Fondamenti di Computer Graphics LM

Lab 2 – Navigazione interattiva in scena con modelli geometrici 3D

Mattia Fucili

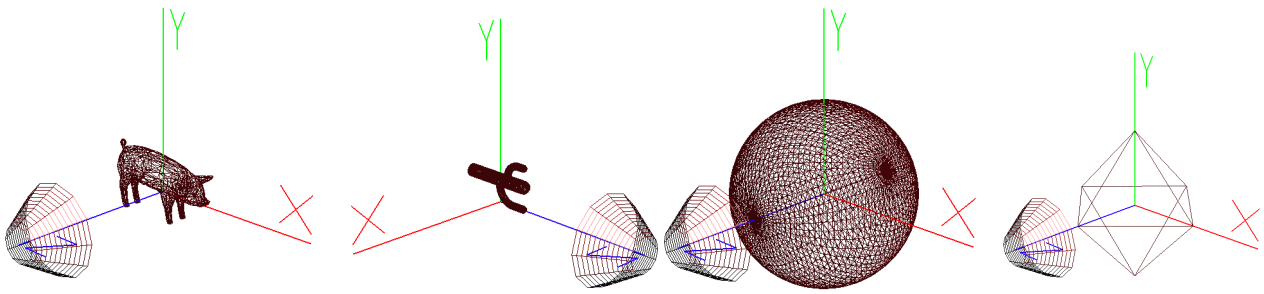
In questa esercitazione siamo passati alla grafica 3D sempre lavorando con OpenGL con l'obiettivo di realizzare degli algoritmi che permettessero di caricare mesh e visualizzarle in varie modalità ed infine di traslarle e ruotarle.

Obiettivo 1

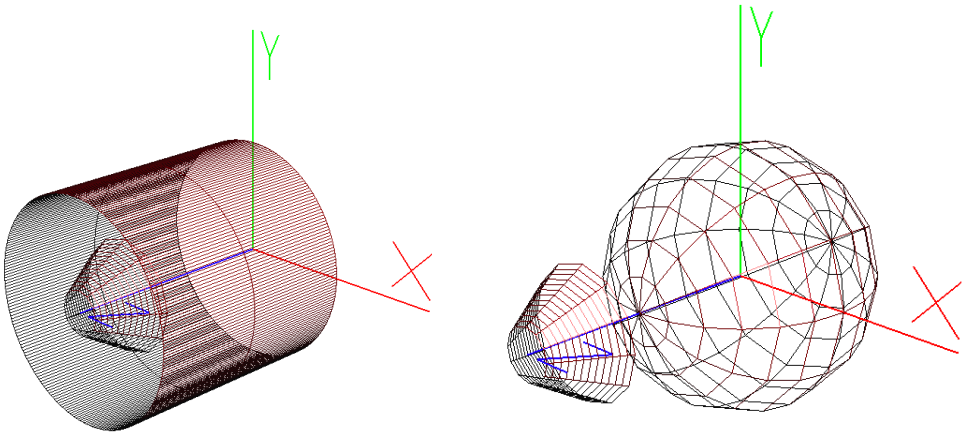
Come primo obiettivo avevamo quello di caricare mesh e poligoni e modificare le modalità di visualizzazione.
Per caricare le mesh bisognava leggere da un file con estensione ".m" i vertici e le facce che componevano l'oggetto. Per disegnarle invece occorreva eseguire questi comandi:

```
printf("Creazione display list .. \n");
	listname = glGenLists(1);
	glNewList(listname,GL_COMPILE);

	for(i = 0; i < nface; i++) {
		ids[2] = faces[i][0];
		ids[1] = faces[i][1];
		ids[0] = faces[i][2];

		glBegin(GL_TRIANGLES);
		glColor3f(1,0,0);
		for(ii = 2; ii >= 0; ii--) {
			glVertex3f(vertices[ids[ii]][0], vertices[ids[ii]][1],
vertices[ids[ii]][2]);
		}
		glEnd();
	}

	glEndList();
```

