Chương 3: QUAN SÁT

Mục đích chương

Sau khi đọc xong chương này bạn có thể:

- Quan sát mô hình hình học ở mọi hướng bằng việc biến đổi nó trong không gian ba chiều.
- Điều khiển được vị trí trong không gian 3 chiều từ mô hình được quan sát.
- Cắt bớt những phần không cần thiết của mô hình khỏi cảnh quan sát.
- Thao tác các ngăn xếp ma trận thích hợp mà chúng điều khiển biến đổi mô hình để quan sát và chiếu mô hình trên màn hình.
- Kết hợp nhiều biến đổi để bắt chước hệ thống phức tạp khi di chuyển, như là hệ mặt trời hay một cánh tay robot có khớp nối.
- Đảo ngược hoặc bắt chước các hoạt động của quy trình xử lý hình học.

Chương 2 giải thích việc chỉ dẫn OpenGL như thế nào để vẽ ra những mô hình hình học bạn muốn hiển thị trong cảnh. Bây giờ bạn phải quyết định cách bạn muốn đặt mô hình trong cảnh như thế nào và phải chọn một vị trí thích hợp để quan sát cảnh. Bạn có thể sử dụng đặt vị trí mặc định và điểm thích hợp, nhưng đa phần bạn muốn chỉ ra chúng.

Xét ảnh trên trang bìa của cuốn sách này. Chương trình tạo ảnh đó chứa một hình học đơn mô tả một khối làm sẵn để xây dựng. Mỗi khối đã được bố trí một cách cẩn thận trong cảnh: Một số khối được để rải rác trên sàn, một số được sắp xếp trên mặt bàn và một số được tập hợp để tạo thành địa cầu. Ngoài ra, một điểm quan sát cụ thể đã được lựa chọn. Rõ ràng, chúng ta muốn nhìn từ góc của căn phòng chứa địa cầu. Nhưng xa như thế nào-và vị trí chính xác của người quan sát nên là bao nhiêu? Chúng ta muốn chắc chắn rằng ảnh cuối cùng của cảnh được đặt trong một quan sát hợp lí bên ngoài cửa sổ, một phần của nền nhà có thể thấy được, và tất cả các đối tượng trong cảnh không chỉ thấy được mà còn biểu diễn theo cách sắp đặt trung tâm. Chương này giải thích việc sử dụng OpenGL như thế nào để hoàn thiện các công việc này: làm thế nào để bố trí và xây các mô hình trong không gian ba chiều và làm thế nào để thiết lập vị trí- cũng trong không gian ba chiều- của điểm nhìn. Tất cả nhân tố này giúp xác định chính xác ảnh xuất hiện như thế nào trên màn hình.

Bạn nhớ rằng điểm trong đồ họa máy tính là để tạo ảnh hai chiều của những đối tượng ba chiều (nó phải là hai chiều vì nó được vẽ trên màn hình phẳng) nhưng bạn cần nghĩ về tọa độ ba chiều khi đang tạo nhiều quyết định xác định những gì vẽ trên màn hình. Một sai lầm phổ biến của người xây dựng khi đang tạo đồ họa ba chiều là bắt đầu nghĩ quá sớm đến ảnh cuối cùng xuất hiện trên một mặt phẳng, màn hình hai chiều. Hãy tránh việc nghĩ về những điểm nào cần được vẽ, và thay vào đó thử chúng không gian ba chiều. Tạo những mô hình của bạn trong mô hình ba chiều mà độ sâu nằm phía bên trong máy tính của bạn, và để cho máy tính làm công việc tính toán điểm ảnh nào được tô màu.

Một dãy ba phép toán máy tính chuyển đổi các tọa độ ba chiều của đối tượng đến vị trí điểm ảnh trên màn hình.

