

Chương 2: Vẽ các đối tượng hình học cơ bản trong OpenGL

1. Vẽ điểm, đường, đa giác (Point, Line, Polygon)

1.1 OpenGL tạo ra điểm, đường, đa giác từ các đỉnh (Vertex). Các đỉnh được liệt kê giữa hai hàm

`glBegin(tham số)`

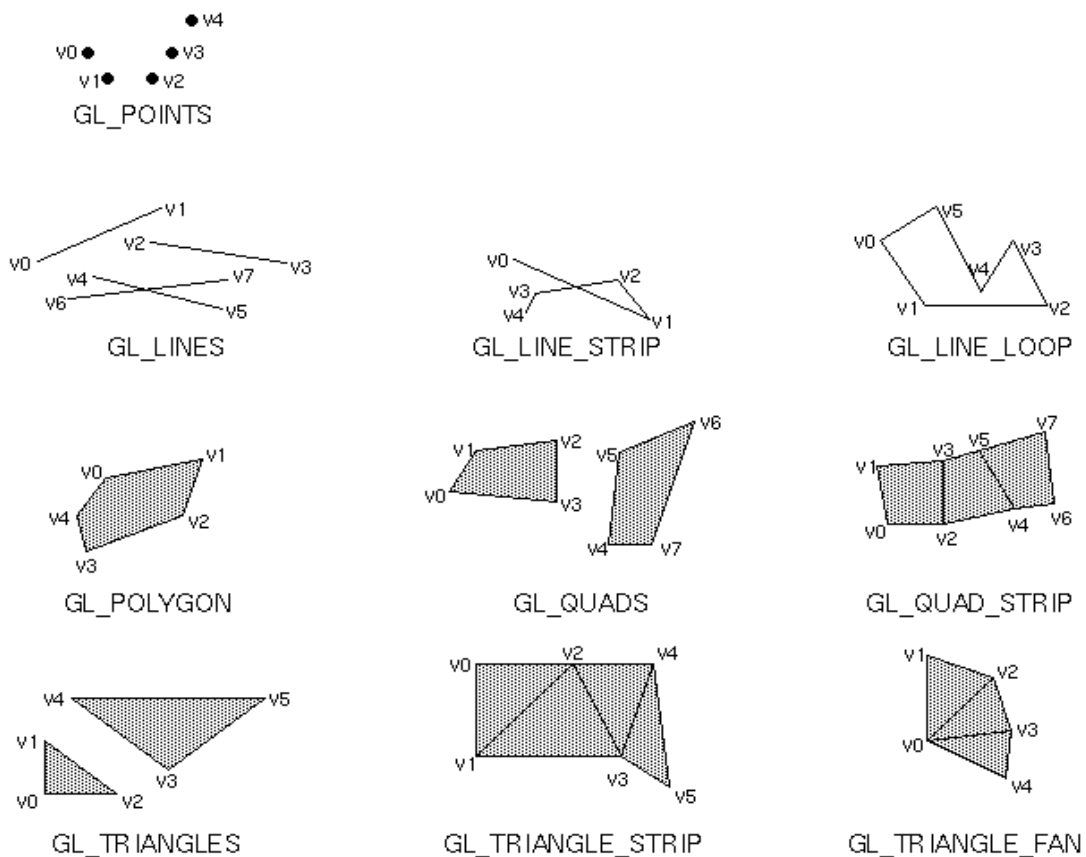
.....
`glEnd();`

Danh sách các *tham số* được liệt kê trong bảng 2.1

GL_POINTS	Các điểm
GL_LINES	Đoạn thẳng
GL_POLYGON	Đa giác lồi
GL_TRIANGLES	Tam giác
GL_QUADS	Tứ giác
GL_LINE_STRIP	Đường gấp khúc không khép kín
GL_LINE_LOOP	Đường gấp khúc khép kín
GL_TRIANGLE_STRIP	Một dải các tam giác liên kết với nhau
GL_TRIANGLE_FAN	Các tam giác liên kết theo hình quạt
GL_QUAD_STRIP	Một dải các tứ giác liên kết với nhau

Bảng 2.1 Các tham số của `glBegin()`

Hình phía dưới minh họa kết quả hiển thị khi ta thay đổi các tham số khác nhau



Hình 2.1 Các đối tượng hình học cơ bản

Để chỉ định một điểm ta dùng lệnh sau

glVertex{234}{sifd}[v](Tọa độ điểm);

Trong đó:

- {234} chỉ định số chiều của không gian
- {sifd} chỉ định kiểu dữ liệu của tọa độ, ý nghĩa được chỉ định trong bảng 2.2

Kí hiệu	Kiểu dữ liệu	Tên kiểu của OpenGL
s	16-bit integer	GLshort
i	32-bit integer	GLint, GLsizei
f	32-bit floating-point	GLfloat, GLclampf
d	64-bit floating-point	GLdouble, GLclampd

Bảng 2.2 Một số kiểu dữ liệu của OpenGL

- [v] nếu tọa độ điểm được truyền từ 1 mảng cho trước

Ví dụ 2.1: Sau đây là một số ví dụ khi ta chỉ định 1 điểm

`glVertex2s(2, 3); //Tọa độ thuộc kiểu GLshort trong không gian 2 chiều`

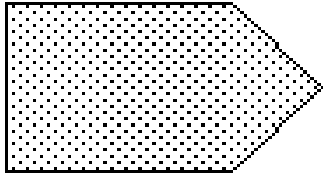
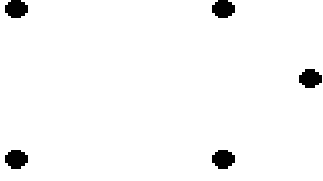
`glVertex3d(0.0, 0.0, 3.1415926535898); //Tọa độ thuộc kiểu GLdouble trong không gian 3 chiều`