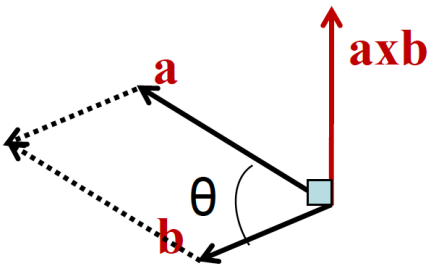
Nello specifico il ciclo che va a disegnare con glVertex3f() viene eseguito al contrario perché in base all'ordine con cui vengono disegnati i vertici di una faccia la sua normale è diretta verso l'esterno dell'oggetto o meno. Se non si facesse in questo modo la normale sarebbe diretta verso l'interno e quando viene mostrata la mesh non risulterebbe illuminata, ma semplicemente tutta nera. Il risultato dei caricamenti di alcuni oggetti sono i seguenti:



Inoltre è possibile disegnare delle superfici quadriche presenti già nella libreria OpenGL come ad esempio cilindri, sfere e tori.



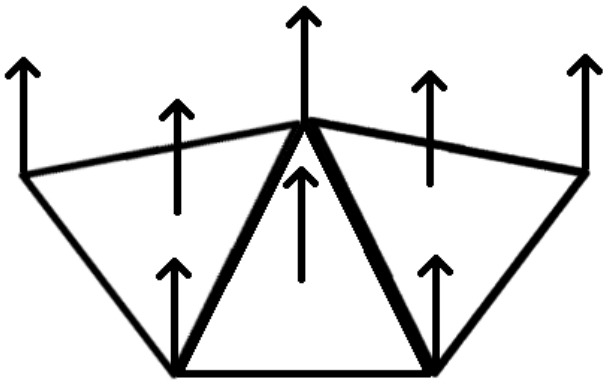
Per poter vedere gli oggetti illuminati con anche le relative ombre è stato necessario calcolare le normali ai vertici ed alle facce. Questo è necessario perché per vedere se una parte dell'oggetto deve essere illuminata o meno dalla luce occorre calcolare l'angolo tra il vettore normale alla faccia e il vettore del raggio di luce che lo colpisce. Se questo angolo è $\geq 90°$ allora la faccia si trova in ombra, altrimenti incide sul colore della faccia tanto più, quanto è piccolo l'angolo. Per calcolare le normali si è calcolato il cross product.



Il calcolo è stato fatto utilizzando le librerie "v3d.c" fornite con il progetto ed il codice è il seguente:

```
faces[*nFaces][0] = (int)a - 1;
faces[*nFaces][1] = (int)b - 1;
faces[*nFaces][2] = (int)c - 1;

float vA[3], vB[3], vAB[3];
v3dSub(vertices[(int) b - 1], vertices[(int)a - 1], vA);
v3dSub(vertices[(int) c - 1], vertices[(int)a - 1], vB);

v3dCross(vA, vB, vAB);

v3dNormalize(vAB);

fnormals[*nFaces][0] = vAB[0];
fnormals[*nFaces][1] = vAB[1];
fnormals[(*nFaces)++][2] = vAB[2];
```

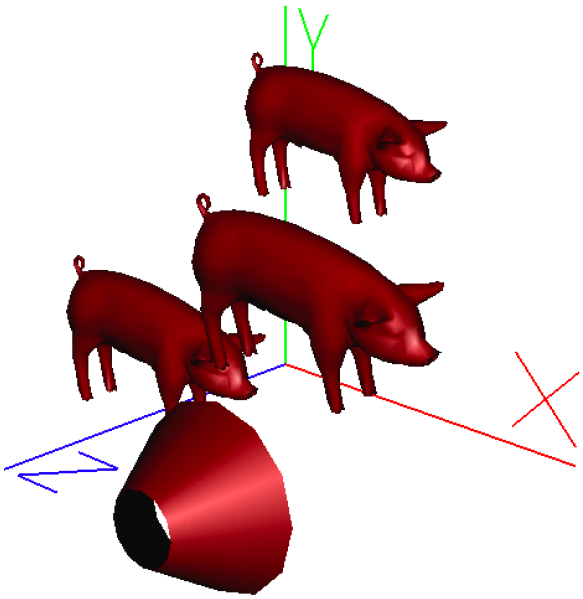
Inoltre per visualizzare la superficie dell'oggetto smussata è anche necessario calcolare le normali di tutti i vertici. Essendo che un vertice compare in più facce per calcolare la sua normale è necessario considerare i contributi di tutte le facce e sommarli insieme:



Per abilitare la visione smussata è necessario aggiungere questo comando:

```
if(shading)
   glShadeModel(GL_SMOOTH);
else
   glShadeModel(GL_FLAT);
```

Il risultato, considerando anche che è stata tolta la modalità wireframe, è il seguente:



Obiettivo 2

Come secondo obiettivo ci era stato chiesto di implementare alcune funzionalità per cambiare il modo di visualizzare gli oggetti.

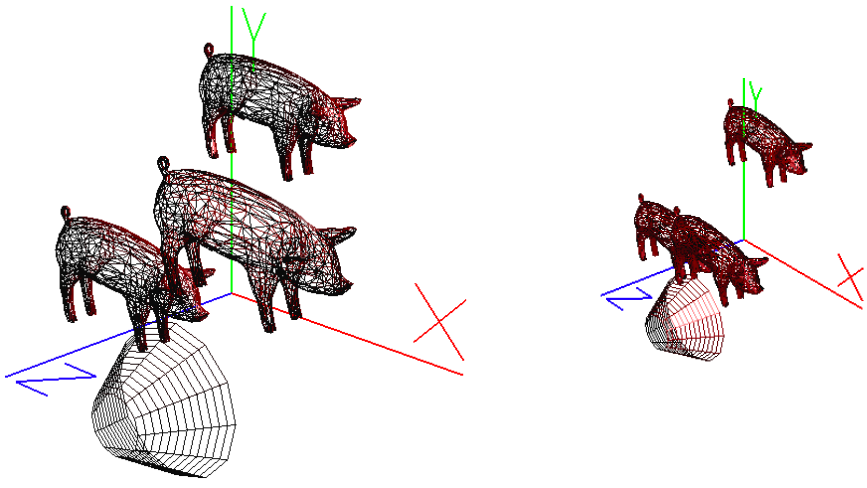
1) Change Eye Point

Questa funzionalità permette di cambiare il punto di vista della camera, cioè tutta la visuale della scena che si ha davanti, in questo caso. Per fare ciò si è sfruttata la relativa variabile globale `MODE_CHANGE_EYE_POS` che viene settata ogni volta che l'utente clicca sull'apposita opzione del menu a comparsa. Per rendere effettivo il cambiamento si sono introdotti dei comandi da tastiera: x, y, z per avvicinare la telecamera nei rispettivi assi e X, Y, Z per allontanarsi. Il tutto è implementato nella callback della keyboard dove viene aggiunto questo pezzo di codice:

```
if(mode == MODE_CHANGE_EYE_POS) {
   pos = camE;
   step = 0.1;
}
```

che serve per settare la variabile che deve essere modificata. Infine si sfrutta una parte di codice già esistente per cambiare il punto di vista in base alla lettera che viene cliccata:

```
if(pos != NULL) {
        if(key == 'x')
                pos[0] += step;
        else if(key == 'X')
                pos[0] -= step;
        else if(key == 'y')
                pos[1] += step;
        else if(key == 'Y')
                pos[1] -= step;
        else if(key == 'z')
                pos[2] += step;
        else if(key == 'Z')
                pos[2] -= step;

        glutPostRedisplay();
}
```