- Biến đổi, chúng được biểu diễn bởi phép nhân ma trận, bao gồm làm mô hình, quan sát và phép chiếu. Các phép toán đó bao gồm phép quay, dịch chuyển, co dãn, phản xạ, phép chiếu trực giao, và phép chiếu phối cảnh. Nói chung, bạn sử dụng một kết hợp của nhiều phép biến đổi để vẽ một cảnh.
- Do cảnh được tô vẽ trên cửa sổ hình chữ nhật, đối tượng (hay những phần của đối tượng) mà nằm ngoài cửa sổ phải được cắt xén. Trong đồ họa máy tính ba chiều, việc cắt xén thực hiện bằng việc loại di chuyển những đối tượng nằm một phía của mặt phẳng cắt xén.
- Cuối cùng, một sự phù hợp phải được thiết lập giữa các tọa độ biến đổi và các điểm ảnh màn hình. Điều này được biết như là một biến đổi *viewport.*

- Chương này mô tả tất cả các phép toán này, và làm thế nào để điều khiển chúng, trong những phần chính sau đây:

"Nhìn tổng quan: giống như máy quay phim" cho một cái nhìn tổng quan của phép biến đổi xử lí bằng việc mô tả giống như việc thu một ảnh của một máy quay phim, đưa ra một chương trình ví dụ đơn giản mà biến đổi một đối tượng, và mô tả ngắn gọn các lệnh biến đổi cơ bản của OpenGL.

"Biến đổi quan sát và mô hình" giải thích chi tiết làm thế nào để xác định và tưởng tượng tác động của biến đổi quan sát và mô hình. Những biến đổi này định hướng mô hình và máy quay phim liên quan với nhau để thu được ảnh cuối cùng như mong muốn.

"Biến đổi phép chiếu" Mô tả làm thế nào để xác định hình dạng và hướng của *khối quan sát*. Khối quan sát xác định cảnh được chiếu như thế nào lên màn hình (với một phép chiếu phối cảnh hay trực giao) và những đối tượng nào hay những phần của đối tượng được cắt xén của cảnh.

"Biến đổi cổng nhìn" giải thích làm thế nào để điều khiển chuyển đổi của tọa độ mô hình ba chiều sang tọa độ màn hình.

"Biến đổi gỡ rối" đưa ra một số mạo để tìm ra tại sao bạn không thể có tác động mong muốn từ những biến đổi mô hình, quan sát, chiếu và cổng nhìn của bạn.

"Thao tác ngăn xếp ma trận" mô tả làm thế nào để xác định mặt phẳng cắt ngoài những phần được định nghĩa bởi khối quan sát.

