glRotatef(angley,0.0,1.0,0.0)* .

A la fin de chaque cas on oublie pas de mettre *glutPostRedisplay()* qui permet de mettre à jour la scène après nos modifications puis un *break* pour indiquer que le cas est fini.

- Partie 7 : Fonction Timer

  Grâce à cette fonction nous allons pouvoir effectuer une animation en continu tout en contrôlant la vitesse de l'animation (nombre d'image par secondes).

  On commence par réaliser l'animation du chat qui avance. Pour cela on avait placé un *glTranslatef(-distance,0.0,0.0)* qui englobait notre objet chat en entier. Ici on va simplement incrémenter la valeur de distance. Lorsque le chat sort de la fenêtre (distance>15) alors on le fait revenir par l'autre côté de la fenêtre (distance=-15).

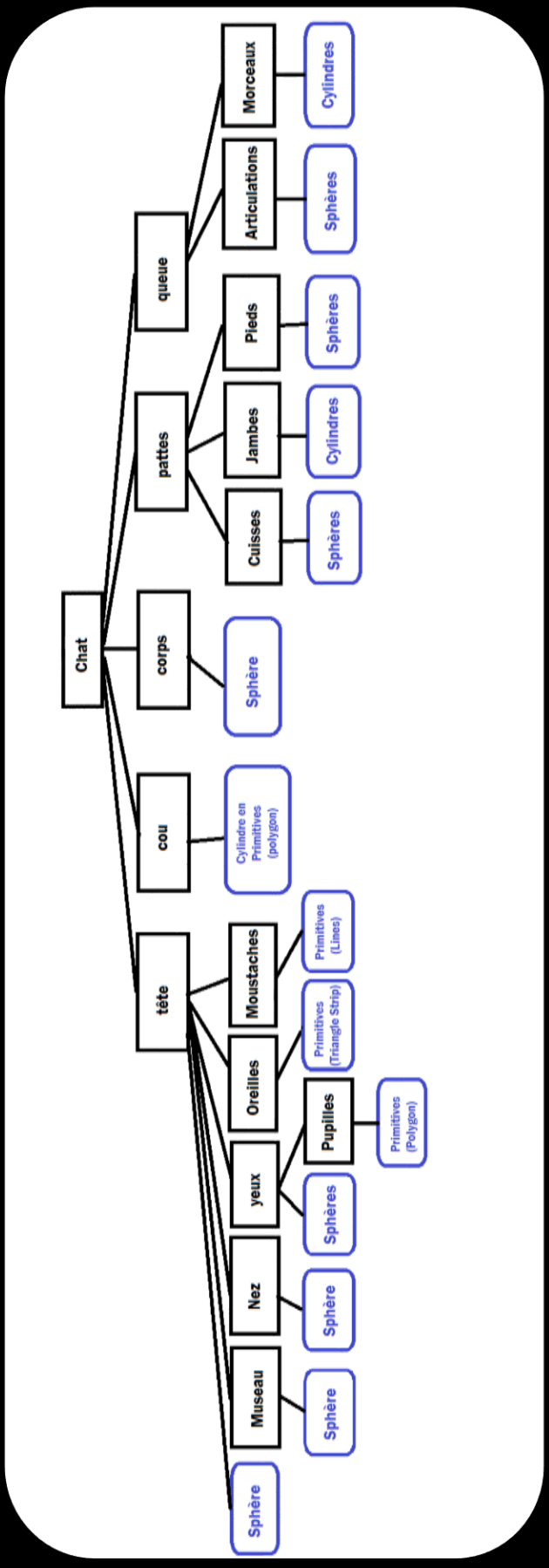
  Ensuite pour que son déplacement soit plus réaliste on va réaliser l'animation des pattes du chat. Pour cela on a défini un *glRotated(angle2,0.0,0.0,1.0)* et *un glRotated(angle,0.0,0.0,1.0)* qui englobent uniquement la jambe et le pied de la patte. La patte avant droite et la patte arrière droite dépendent de angle et la patte avant gauche et la patte arrière gauche dépendent de angle2. Pour que les rotations se déroulent bien on a fait à chaque fois un déplacement à l'origine puis un retour à la place initiale. Ainsi dans notre fonction timer on incrémente angle et on décrémente angle2 pour que cela donne l'aspect que les pattes bougent en décalé. Puis on fixe une valeur d'arrêt à 30° pour éviter que les pattes ne fassent un tour complet. Pour finir on fait un appel récursif à notre fonction timer pour que l'animation se fasse en continu : *glutTimerFunc(1000.0/6.0, timer, 0).* Ici la vitesse de l'animation est fixée à 6 images par seconde.