`glVertex4f(2.3, 1.0, -2.2, 2.0); //Tọa độ đồng nhất kiểu GLfloat`

`GLdouble dvect[3] = {5.0, 9.0, 1992.0};`

`glVertex3dv(dvect); //Tọa độ được lấy từ mảng dvect`

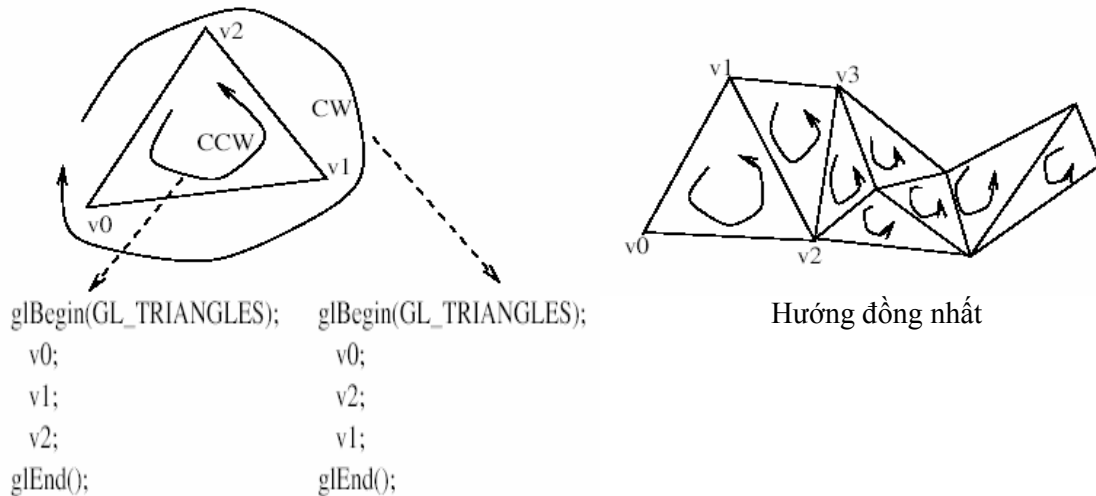
Ví dụ 2.2:

Lệnh	Kết quả hiển thị
<pre>glBegin(GL_POLYGON); glVertex2f(0.0, 0.0); glVertex2f(0.0, 3.0); glVertex2f(3.0, 3.0); glVertex2f(4.0, 1.5); glVertex2f(3.0, 0.0); glEnd();</pre>	
<pre>glBegin(GL_POINTS); glVertex2f(0.0, 0.0); glVertex2f(0.0, 3.0); glVertex2f(3.0, 3.0); glVertex2f(4.0, 1.5); glVertex2f(3.0, 0.0); glEnd();</pre>	

1.2 Hướng của tam giác (Orientation)

Hướng của một tam giác là hướng các đỉnh của nó, hướng có thể là cùng chiều kim đồng hồ (clockwise-CW) hoặc ngược chiều kim đồng hồ (counter-clockwise-CCW). Tập hợp các tam giác trên một mặt phẳng được gọi là hướng đồng nhất (consistently oriented) nếu hướng của các tam giác là như nhau.

Hướng của tam giác được xác định dựa vào thứ tự các điểm được chỉ định giữa glBegin và glEnd



Hình 2.2 Hướng của tam giác

Chú ý quan trọng: Khi ta vẽ các tam giác liên tiếp trong OpenGL, chúng ta phải đảm bảo chắc chắn rằng hướng của các tam giác đó là đồng nhất.

1.3 Một số lệnh khác

Thiết lập màu nền cho cửa sổ hiển thị

glClearColor(red, green, blue, anpha); ($0 \leq \text{red, green, blue, anpha} \leq 1$)

Thiết lập màu cho đối tượng vẽ

glColor3f(red, green, blue); ($0 \leq \text{red, green, blue} \leq 1$)

Vẽ hình chữ nhật

glRect{sifd}(x1, y1, x2, y2);

Một số thủ tục trong GLUT cho phép vẽ hình trong không gian 3 chiều (solid là hình đặc, wire là hình khung)

Vẽ hình lập phương

glutWireCube(GLdouble size);

glutSolidCube(GLdouble size);

Vẽ hình cầu

glutWireSphere(GLdouble radius, GLdouble slices, GLdouble stacks);

glutSolidSphere(GLdouble radius, GLdouble slices, GLdouble stacks);

Vẽ hình đế hoa

glutWireTorus(GLdouble innerRadius, GLdouble outerRadius,
GLdouble nsides, GLdouble rings);

glutSolidTorus(GLdouble innerRadius, GLdouble outerRadius,
GLdouble nsides, GLdouble rings);

Vẽ hình ấm pha trà

glutWireTeapot(GLdouble size);

glutSolidTeapot(GLdouble size);

Vẽ hình nón

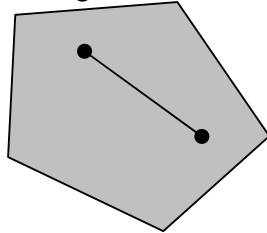
glutWireCone(GLdouble *Radius*, GLdouble *height*, GLint *slices*, GLint *stacks*);

glutSolidCone(GLdouble *Radius*, GLdouble *height*, GLint *slices*, GLint *stacks*);

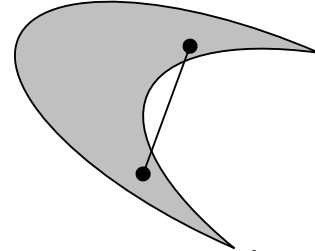
2. Tập lồi và phép đặc tam giác hợp lệ (Convexity and Valid Triangulation)

2.1 Tập lồi

Một tập S được gọi là lồi (convex) nếu với 2 điểm bất kì P, Q thuộc S thì đoạn thẳng PQ nằm trọn vẹn trong S



Tập lồi



Không phải tập lồi

Hình 2.3 Tập lồi và tập không lồi

Cho một tập hợp các điểm $T = \{P_1, P_2, \dots, P_n\}$, bao lồi (convex hull) của T là tập lồi nhỏ nhất F chứa T . Khi đó bất kì điểm X thuộc F đều có thể biểu diễn dưới dạng như sau:

$$X = \alpha_1 P_1 + \alpha_2 P_2 + \dots + \alpha_n P_n \quad (0 \leq \alpha_i, \alpha_1 + \alpha_2 + \dots + \alpha_n = 1)$$

Trường hợp đặc biệt:

- Một đoạn thẳng AB luôn là một tập lồi, một điểm P bất kỳ thuộc AB được biểu diễn duy nhất dưới dạng $P = \alpha_1 A + \alpha_2 B$ ($\alpha_i \geq 0, \alpha_1 + \alpha_2 = 1$)

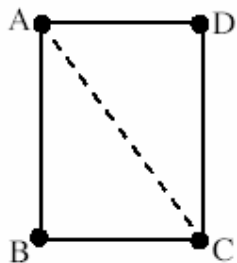
- Một tam giác ABC luôn là một tập lồi, một điểm P bất kỳ thuộc ABC được biểu diễn duy nhất dưới dạng $P = \alpha_1 A + \alpha_2 B + \alpha_3 C$ ($\alpha_i \geq 0, \alpha_1 + \alpha_2 + \alpha_3 = 1$)