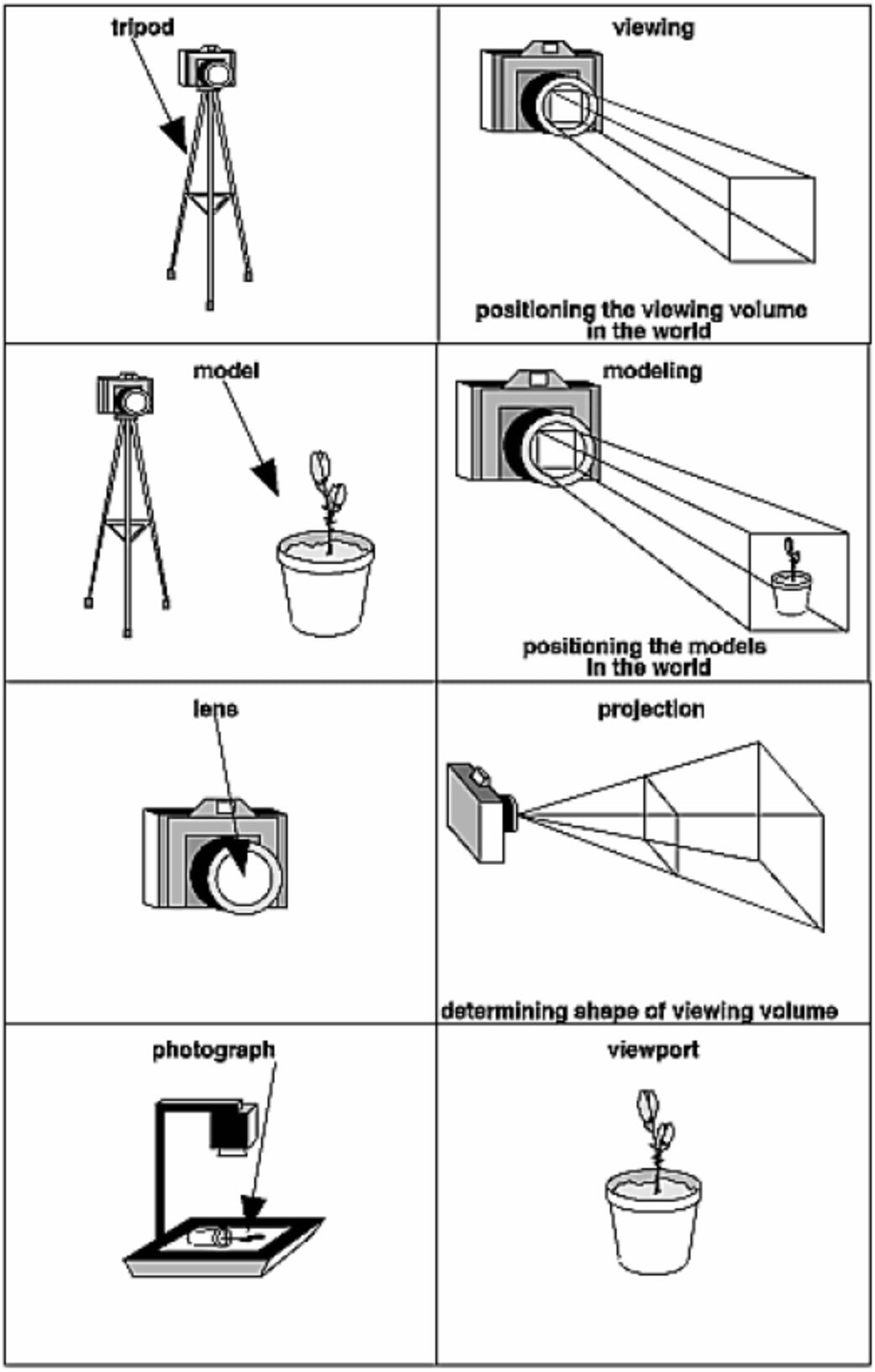
"Ví dụ của một số biến đổi" nói cho bạn làm thế nào để lấy một điểm đã biến đổi trong tọa độ cửa sổ và đảo ngược biến đổi để thu được tọa độ đối tượng ban đầu. Bản thân biến đổi (mà không hoán vị) cũng có thể được mô phỏng.

Tổng quan: Mô phỏng máy quay phim

Xử lí biến đổi để tạo cảnh mong muốn giống như việc thu một ảnh của một máy quay phim. Như chỉ ra trong Hình 3-1, các bước với một máy quay phim (hay một máy tính) có thể như sau:

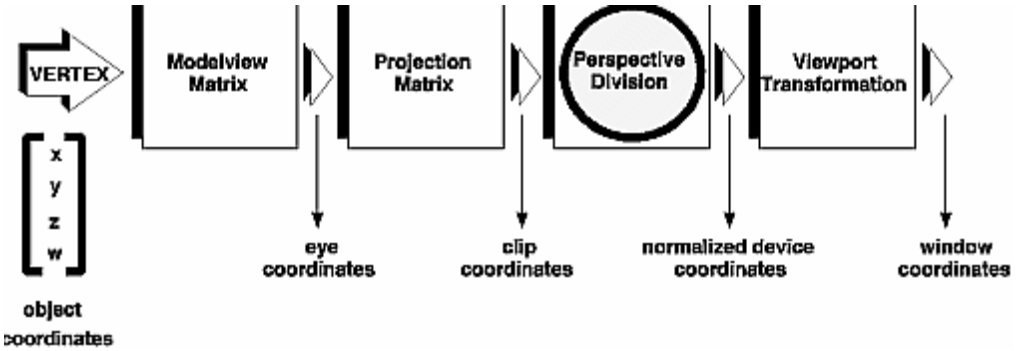
1. Thiết lập giá ba chân của bạn và đặt máy quay phim vào cảnh (biến đổi quan sát)
2. Sắp đặt cảnh để chụp thành tác phẩm mong muốn (Biến đổi mô hình)
3. Lựa chọn thấu kính máy quay phim và điều chỉnh độ phóng (biến đổi phép chiếu)
4. Quyết định bạn muốn độ lớn ảnh cuối cùng như thế nào – ví dụ, bạn có thể muốn nó mở rộng (biến đổi cổng nhìn)

Sau khi các bước này được thực hiện, ảnh có thể được căn lề hay cảnh có thể được vẽ.