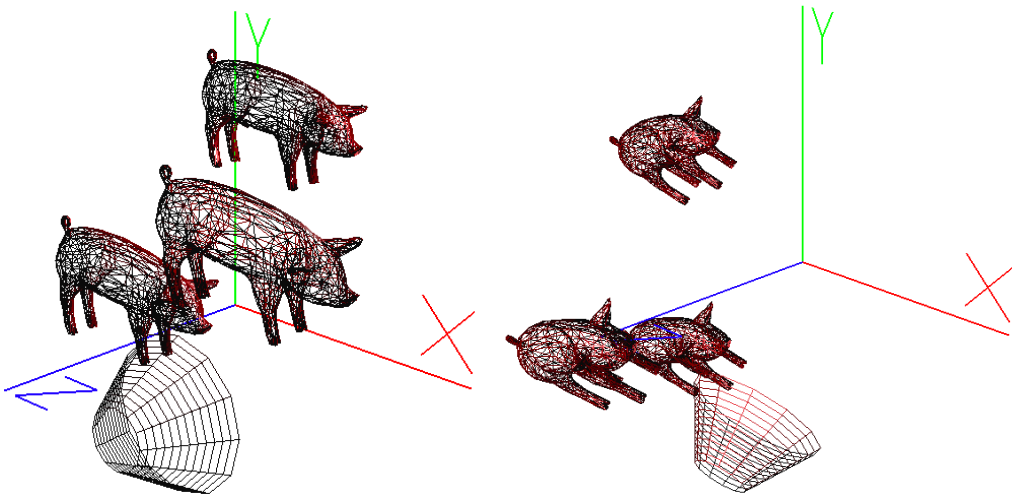


2) Rotate model

Per implementare questa funzionalità si è sfruttata l'opportuna variabile globale `MODE_ROTATE_OBJECT` che viene settata quando viene abilitata la relativa funzione di rotazione degli oggetti in scena dal menù. Anche qui con i tasti x, y, z si aumenta l'angolo relativo ai rispettivi assi e con X, Y, Z si diminuisce. Per realizzare effettivamente la rotazione si è sfruttato anche in questo caso la callback della keyboard in questo modo:

```
} else if(mode == MODE_ROTATE_OBJECT) {
    pos = angle;
    step = 1.0;
}
```

Il risultato è il seguente

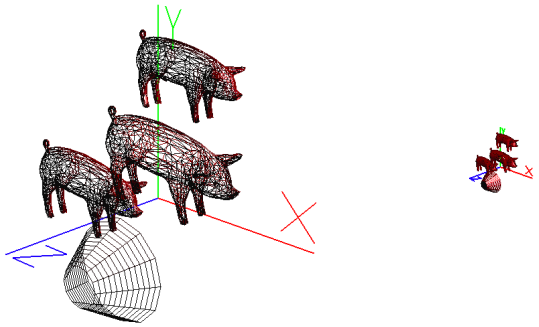


3) Zoom in/out

Per simulare lo zoom si è andato a collegare la variabile che descrive il field of view con la relativa opzione del menù a comparsa. Prima di tutto si è associata la variabile `MODE_CHANGE_ZOOM` dopo di ché si è modificata la callback della keyboard nel modo seguente:

```
} else if(mode == MODE_CHANGE_ZOOM) {
    pos = &fovy;
    step = 1.0;
}
...
if( pos != NULL ) {
...
   } else if(key == 'f')
          *pos += step;
     else if(key == 'F')
          *pos -= step;
   glutPostRedisplay();

}
```

Il risultato è:

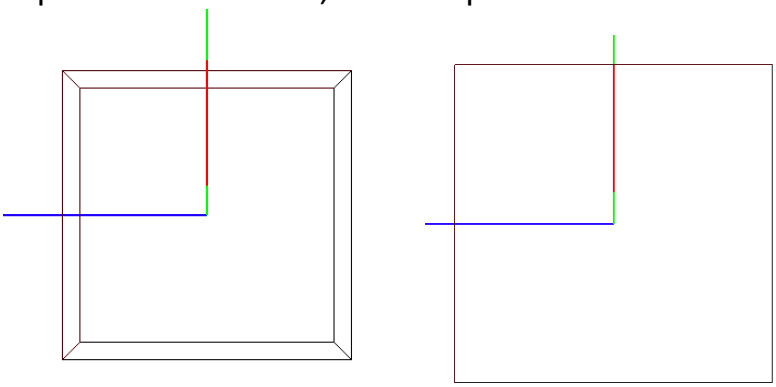


4) Proiezione

Per cambiare il tipo di proiezione si è sfruttata la variabile globale `MODE_CHANGE_PROJECTION` associata alla selezione dal menù che permette attraverso i seguenti comandi

```
if(sel == MODE_CHANGE_PROJECTION) {
    orpro = !orpro;
}

if(orpro)
    gluPerspective(fovy, aspect, 1, 100);
else
    glOrtho(-1.0, 1.0, -1.0, 1.0, -10, 100);
```

di passare da una proiezione all'altra, ad esempio con un cubo



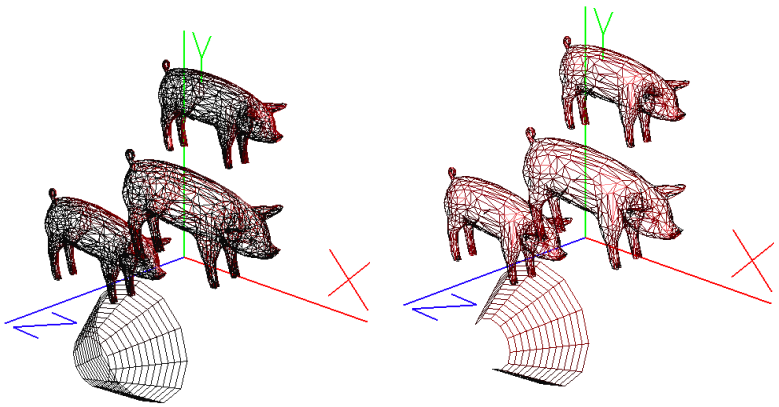
5) Culling

Il culling è una funzionalità che permette di eliminare o meno dalla visuale le facce che si trovano dietro la parte visibile dell'oggetto. Per fare ciò si è modificato il codice nel seguente modo

```
if(cull) {
    glEnable(GL_CULL_FACE);
    glCullFace(GL_BACK);
} else {
    glDisable(GL_CULL_FACE);
}
```

che in base a se il culling è attivo o meno abilita il taglio delle facce dietro con la costante `GL_BACK`. La variabile `cull` viene settata al momento della selezione dal menù.

Il risultato è:

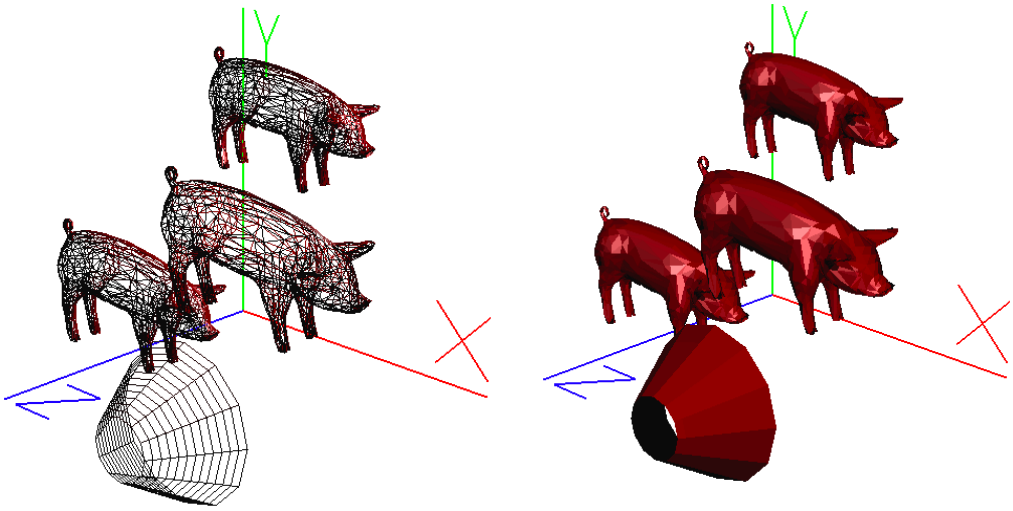


6) Wireframe

Abilitando o meno questa funzione è possibile vedere gli oggetti in versione “fil di ferro” e quindi solo vertici e spigoli oppure, se disabilitata, è possibile vederli “pieni” cioè vengono colorati i singoli triangoli che li compongono. È possibile scegliere se abilitare o meno questa funzione dal menù attraverso questa variabile `MODE_CHANGE_WIREFRAME` che a default è abilitata. Il pezzo di codice che gestisce questo cambio è il seguente:

```
if(wireframe)
   glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
else
   glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
```

che proprio grazie alla costante `GL_FILL` permette di riempire i vari triangoli. Come si può vedere qua sotto:



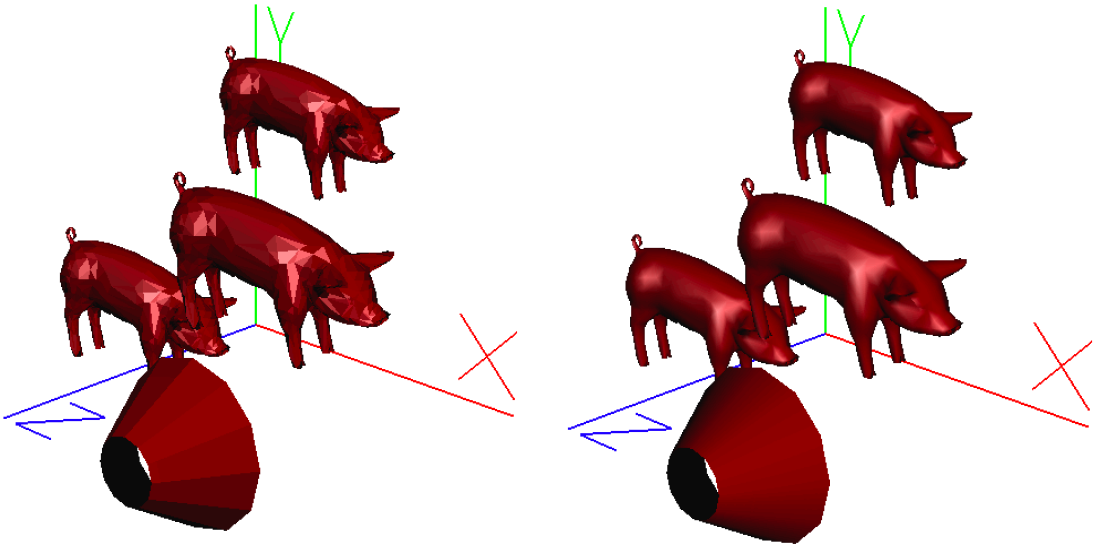
7) Shading

Lo shading permette di simulare una visualizzazione smussata dei triangoli che compongono la mesh. Questa funzione in realtà non smussa realmente l’oggetto ma semplicemente calcola la normale di ogni singolo pixel che compone la mesh e ne applica il colore, così da rendere la superficie più uniforme. Se non abilitato invece per ogni singolo triangolo considera solo la normale di un vertice che lo compone e

applica il colore associato a quel valore a tutta la faccia. L'eventuale selezione di questa funzione dal menù comporta l'esecuzione di questo comando:

```
if(shading)
   glShadeModel(GL_SMOOTH);
else
   glShadeModel(GL_FLAT);
```

Come si può intuire se abilitato "smussa" i triangoli, nell'altro caso no. A default non è abilitato quindi la visuale è "spigolosa".