OpenGL tô màu đoạn thẳng và tam giác bằng cách nội suy véc tơ màu (interpolating the color vectors) tại mỗi đỉnh của đoạn thẳng và tam giác. Ví dụ nếu màu tại ba đỉnh A, B, C của tam giác lần lượt là C_1, C_2, C_3 khi đó nếu điểm P thuộc tam giác ABC và $P = \alpha_1 A + \alpha_2 B + \alpha_3 C$ thì màu tại điểm P là $\alpha_1 C_1 + \alpha_2 C_2 + \alpha_3 C_3$

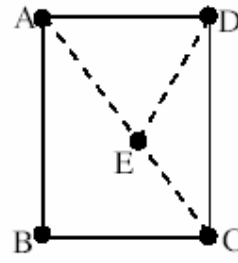
2.2 Phép đặc tam giác hợp lệ

Một phép đặc tam giác hợp lệ của một hình X là một tập hợp các tam giác thỏa mãn hai điều kiện sau:

- Hợp của các tam giác chính là hình X
- Hai tam giác bất kì thỏa mãn: Tách rời nhau hoặc chung nhau duy nhất 1 đỉnh hoặc chung nhau duy nhất một cạnh



Phép đặc tam giác
(a)



Không phải phép đặc tam giác
(b)

Hình 2.4 Phép đặc tam giác hợp lệ và không hợp lệ

Trong OpenGL khi áp dụng một phép đặc tam giác cho một tập lồi thì yêu cầu bắt buộc đó phải là một phép đặc tam giác hợp lệ. Tại sao phải có yêu cầu như vậy? Ta hãy xét ví dụ tại hình 2.3(b), giả sử E là trung điểm cạnh AC và các véc tơ biểu diễn màu tại các điểm A, E

và C lần lượt là (1,0,0), (0,1,0) và (0,0,1). Xét trong tam giác ABC, vì E là trung điểm cạnh AC do đó màu tại E là trung bình cộng màu tại A và C do đó màu tại E được tính là (0.5,0,0.5) điều đó có nghĩa là cùng tại một điểm E có hai cách biểu diễn màu khác nhau.

3 Tổng quan quá trình biểu diễn ảnh trong đồ họa máy tính

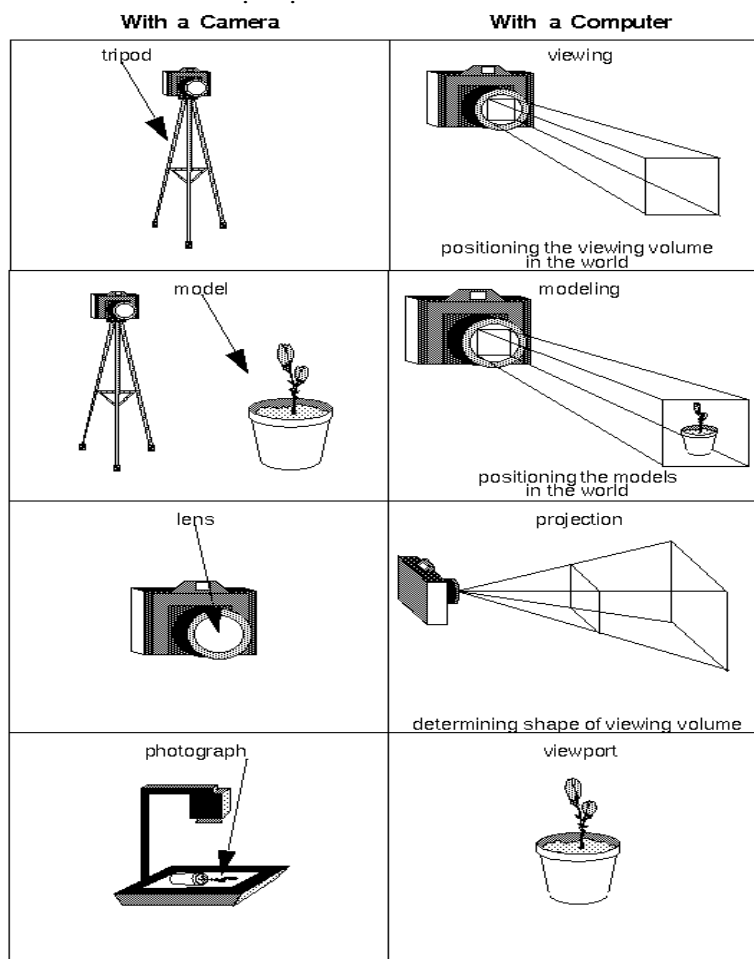
Mục đích của đồ họa máy tính đó là tạo ra hình ảnh 2 chiều (2D) trên màn hình máy tính của các vật thể trong không gian 3 chiều (3D). Quá trình chuyển đổi tọa độ từ không gian 3D thành các điểm ảnh (pixel) xuất hiện trên màn hình được mô tả như sau:

- **Bước 1:** Áp dụng các phép biến đổi được biểu diễn bởi các phép nhân ma trận (matrix multiplication) bao gồm phép biến đổi mô hình (modeling), phép biến đổi điểm nhìn (viewing) và phép chiếu (projection). Nhìn chung ta kết hợp một số phép biến đổi khi đó đối tượng cần vẽ trong không gian 3D được cắt theo một không gian thực được gọi là viewing volume. Viewing volume xác định cách thức mà vật thể được chiếu lên màn hình là phép chiếu phối cảnh (perspective projection) hay phép chiếu trực giao (orthographic projection) và phần nào của đối tượng được hiển thị.

- **Bước 2:** Viewing volume được chiếu lên một mặt phẳng chiếu hình chữ nhật được gọi là khung nhìn (rectangular window)

- **Bước 3:** Ánh xạ hình ảnh trong khung nhìn thành các điểm ảnh trên một cửa sổ của màn hình máy tính mà ta gọi đó là cổng nhìn (viewport).