Hình 3-1. Mô phỏng máy quay phim

Chú ý rằng những bước này tương ứng với trật tự mà bạn chỉ ra những biến đổi mong muốn trong chương trình của bạn, không nhất thiết phải theo thứ tự mà phép toán toán học liên quan được thực hiện trên một đỉnh của đối tượng. Những biến đổi quan sát phải được có trước biến đổi mô hình trong đoạn mã của bạn, nhưng bạn có thể chỉ định biến đổi phép chiếu và biến đổi cổng nhìn tại mọi điểm trước khi việc vẽ diễn ra. Hình 3-2 chỉ ra thứ tự mà phép toán này xảy ra trên máy tính



Hình 3-2: Các giai đoạn của biến đổi đỉnh

Để chỉ ra các phép biến đổi quan sát, mô hình, và phép chiếu, bạn hãy tạo ma trận M 4x4, sau đó chúng được nhân với tọa độ của mỗi đỉnh v trong cảnh để thực hiện biến đổi.

v'=Mv

(Nhớ rằng các đỉnh luôn luôn có 4 tọa độ(x, y, z, w), mặc dù phần lớn các trường hợp w là 1 và đối với dữ liệu hai chiều z là 0). Chú ý rằng biến đổi quan sát và mô hình được tự động áp dụng cho bề mặt của vectơ pháp tuyến, cùng với các đỉnh. (những vectơ pháp tuyến được sử dụng duy nhất trong các tọa độ mắt). Điều này đảm bảo mối quan hệ của vectơ pháp tuyến với dữ liệu đỉnh là được duy trì hợp lí.

Biến đổi quan sát và mô hình được kết hợp để tạo thành ma trận quan sát mô hình, chúng được áp dụng cho tọa độ đối tượng đến tọa độ mắt. Tiếp theo, nếu bạn đã chỉ rõ các mặt phẳng cắt bổ sung để xóa đi những đối tượng nào đó từ cảnh hay để cung cấp quan sát phần cắt bên trong của đối tượng, những mặt phẳng cắt này được áp dụng.

Sau đó, Open GL áp dụng với ma trận chiếu để sinh ra *tọa độ cắt*. Biến đổi này định nghĩa khối quan sát; các đối tượng bên ngoài khối này này bị loại bỏ sao cho chúng không được vẽ trong cảnh cuối cùng. Sau giai đoạn này, phép chia phối cảnh được thực hiện bằng việc chia các giá trị tọa độ cho w, để tạo ra *tọa độ thiết bị chuẩn hóa*.(Xem Phụ lục F để biết thêm thông tin về ý nghĩa của tọa độ w và nó tác động như thế nào biến đổi ma trận). Cuối cùng tọa độ biến đổi được chuyển đến tọa độ cửa sổ bằng việc áp dụng biến đổi cổng nhìn. Bạn có thể điều chỉnh hướng của cổng nhìn để có được ảnh cuối cùng rộng ra, co lại hay kéo dài. Bạn có thể giả sử tọa độ x, y là đủ để xác định điểm nào cần được vẽ trên màn hình. Tuy nhiên, mọi biến đổi được thực hiện trên tọa độ z cũng không sao. Cách này, tại phần cuối của quá trình biến đổi, giá trị z phản ánh một cách chính xác độ sâu của đỉnh đã cho (đo theo khoảng cách tính từ màn hình). Sử dụng một cho giá trị độ sâu này là để giới hạn việc vẽ không cần thiết. Ví dụ, giả sử hai đỉnh có cùng giá trị x và y nhưng z khác nhau. OpenGL có thể sử dụng thông tin này để quyết định xem bề mặt được che khuất bởi các bề mặt khác và có thể tránh vẽ những bề mặt ẩn. (Xem Chương 10 để biết thêm thông tin về kĩ thuật này, chúng được gọi là *khử mặt khuất.*)

Bây giờ, có thể bạn đã phỏng đoán, bạn cần biết một chút về ma trận toán học để hiểu được chương này. Nếu bạn muốn nâng cao kiến thức của mình trong phần này, bạn có thể tham khảo cuốn sách về đại số tuyến tính.