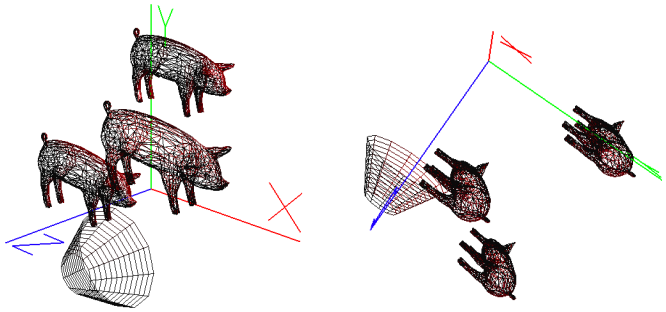


8) Track ball

La track ball permette di ruotare il modello 3D calcolando le coordinate, l'asse di rotazione e l'angolo di rotazione in base a dove si muove il mouse in 2D. Per ruotare il modello basta semplicemente cliccare con il sinistro e tenere premuto muovendosi. All'inizio ogni volta che si lasciava il mouse e si ripeteva la rotazione il modello ripartiva dallo stato iniziale, ma modificando il codice e mettendo queste righe

```
tbV[0] = tbW[0];
tbV[1] = tbW[1];
tbV[2] = tbW[2];
```

si va a salvare ogni volta lo stato attuale così da permettere ogni volta di ruotare la reale visuale che si ha in quel momento.
Esempio di rotazione del modello tramite track ball è il seguente:

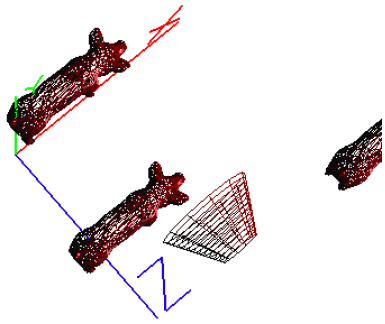


9) Camera motion

Questa funzione permette alla camera di muoversi lungo una curva. Per rendere effettiva questa funzione è stata inserita una idleFunction che viene abilitata alla pressione del tasto “s”. La funzione, riportata qua sotto:

```
void idle() {
    float t;
    float result[3];

    if(flagMotion) {
        t = (GLfloat) indexMotion / 100;
        deCasteljau(t, &result[0]);
        camE[0] = result[0];
        camE[1] = result[1];
        camE[2] = result[2];

        indexMotion++;
        if(indexMotion == 100)
            indexMotion = 0;

        glutPostRedisplay();
    }
}
```

come si può vedere sfrutta l’algoritmo di De Casteljau (esercitazione 1) per calcolare tutti i punti interpolanti dei control point. Questi control point sono definiti a priori in maniera tale da far percorrere alla camera un cerchio dall’alto.



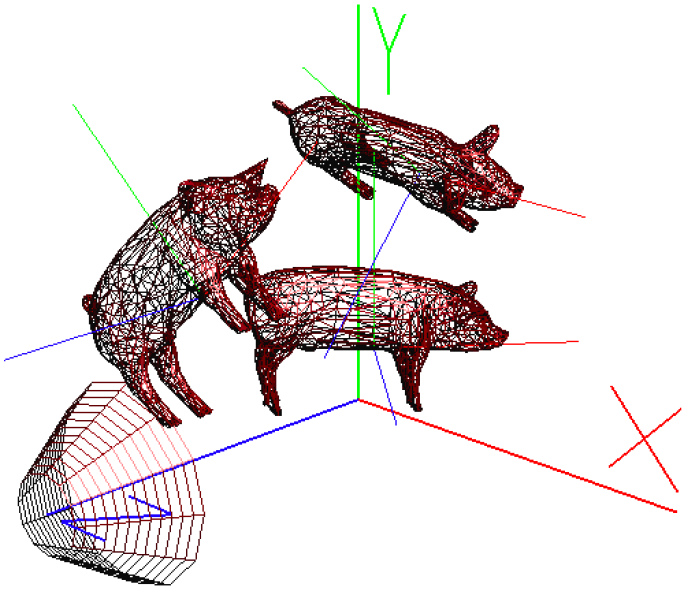
Obiettivo 3

Nella terza parte dell’esercitazione ci è stato chiesto di manipolare gli oggetti in scena mediante le matrici all’interno dello stack.