Toàn bộ các bước trên tương tự như khi ta sử dụng máy ảnh (camera) để chụp một bức ảnh trong thực tế. Hình 2.4 minh họa sự so sánh đó



Hình 2.5 So sánh giữa camera và máy tính

4 Phép biến đổi điểm nhìn và biến đổi mô hình (Viewing and Modeling Transformations)

OpenGL sử dụng ma trận và phép nhân ma trận để biểu diễn các phép biến đổi (ta sẽ đề cập đến vấn đề này sau) do đó một chú ý quan trọng là trước khi áp dụng các phép biến đổi ta phải xóa và thiết lập ma trận hiện thời (current matrix) về ma trận đơn vị (identity matrix) bằng lệnh **glLoadIdentity()**.

4.1 Phép biến đổi điểm nhìn

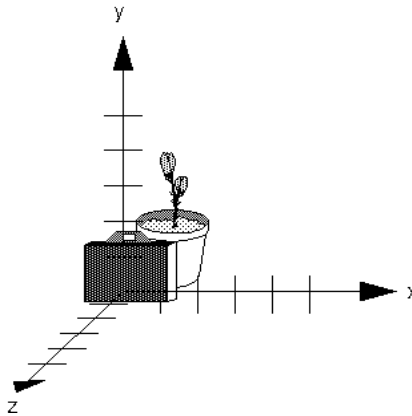
Phép biến đổi điểm nhìn tương tự như ta chọn vị trí và hướng của camera trong thực tế vì vậy ta sẽ quy ước tại vị trí điểm nhìn (viewpoint) có đặt một camera cho dễ hình dung. OpenGL sử dụng lệnh **gluLookAt()** để định vị vị trí và hướng của điểm nhìn, cú pháp lệnh như sau

```
gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez,  
           GLdouble centerx, GLdouble centery, GLdouble centerz,  
           GLdouble upx, GLdouble upy, GLdouble upz)
```

Trong đó: (*eyex*, *eyey*, *eyez*) là vị trí của điểm nhìn, (*centerx*, *centery*, *centerz*) chỉ định một điểm nào đó thuộc đường ngắm và (*upx*, *upy*, *upz*) xác định hướng nhìn của camera (bản chất chính là hướng từ đáy lên đỉnh của viewing volume).

Ví dụ lệnh **gluLookAt(0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0)** sẽ định vị camera tại vị trí (0.0, 0.0, 5.0), đường ngắm là hướng âm của trục oz (nhìn về gốc tọa độ) và hướng của đỉnh là hướng dương của trục oy

Mặc định vị trí của camera tại gốc tọa độ, đường ngắm là hướng âm của trục oz và hướng của đỉnh là hướng dương của trục oy



Hình 2.6 Vị trí mặc định của điểm nhìn là tại gốc tọa độ

4.2 Phép biến đổi mô hình

Bản chất phép biến đổi mô hình là xác định vị trí và hướng của mô hình (đối tượng cần vẽ trong không gian). Ví dụ chúng ta có thể áp dụng các phép tịnh tiến (translation), phép quay (rotation), phép tỉ lệ (scale) hoặc kết hợp các phép biến đổi đó với nhau. Một số lệnh liên quan đến biến đổi mô hình của OpenGL

Phép tịnh tiến: **glTranslate{fd}(TYPE x, TYPE y, TYPE z);**

Ý nghĩa: Tịnh tiến mô hình theo véc tơ tịnh tiến (*x*, *y*, *z*)

Phép quay: **glRotate{fd}(TYPE angle, TYPE x, TYPE y, TYPE z);**

Ý nghĩa: Quay mô hình một góc *angle* ngược chiều kim đồng hồ xung quanh tia nối từ gốc tọa độ đến điểm (*x*, *y*, *z*)

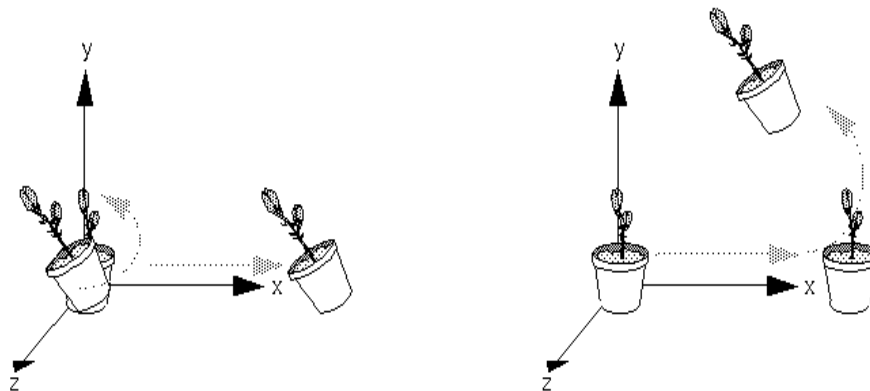
Phép tỉ lệ: **glScale{fd}(TYPE x, TYPE y, TYPE z);**

Ý nghĩa: Biến đổi tỉ lệ mô hình với hệ số tỉ lệ tương ứng với ba trục ox, oy, oz lần lượt là x, y, z

Chú ý: Với $x=-1, y=1, z=1$ thì phép tỉ lệ trở thành phép đối xứng qua mặt phẳng oyz, tương tự với mặt phẳng oxy và oxz.

4.3 Kết hợp các phép biến đổi

Trong khi biểu diễn đồ họa thường ta phải kết hợp nhiều phép biến đổi, khi đó thứ tự áp dụng các phép biến đổi là rất quan trọng. Ta hãy xét ví dụ trong hình vẽ 2.6 khi áp dụng liên tiếp hai phép biến đổi là phép quay và phép tịnh tiến với một vật thể được vẽ tại gốc tọa độ. Hình vẽ bên trái minh họa kết quả khi ta áp dụng phép quay trước còn ở hình vẽ bên phải ta áp dụng phép tịnh tiến trước. Dễ dàng nhận xét rằng kết quả thu được là khác nhau:



Hình 2.7 Quay trước hoặc tịnh tiến trước

OpenGL biểu diễn các phép biến đổi bằng ma trận vuông 4×4 , có một ma trận gọi là ma trận hiện thời (current matrix) lưu trạng thái hiện thời của các phép biến đổi. Trước khi áp dụng các phép biến đổi điểm nhìn và mô hình ta phải thiết lập chế độ ma trận (matrix mode) bằng câu lệnh **glMatrixMode(GL_MODELVIEW)** ($\text{MODELVIEW} = \text{MODEL} + \text{VIEWING}$) và thiết lập ma trận hiện thời về ma trận đơn vị I bằng câu lệnh **glLoadIdentity()**. Mỗi lần một phép biến đổi có hiệu lực OpenGL sẽ nhân ma trận hiện thời với ma trận tương ứng của phép biến đổi đó, kết quả được gán cho ma trận hiện thời. Giả sử các phép biến đổi được viết theo thứ tự là t_1, t_2, \dots, t_n và các ma trận biến đổi tương ứng là T_1, T_2, \dots, T_n , ma trận hiện thời được biến đổi từ I sang $I * T_1 = T_1$, sang $T_1 * T_2$, sang $T_1 * T_2 * T_3, \dots$, và cuối cùng là $T_1 * T_2 * \dots * T_n = T$.