Một ví dụ đơn giản: Vẽ một hình lập phương

Hình 3-1 vẽ hình lập phương mà được co dãn bởi một biến đổi mô hình (xem Hình 3-3) biến đổi quan sát, **GluLook At()**, những vị trí và đích máy quay phim hướng về phía hình lập phương được vẽ. Một biến đổi phép chiếu và biến đổi cổng nhìn cũng được chỉ ra. Phần còn lại của phần này cho bạn xem qua Ví dụ 3-1 và giải thích một cách ngắn gọn lệnh biến đổi nó sử dụng. Những phần tiếp theo bao gồm hoàn thiện, trình bày chi tiết của tất cả các lệnh biến đổi OpenGL.

Hình 3-3: Hình lập phương qua biến đổi.

Ví dụ 3-1 : Hình lập phương qua biến đổi: cube.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
void init(void)
{
glClearColor (0.0, 0.0, 0.0, 0.0);
glShadeModel (GL_FLAT);
}
void display(void)
{
glClear (GL_COLOR_BUFFER_BIT);
glColor3f (1.0, 1.0, 1.0);
glLoadIdentity (); /* clear the matrix */
/* viewing transformation */
gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
glScalef (1.0, 2.0, 1.0); /* modeling transformation */
glutWireCube (1.0);
glFlush ();
}
void reshape (int w, int h)
{
glViewport (0, 0, (GLsizei) w, (GLsizei) h);
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
glFrustum (-1.0, 1.0, -1.0, 1.0, 1.5, 20.0);
glMatrixMode (GL_MODELVIEW);
}
int main(int argc, char** argv)
{
```