1) Posizionare più mesh nella scena

Per realizzare questo punto è stato necessario modificare la modalità di caricamento degli oggetti. Infatti è stato creato un metodo, `loadMesh(...)`, richiamato tante volte quante sono le mesh che si vogliono caricare che racchiude tutta la logica di caricamento dei vertici e calcolo delle normali discusso nel primo punto della

relazione. Dopo aver applicato eventuali traslazioni e rotazioni per evitare che le varie mesh vengano visualizzate sovrapposte il risultato è il seguente:



2) Traslazione e rotazione rispetto ai sistemi WCS ed OCS

Per sviluppare questo punto è necessario ricordare che la moltiplicazione fra matrici non gode della proprietà commutativa, pertanto l'applicazione delle varie trasformazioni deve seguire un preciso ordine.

Per prima cosa è stato necessario rendere il sistema sensibile alla pressione dei tasti utili ad abilitare le varie funzioni di traslazione/rotazione rispetto ai due sistemi di riferimento. Per fare ciò è stata modificata la callback della keyboard nel seguente modo:

```
if(key == 'w')
    mode = MODE_TRANSLATE_WCS;
else if(key == 'W')
    mode = MODE_ROTATE_WCS;
else if(key == 'o')
    mode = MODE_TRANSLATE_OCS;
else if(key == 'O')
    mode = MODE_ROTATE_OCS;
else if(key == '1')
    selectedMesh = 0;
else if(key == '2')
    selectedMesh = 1;
else if(key == '3')
    selectedMesh = 2;
```

Infine ogni possibile modifica apportabile dall'utente viene gestita nel seguente modo

```
if(mode == MODE_ROTATE_WCS) {
    glPushMatrix();
    glLoadIdentity();
    step = 5.0;
    if(key == 'x')
        glRotatef(step, 1, 0, 0);
    else if(key == 'X')
        glRotatef(-step, 1, 0, 0);
    else if(key == 'y')
        glRotatef(step, 0, 1, 0);
    else if(key == 'Y')
        glRotatef(-step, 0, 1, 0);
    else if(key == 'z')
        glRotatef(step, 0, 0, 1);
    else if(key == 'Z')
        glRotatef(-step, 0, 0, 1);

    glMultMatrixf(matrixWCS[selectedMesh]);
    glGetFloatv(GL_MODELVIEW, matrixWCS[selectedMesh]);
    glPopMatrix();
}
```

Spiegazione primitive utilizzate:

- `glPushMatrix()`: serve per duplicare la matrice c'è in cima allo stack;
- `glLoadIdentity()`: carica la matrice identità al posto della matrice che si trova in cima allo stack;
- `glRotatef(...)`: moltiplica la matrice attuale per quella di rotazione ottenuta dai parametri passati, in questo caso sarà una matrice di rotazione di 5 gradi nell'asse x, e la sovrascrive;
- `glMultMatrixf(...)`: moltiplica la matrice ottenuta per quella contenente le informazioni dell'oggetto rispetto al WCS;
- `glGetFloatv(...)`: ottiene la matrice dallo stack e la sovrascrive al valore attuale;
- `glPopMatrix()`: infine viene rimossa la matrice attuale (che è stata salvata) e viene ripristinata la matrice che c'era all'inizio delle modifiche.

Una volta calcolata la matrice da applicare alla mesh è necessario modificare la display in questo modo:

```
for(int i = 0; i < N_MESH; i++) {
    glPushMatrix();
    glMultMatrixf(matrixWCS[i]);
    glMultMatrixf(matrix[i]);
    drawAxis(1.0, 0);
    glMultMatrixf(matrixOCS[i]);
    glCallList(listname[i]);
    glPopMatrix();
}
```

l'ordine di applicazione delle moltiplicazioni non è casuale infatti per prima cosa occorre applicare eventuali traslazioni e rotazioni rispetto al WCS all'oggetto, poi occorre moltiplicare per la posizione iniziale dell'oggetto ed infine moltiplicare

eventuali trasformazioni effettuate in base al sistema di riferimento dell'oggetto prima di disegnarlo tramite la `glCallList(...)`.