Khi đối tượng được vẽ các đỉnh của nó được tính như sau:

$$v \rightarrow T * v = (T_1 * T_2 * \dots * T_n) * v = (T_1 * (T_2 * (\dots (T_{n-1} * (T_n * v)) \dots)))$$

Điều đó nghĩa là phép biến đổi T_n tác động đến đối tượng trước, tiếp đó là T_{n-1}, \dots cuối cùng là T_1 . Như vậy là thứ tự thực hiện các phép biến đổi ngược với thứ tự các câu lệnh (backwards).

Quay lại ví dụ ở hình vẽ 2.6, hình bên trái tương đương với đoạn mã chương trình

```
glTranslate*();
glRotate*();
Draw_Object();
```

Hình bên phải tương đương với đoạn mã chương trình

```
glRotate*();
glTranslate*();
Draw_Object();
```

4.4 Cách chuyển đổi từ phép biến đổi điểm nhìn sang phép biến đổi mô hình

Ta đã biết cách sử dụng câu lệnh **gluLookAt()** để thay đổi vị trí và hướng của camera, đây là câu lệnh thuộc OpenGL Utility Library (GLU). Ta cũng đã biết các câu lệnh của OpenGL thực hiện phép biến đổi mô hình đó là **glTranslate*()**, **glRotate*()** và **glScale*()**. Thực chất phép biến đổi điểm nhìn có thể được thay thế bởi một vài câu lệnh của phép biến đổi mô hình. Quay lại ví dụ **gluLookAt(0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0)** sau lệnh này camera được tịnh tiến từ vị trí mặc định là gốc tọa độ về điểm (0.0, 0.0, 5.0), điều đó tương đương với việc giữ nguyên camera tại gốc tọa độ và tịnh tiến vật thể 5 đơn vị về hướng ngược lại dọc theo trục oz **glTranslatef(0.0, 0.0, -5.0)**. Xét trường hợp tổng quát:

gluLookAt(eyex, eyey, eyez, centerx, centery, centerz, upx, upy, upz)

Để thay thế lệnh trên bằng các phép biến đổi mô hình ta áp dụng các phép biến đổi mục đích là đưa camera về vị trí mặc định (tại gốc tọa độ, hướng nhìn về hướng âm của trục oz và hướng đỉnh là hướng dương của trục oy).

Bước 1: Tịnh tiến camera về gốc tọa độ bằng cách áp dụng lệnh

glTranslatef(-eyex, -eyey, -eyez)

Bước 2: Áp dụng các phép quay để đưa hướng nhìn của camera về hướng âm của oz và đưa hướng đỉnh của camera về hướng dương của oy

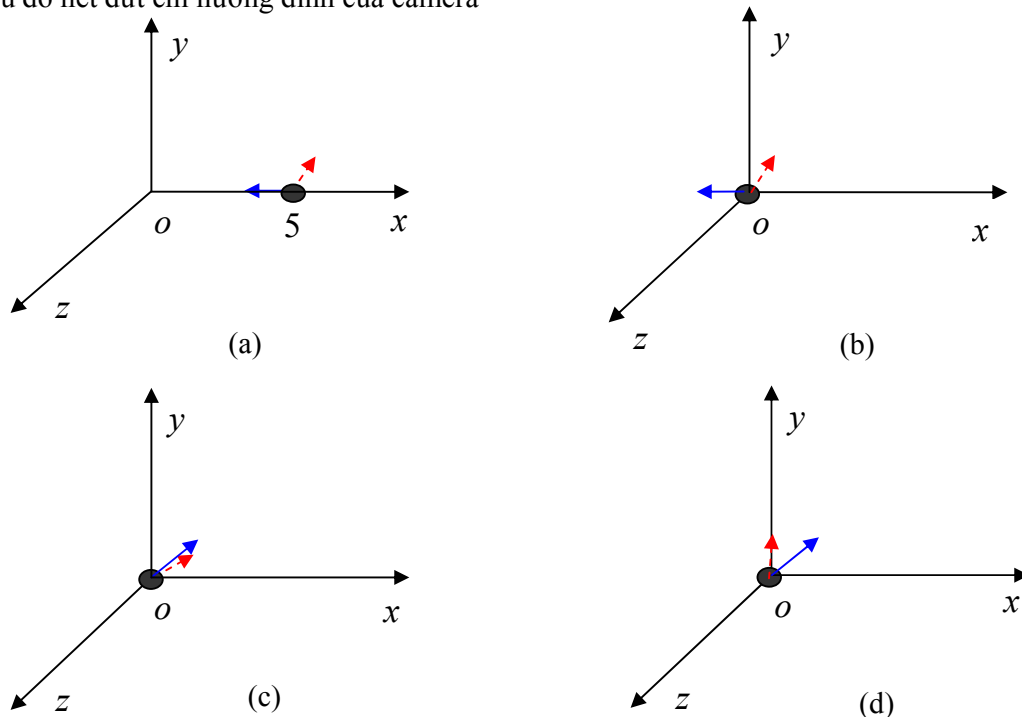
Ví dụ 2.3: Lệnh **gluLookAt(5.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, -1.0)**; được thay thế tương đương bởi các câu lệnh sau

glRotatef(45, 0.0, 0.0, 1.0); // Quay một góc 45 độ quanh trục oz đưa hướng đỉnh camera về hướng dương trục oy

glRotatef(-90, 0.0, 1.0, 0.0); // Quay một góc -90 độ quanh trục oy đưa hướng nhìn camera về hướng âm trục oz

glTranslatef(-5.0, 0.0, 0.0); // Đưa camera về gốc tọa độ

Chú ý: Thứ tự thực hiện các lệnh theo thứ tự ngược lại thứ tự của các câu lệnh. Quá trình được minh họa trên hình 2.7, mũi tên màu xanh chỉ hướng nhìn của camera và mũi tên màu đỏ nét đứt chỉ hướng đỉnh của camera