```
glutInit(&argc, argv);
glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
glutInitCửa sổSize (500, 500);
glutInitCửa sổPosition (100, 100);
glutCreateCửa sổ (argv[0]);
init ();
glutDisplayFunc(display);
glutReshapeFunc(reshape);
glutMainLoop();
return 0;
}
```

Phép biến đổi quan sát

Nhắc lại rằng biến đổi quan sát là tương tự vị trí và hướng của máy quay phim. Trong ví dụ đoạn mã này, trước khi biến đổi quan sát có thể được xác định, ma trận hiện thời được thiết lập ma trận đơn vị với **glLoad Identity().** Bước này là cần thiết vì phần lớn lệnh biến đổi nhân với ma trận hiện thời với ma trận chỉ định và sau đó thiết lập kết quả cho ma trận hiện thời. Nếu bạn không xóa ma trận hiện thời bằng việc chuyển nó về ma trận đơn vị, bạn tiếp tục kết hợp với ma trận biến đổi trước với một ma trận mới mà bạn đưa ra. Trong vài trường hợp, bạn muốn thực hiện những kết hợp đó, đôi khi bạn vẫn cần phải xóa những ma trận đó.

Trong Ví dụ 3-1, sau khi ma trận được khởi tạo, biến đổi quan sát được xác định với **gluLookAt().** Đối số cho lệnh này chỉ ra vị trí được đặt của máy quay phim (hay vị trí mắt nhìn), vị trí nó hướng tới hay hướng nào là đỉnh đầu. Đối số sử dụng đặt máy quay phim ở đây là (0,0,5), hướng thấu kính máy quay phim là (0, 0, 0) và xác định vecto hướng đỉnh như (0, 1, 0). Vecto hướng đỉnh định nghĩa là hướng duy nhất cho máy quay phim. Nếu **gluLookAt()** không được gọi, máy quay phim có vị trí và hướng mặc định . Mặc định, máy quay phim được đặt ở gốc, trỏ xuống chiều âm trục z, và có vecto hướng đỉnh (0, 1, 0). Trong Ví dụ 3-1, tác động tổng thể là **gluLookAt()** di chuyển máy quay phim 5 đơn vị theo trục z. (Xem "Các biến đổi quan sát và mô hình" để biết thêm thông tin về biến đổi quan sát)

Biến đổi mô hình

Bạn sử dụng biến đổi mô hình để đặt vị trí và định hướng mô hình. Ví dụ, bạn có thể quay, tịnh tiến, hay tỉ lệ mô hình hoặc thực hiện một số kết hợp các phép toán này. Trong Ví dụ 3-1, **glScale()** là biến đổi đối tượng được sử dụng. Những đối số của lệnh này chỉ ra tỉ lệ xảy ra như thế nào trên 3 trục. Nếu tất cả các đối số là 1.0, lệnh này không có tác dụng. Trong Ví dụ 3-1, hình lập phương được vẽ với độ lớn gấp đôi theo trục y. Vì vậy, nếu một góc của hình lập phương ban đầu là (3.0, 3.0, 3.0), góc đó biến đổi thành (3.0, 6.0,3.0). Kết quả của biến đổi mô hình này là biến đổi hình lập phương sao cho nó không còn là hình lập phương mà là một hình hộp chữ nhật.

Thử điều này

Thay đổi **gluLookAt()** trong Ví dụ 3-1 theo biến đổi mô hình **glTranslatef()** với tham số (0.0,0.0,5.0). Kết quả sẽ thấy giống như khi bạn sử dụng **gluLookAt()**. Tại sao kết quả của hai lệnh này lại giống nhau?

Chú ý rằng thay vì di chuyển máy quay phim (với biến đổi quan sát) sao cho hình lập phương được nhìn thấy, bạn có thể di chuyển hình lập phương xa máy quay phim (với một biến đổi mô hình). Tính đối ngẫu bản chất của biến đổi mô hình và quan sát là lý do bạn cần nghĩ kết quả của cả hai kiểu biến đổi một cách đồng thời. Điều đó không có nghĩa là phân biệt các kết quả nhưng đôi khi dễ dàng

hơn nghĩ về chúng theo một cách hơn là những cách khác. Đây cũng là lí do biến đổi quan sát và mô hình được kết hợp thành *ma trận mô hình quan sát* trước khi áp dụng các biến đổi.(Xem "Biến đổi mô hình và quan sát" để biết thêm thông tin về biến đổi mô hình và quan sát và xác định chúng như thế nào để có kết quả mong muốn.)

Cũng nên chú ý rằng biến đổi quan sát và mô hình được chứa trong thủ tục **display()**, cùng với lời gọi được sử dụng để vẽ hình lập phương, **glutWireCube()**. Cách này, **display()** có thể được lặp để vẽ nội dung của cửa sổ nếu, ví dụ, cửa sổ bị di chuyển hay không được bao phủ hết và bạn đảm bảo rằng mỗi lần, hình lập phương được vẽ theo cách mong muốn, với những biến đổi thích hợp. Cách sử dụng lặp lại của **display()** nhấn mạnh cần gọi ma trận đơn vị trước khi thực hiện biến đổi quan sát và mô hình, đặc biệt khi các biến đổi khác có thể được thực hiện giữa những lần gọi **display().**