- Partie 8 : Fonction LoadJpegImage

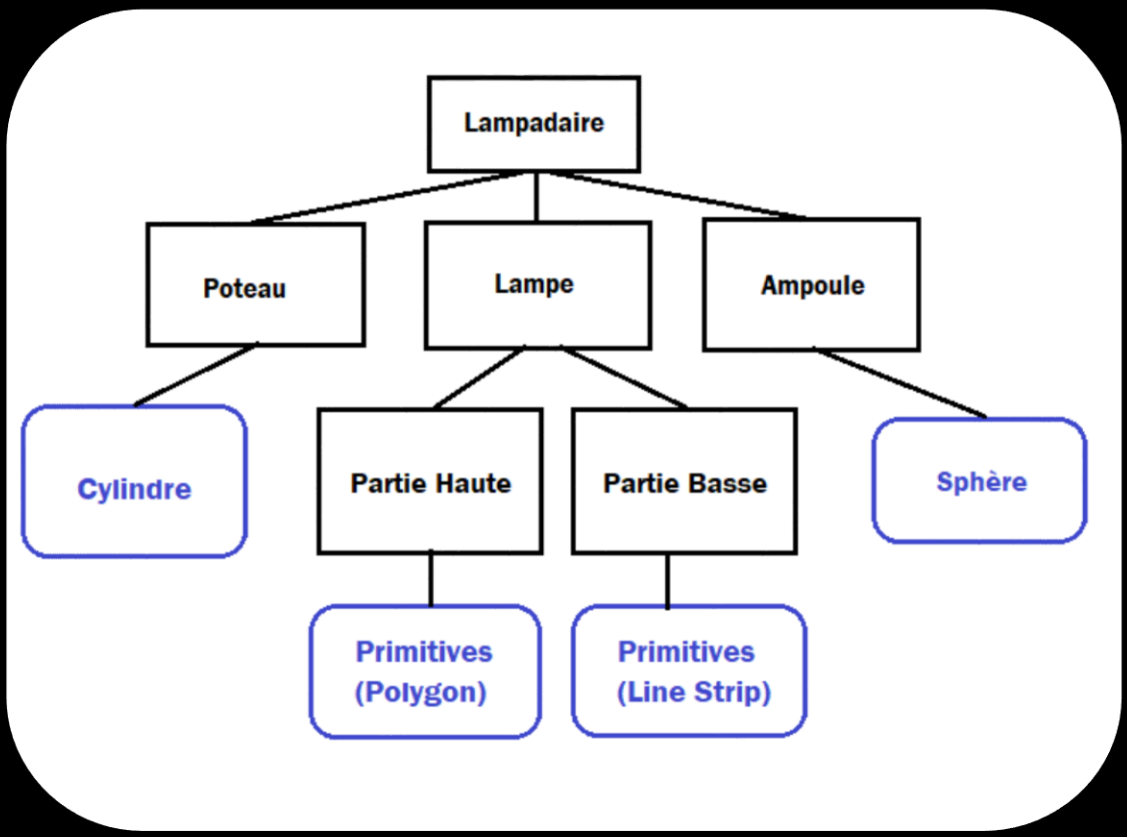
  Nous avons récupéré cette fonction du TP sur les textures. Cette fonction nous permet de charger une image .jpg dans notre programme afin de pouvoir l'utiliser comme texture par la suite. Ici nous ne récupérons que des images à taille fixe de 256*256pixels.

Annexe

- Schéma du chat :



- Schéma du lampadaire :



- Schéma du sol :