Hình 2.8 Chuyển biến đổi điểm nhìn sang biến đổi mô hình

5 Phép chiếu phối cảnh và phép chiếu trực giao (Perspective and Orthographic Projection)

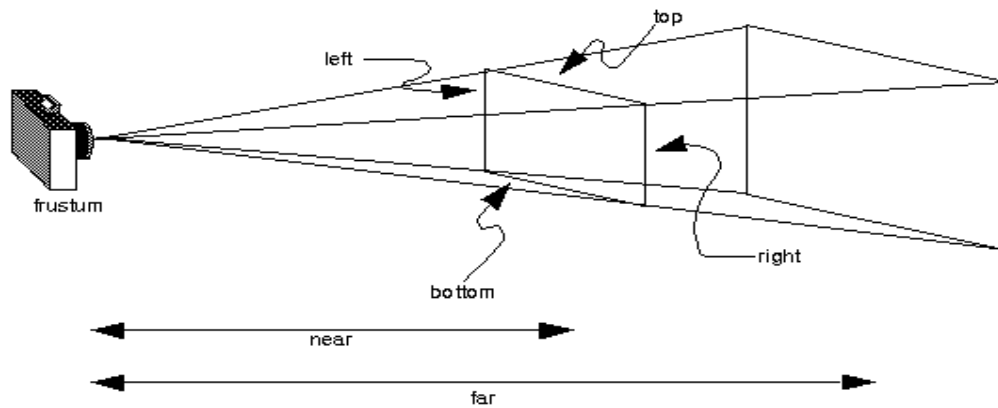
Mục đích của phép chiếu là định nghĩa viewing volume. Trước khi định nghĩa phép chiếu ta phải thiết lập chế độ ma trận bằng câu lệnh **glMatrixMode(GL_PROJECTION)**. Các lệnh định nghĩa phép chiếu không làm ảnh hưởng đến vị trí của camera được thiết lập bởi lệnh **gluLookAt()**.

5.1 Định nghĩa phép chiếu phối cảnh

Cách 1: Sử dụng lệnh của OpenGL

glFrustum(GLdouble *left*, GLdouble *right*, GLdouble *bottom*, GLdouble *top*, GLdouble *near*, GLdouble *far*);

Viewing volume được định nghĩa là một hình chóp cắt, đỉnh của hình chóp tại gốc tọa độ. Đáy lớn của viewing volume nằm trên mặt phẳng $z=-far$ và đáy nhỏ của viewing volume (ta còn gọi là khung nhìn) nằm trên mặt phẳng $z=-near$ là mặt phẳng mà vật thể sẽ được chiếu lên đó. Các giá trị *near* và *far* là các số dương.

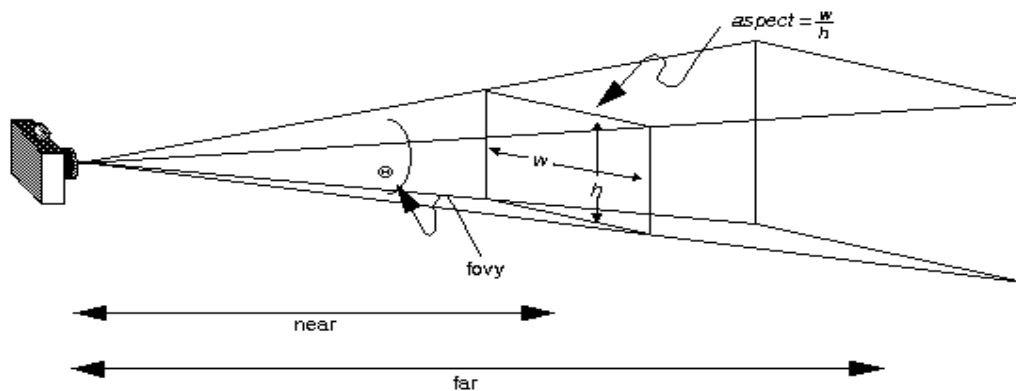


Hình 2.9 Viewing volume được định nghĩa bởi glFrustum()

Cách 2: Sử dụng lệnh của OpenGL Utility Library

gluPerspective(GLdouble *fovy*, GLdouble *aspect*, GLdouble *near*, GLdouble *far*);

Tác dụng của lệnh giống như lệnh glFrustum nhưng các tham số mang một ý nghĩa khác. Tham số *fovy* xác định góc tại đỉnh của hình chóp dọc theo mặt phẳng yz, tham số *aspect* chính là tỉ số giữa bề rộng và chiều cao đáy nhỏ của hình nón cắt, hai tham số *near* và *far* giống như lệnh glFrustum.



Hình 2.10 Viewing volume được định nghĩa bởi gluPerspective()

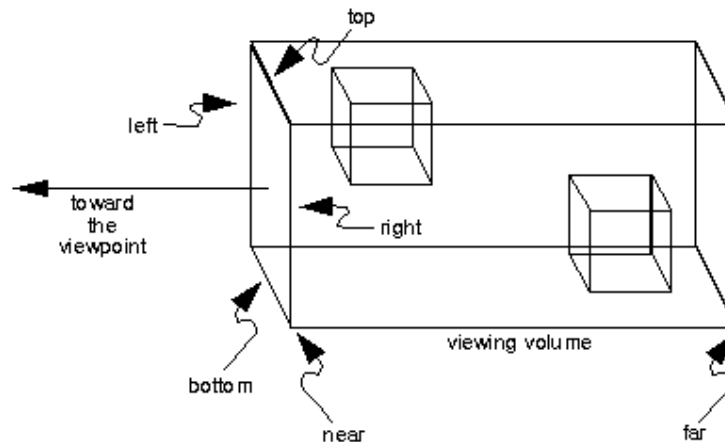
5.2 Định nghĩa phép chiếu trực giao

Với phép chiếu trực giao, viewing volume được định nghĩa là một hình hộp chữ nhật. Vật thể nằm trong viewing volume được chiếu trực giao lên khung nhìn do đó trong phép chiếu trực giao khoảng cách từ camera đến vật thể không ảnh hưởng đến độ lớn của ảnh. Phép chiếu trực giao thường được dùng nhiều trong các ứng dụng CAD/CAM ví dụ các chương trình ứng dụng tạo ra các bản vẽ kỹ thuật trong lĩnh vực thiết kế

Cách 1: Sử dụng lệnh của OpenGL

glOrtho(GLdouble *left*, GLdouble *right*, GLdouble *bottom*, GLdouble *top*, GLdouble *near*, GLdouble *far*);

Đối tượng nằm trong viewing volume được chiếu trực giao lên đáy của hình hộp chữ nhật nằm trên mặt phẳng $z=-near$, phương của phép chiếu song song với trục oz. Ảnh thu được không bị co lại và không có chiều sâu.