Biến đổi phép chiếu

Xác định biến đổi phép chiếu giống như việc chọn thấu kính cho máy quay phim. Bạn có thể coi sự biến đổi này như việc xác định phạm vi quan sát hay khối quan sát là gì và do đó những đối tượng nào bên trong nó và quy mô chúng như thế nào. Điều này tương đương với việc chọn góc rộng, pháp tuyến, và thấu kính chụp xa, ví dụ. Với một thấu kính góc rộng, này bạn có thể bao trùm cảnh rộng hơn trong ảnh cuối hơn là với thấu kính chụp xa, nhưng một thấu kính chụp xa cho phép bạn chụp đối tượng như chúng đang ở gần với bạn hơn vị trí thực của chúng**.** Trong đồ họa máy tính, bạn không phải trả 10,000$ cho một thấu kính chụp xa 2000milimet; mỗi khi bạn mua trạm làm việc đồ họa của bạn, tất cả bạn cần làm là sử dụng một số nhỏ hơn đối với phạm vi quan sát của bạn.

Cùng với việc đưa ra phạm vi quan sát, biến đổi phép chiếu xác định đối tượng được *chiếu* như thế nào trên màn hình. Hai loại cơ bản chiếu được cung cấp bởi OpenGl, với một số lệnh tương ứng để miêu tả các thông số liên quan theo cách khác nhau. Một kiểu là phép chiếu phối cảnh, chúng giống với những gì bạn thấy trong cuộc sống hàng ngày. Phối cảnh làm cho những đối tượng xa hơn thì xuất hiện nhỏ hơn; ví dụ, nó làm cho đường ray hội tụ phía xa. Nếu bạn thử tạo các bức ảnh thực tế, bạn sẽ muốn chọn phép chiếu phối cảnh, chúng được chỉ ra với lệnh **glFrustum()** trong đoạn mã ví dụ**.**

Một kiểu phép chiếu khác là trực giao, chúng ánh xạ các đối tượng trực tiếp lên màn hình mà không ảnh hưởng gì đến kích thước liên quan của chúng. Phép chiếu trực giao là được sử dụng trong ứng dụng kiến trúc và thiết kế với hỗ trợ máy tính nơi mà hình ảnh cần phản xạ kích thước của đối tượng hơn là chúng có thể trông như thế nào. Các kiến trúc sư tạo ra các bản vẽ phối cảnh để chỉ ra các tòa nhà cụ thể hay không gian trông như thế nào khi tạo bản đồ thiết kế hay mặt chiếu, chúng được sử dụng trong kiến trúc của tòa nhà. (Xem "biến đổi phép chiếu" thảo luận các cách để xác định cả hai loại biến đổi phép chiếu)

Trước khi **glFrustum()** có thể được gọi để thết lập biến đổi phép chiếu, cần một số chuẩn bị. Như thể hiện trong thủ tục **reshape()** ở Ví dụ 3-1, lệnh được gọi ra đầu tiên là **glMatrixMode()** với đối số GL_PROJECTION. Điều này chỉ ra rằng ma trận hiện tại xác định biến đổi phép chiếu, biến đổi sau đây gọi sau tác đông của ma trận phép chiếu. Khi bạn có thể thấy, một số dòng **glMatrixMode()** sau đó được gọi lại, lần này với GL_MODELVIEW như là một đối số. Điều này chỉ ra rằng hiện tại biến đổi thành công ảnh hưởng ma trận mô hình quan sát thay thế ma trận chiếu. (Xem "thao tác ngăn xếp ma trận" để biết thêm thông tin về điều khiển ma trận mô hình quan sát và phép chiếu.)

Chú ý rằng **glLoadIdentity**() được sử dụng để khởi tạo ma trận phép chiếu hiện tại sao cho chỉ biến đổi pháp chiếu được chỉ định có kết quả. Bây giờ **glFrustum()** có thể được gọi. với đối số được định nghĩa thông số của biến đổi phép chiếu. Trong ví dụ này, cả hai phép biến đổi phép chiếu và biến đổi quan sát được bao gồm trong thủ tục **reshape(),** chúng được gọi khi cửa sổ được tạo ra đầu tiên và

bất cứ khi nào cửa sổ được di chuyển hay chỉnh kích cỡ. Điều này tạo cảm giác, vì cả hai việc chiếu (tỉ lệ co chiều rộng và chiều cao của phép chiếu trong khối quan sát) và áp dụng công nhìn liên quan trực tiếp tới màn hình, và đặc biệt tới kích cỡ và tỉ lệ co của cửa sổ trên màn hình.

Thử điều này

Thay đổi lời gọi **glFrustum()** trong Ví dụ 3-1 với thủ tục Utility Library sử dụng phổ biến hơn **gluPerspective()** với tham số (60.0, 1.0, 1.5, 20.0). Sau đó thử với giá trị khác nhau, đặc biệt với fovy và aspect.