Hình 2.11 Viewing volume được định nghĩa bởi glOrtho()

Cách 2: Sử dụng lệnh của OpenGL Utility Library

glOrtho2D(GLdouble *left*, GLdouble *right*, GLdouble *bottom*, GLdouble *top*);

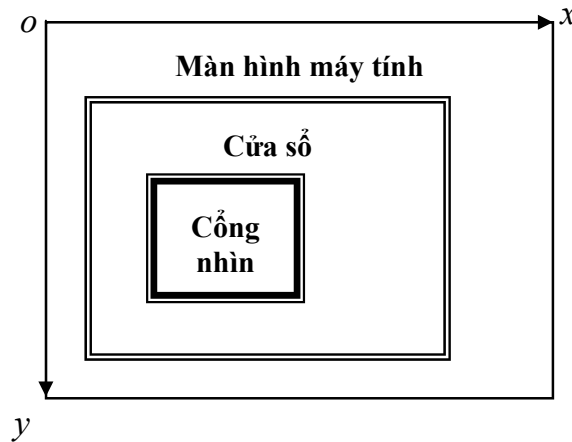
Lệnh có ý nghĩa giống như lệnh glOrtho() ngoại trừ khoảng giá z mặc định của vật thể nằm trong viewing volume mặc định là $[-1,1]$.

6 Phép biến đổi cổng nhìn (Viewport Transformation)

Cổng nhìn là một miền hình chữ nhật nằm phía trong một cửa sổ (window) đang mở trên màn hình cho phép hiển thị hình ảnh. Hình ảnh của vật thể sau khi được chiếu lên khung nhìn sẽ được ánh xạ lên cổng nhìn. Ta có thể định nghĩa nhiều cổng nhìn để hiển thị nhiều khung cảnh khác nhau trên cửa sổ. Định nghĩa cổng nhìn ta dùng câu lệnh sau:

glViewport(GLint *x*, GLint *y*, GLsizei *width*, GLsizei *height*);

Trong đó (*x*,*y*) xác định tọa độ góc trái phía dưới của cổng nhìn, tọa độ này là tọa độ trong cửa sổ đang mở. Góc trái phía dưới của cửa sổ có tọa độ là (0,0). Hai tham số *width* và *height* là chiều rộng và chiều cao của khung nhìn. Ở chế độ mặc định giá trị tham số của khung nhìn là (0, 0, *winWidth*, *winHeight*) trong đó *winWidth* và *winHeight* là kích thước của cửa sổ nơi hiển thị khung nhìn.



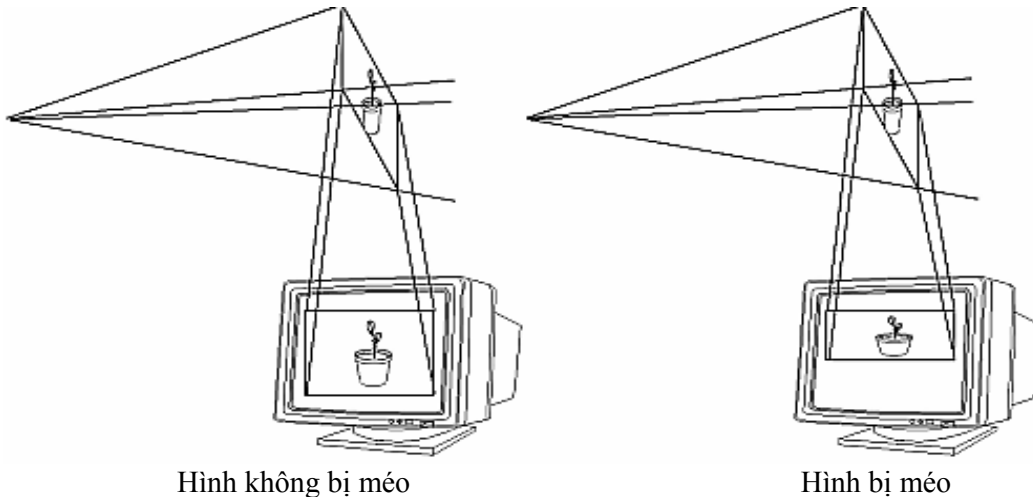
Hình 2.12 Màn hình máy tính, cửa sổ và cổng nhìn

Để định nghĩa cửa sổ trên màn hình ta dùng hai câu lệnh sau của GLUT:

glutInitWindowPosition (int x, int y); //Chỉ định vị trí góc trái trên của cửa sổ trong hệ tọa độ xoy của màn hình (Hình 2.11)

glutInitWindowSize (int Width, int Height); //Chỉ định kích thước của cửa sổ (pixels), hai kích thước thường được chọn bằng nhau để hình không bị méo

Tỉ lệ hai kích thước của cổng nhìn nên chọn bằng với tỉ lệ hai kích thước của khung nhìn. Nếu hai tỉ lệ này không bằng nhau thì hình ảnh hiển thị trên màn hình sẽ bị méo (distorted). Hình 2.11 minh họa điều này



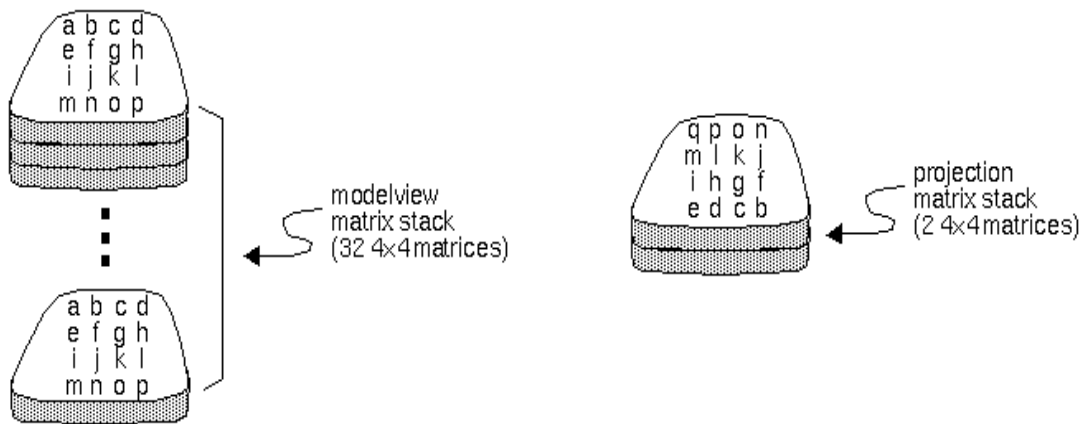
Hình 2.13 Ảnh được ánh xạ từ khung nhìn lên cổng nhìn

7. Điều khiển các ngăn xếp ma trận (Matrix Stacks)

OpenGL biểu diễn các phép biến đổi điểm nhìn và mô hình bằng ngăn xếp ma trận Modelview (Modelview matrix stack). Ngăn xếp này có tối đa là 32 phần tử là các ma trận vuông 4x4 tương ứng các phép biến đổi tác động lên đối tượng vẽ và ma trận hiện thời luôn là ma trận nằm trên đỉnh của stack. Mỗi một phép biến đổi điểm nhìn hoặc biến đổi mô hình tạo ra một ma trận mới, ma trận này được nhân với ma trận hiện thời và kết quả trở thành ma trận hiện thời mới (ngăn xếp thêm 1 phần tử mới trên đỉnh).

Để biểu diễn các phép chiếu, OpenGL sử dụng một ngăn xếp khác được gọi là ngăn xếp ma trận phép chiếu (Projection matrix stack) gồm tối đa 2 phần tử ma trận vuông 4x4. Ma trận

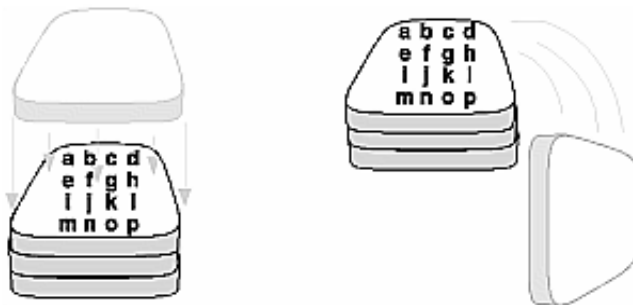
hiện thời mô tả phép chiếu đang được áp dụng cho đối tượng vẽ, hay nói cách khác ma trận này mô tả viewing volume.



Hình 2.14 Ngăn xếp ma trận modelview và ma trận phép chiếu

Ta có thể điều khiển ma trận hiện thời bằng các câu lệnh sau:

```
glLoadIdentity(); //Thiết lập ma trận hiện thời về ma trận đơn vị
glLoadMatrix{fd}(const TYPE *m); // Thiết lập giá trị cho ma trận hiện thời từ 16 giá
trị được trả bởi m (m là ma trận một chiều hoặc hai chiều)
glMultMatrix{fd}(const TYPE *m); //Nhân ma trận có 16 giá trị được trả bởi m với
ma trận hiện thời, kết quả là ma trận hiện thời mới
glPushMatrix(); //Sao chép thêm một ma trận hiện thời và đưa lên đỉnh của ngăn xếp
glPopMatrix(); //Loại bỏ ma trận hiện thời khỏi ngăn xếp
```



Hình 2.15 Sao chép và loại bỏ ma trận hiện thời

Lệnh **glPushMatrix();** được hiểu là "Ghi nhớ bạn đã ở đâu" (remember where you are) và **glPopMatrix();** được hiểu là "Quay lại vị trí bạn đã ở" (go back to where you were). Đây là hai câu lệnh rất hữu ích của OpenGL. Ví dụ ta muốn vẽ một xe ô tô có bốn bánh, quá trình vẽ có thể được mô tả như sau: Vẽ thân xe. Ghi nhớ bạn đã ở đâu, tịnh tiến về vị trí bánh xe phải phía trước. Vẽ bánh xe, quay lại vị trí bạn đã ở (đưa thân xe về vị trí trước khi tịnh tiến). Ghi nhớ bạn đã ở đâu, tịnh tiến về vị trí bánh xe trái phía trước....

Việc tổ chức các ma trận theo ngăn xếp là rất phù hợp khi ta cần xây dựng các mô hình có sự phân cấp (hierarchical models) trong đó một đối tượng phức tạp được xây dựng từ những đối tượng đơn giản hơn. Đặc biệt khi đó là một hình ảnh động, sự thay đổi trạng thái của một bộ phận kéo theo sự thay đổi của nhiều bộ phận khác.

Giả sử ta có đoạn chương trình như sau mô tả chuyển động một cánh tay của một robot:

```
glTranslatef() ; //Lệnh T1
glRotatef() ; //Lệnh T2

glPushMatrix() ;
glTranslatef() ; //Lệnh T3
glRotatef() ; //Lệnh T4
gluCylinder() ; //Vẽ cánh tay phía dưới
glPopMatrix() ;

glScalef() ; //Lệnh T5
gluCylinder() ; //Vẽ cánh tay phía trên
```

Đoạn chương trình trên có thể có thể được diễn tả bởi sơ đồ sau

T1 → T2 → T5 → Cánh tay phía trên
 ↓
 T3 → T4 → Cánh tay phía dưới

Muốn biết ảnh hưởng của các câu lệnh với hai hình lăng trụ ta duyệt sơ đồ trên theo thứ tự ngược lại.

Đây là những gì tác động lên cánh tay phía trên:

T5: Kéo dẫn đối tượng

T2: Quay đối tượng quanh một trục xuyên qua tâm của nó

T1: Tịnh tiến đối tượng

Đây là những gì tác động lên cánh tay phía dưới:

T4: Quay đối tượng quanh một trục xuyên qua trọng tâm của nó

T3: Tịnh tiến đối tượng

T2: Quay đối tượng một lần nữa, lần này trục quay không qua tâm của nó

T1: Tịnh tiến đối tượng một lần nữa

Nhận xét:

- Phép quay T2 tương đương với động tác quay tại khớp vai và phép quay T4 tương đương với động tác quay tại khuỷu tay

- T3 và T4 nằm giữa hai lệnh glPushMatrix() và glPopMatrix() chỉ tác động lên cánh tay phía dưới

- Lệnh T1 và T2 tác động lên cánh tay phía trên và đương nhiên cũng có tác động đến cánh tay phía dưới vì khi cánh tay phía trên chuyển động thì cánh tay phía dưới cũng chuyển động theo.