Phép biến đổi cổng nhìn

Cũng như, biến đổi phép chiếu và biến đổi cổng nhìn xác định cảnh được ánh xạ như thế nào lên màn hình máy tính. Biến đổi phép chiếu chỉ ra cơ chế của việc ánh xạ diễn ra như thế nào và cổng nhìn chỉ ra hình dạng vùng màn hình có sẵn thành cảnh được ánh xạ. Do cổng nhìn xác định vùng ảnh chiếm trên màn hình, bạn có thể nghĩ phép biến đổi cổng nhìn như việc định nghĩa kích thước và vị trí của ảnh xử lí cuối cùng - ví dụ, liệu ảnh nên dãn ra hay co lại. Đối số của **glViewport**() miêu tả gốc tọa độ của không gian màn hình có sẵn trong cửa sổ -(0,0) trong ví dụ này - và độ rộng và độ cao của vùng màn hình có sẵn, tất cả được đo theo đơn vị điểm ảnh trên màn hình. Đây là lý do tại sao lệnh này cần được gọi bên trong **reshape()**- nếu cửa sổ thay đổi kích thước, cổng nhìn cần thay đổi kích thước tương ứng. Chú ý rằng độ rộng và chiều cao được chỉ ra với việc sử dụng với kích thước thực của cửa sổ, thường là bạn muốn xác định cổng nhìn theo cách này hơn là đưa ra một kích thước tuyệt đối. (Xem "Biến đổi cổng nhìn" để biết thêm thông tin về định nghĩa cổng nhìn.)

Vẽ cảnh

Mỗi khi tất cả những biến đổi cần thiết được xác định, bạn có thể vẽ cảnh (tức là tạo một ảnh). Khi cảnh được vẽ, OpenGL biến đổi mỗi đỉnh của tất cả đối tượng trong cảnh bằng biến đổi quan sát và mô hình. Mỗi đỉnh được biến đổi sau đó được chỉ ra bởi biến đổi phép chiếu và cắt xén nếu nó nằm bên ngoài khối quan sát được mô tả bởi biến đổi phép chiếu. Cuối cùng, các đỉnh biến đổi còn lại được chia cho w và ánh xạ lên cổng nhìn.

Các lệnh biến đổi đa năng.

Phần này thảo luận một số lệnh OpenGL mà bạn có thể thấy hữu ích khi bạn chỉ ra những biến đổi mong muốn. Bạn đã vừa thấy một cặp lệnh này **glMatrixMode()** và **glLoadIdentity()**. Hai lệnh khác được miêu tả ở đây - **glLoadMatrix*()** và **glMultMatrix*()** – cho phép bạn chỉ ra trực tiếp ma trận biến đổi bất kì và sau đó để nhân ma trân hiện thời với ma trận được chỉ ra đó. Các lệnh biến đổi cụ thể hơn – như là **gluLookAt()** and **glScale*()** – được miêu tả trong những phần sau. Như đã miêu tả phần trước, bạn cần nói rõ liệu bạn muốn sửa đổi mô hình quan sát hay ma trận phép chiếu trước khi đưa ra lệnh biến đổi. Bạn chọn ma trận với **glMatrixMode()**. Khi bạn sử dụng tập hợp các lệnh OpenGL lồng nhau nhau mà có thể được gọi lặp lại, nhớ khởi động lại chế độ ma trận phù hợp. (Lệnh **glMatrixMode()** cũng có thể được sử dụng chỉ ra ma trận kết cấu; kết cấu được trình bày chi tiết trong "ngăn xếp ma trận kết cấu" trong Chương 9.)

*void **glMatrixMode**(GLenum mode);*

Chỉ ra xem ma trận mô hình quan sát, phép chiếu hay kết cấu sẽ được sửa đổi hay không,việc sử dụng đối số GL_MODELVIEW, GL_PROJECTION, hay GL_TEXTURE cho chế độ. Trình tự các lệnh biến đổi ảnh hưởng đến ma trận được chỉ ra. Chú ý rằng chỉ có một ma trận được sửa đổi tại một thời điểm. Mặc định, ma trận mô hình quan sát là một mà có thể sửa đổi, và cả ba ma trận chứa ma trận đơn vị.

Bạn có thể sử dụng lệnh **glLoadIdentity()** để xóa ma trận sửa đổi hiện tại cho các lệnh biến đổi trong tương lai, do những lệnh này sửa đổi ma trận hiện tại. Thông thường, bạn luôn gọi lệnh này trước khi

chỉ rõ phép chiếu hay biến đổi quan sát, nhưng bạn có thể gọi nó trước khi chỉ rõ một biến đổi mô hình.

void ***glLoadIdentity****(void);*

Thiết lập ma trận sửa đổi hiện thời thành ma trận 4 × 4.

Nếu bạn muốn chỉ ra một ma trận có giá trị cụ thể được nạp như ma trận hiện thời, sử dụng **glLoadMatrix*()**. Một cách tương tự, sử dụng **glMultMatrix*()** để nhân ma trận hiện thời bằng ma trận truyền như một đối đố. Đối số cho cả hai lệnh này là một vecto của 6 giá trị (m1, m2, ... , m16) chúng chỉ ra một ma trận M như sau:

$$M = \begin{bmatrix} m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \\ m_4 & m_8 & m_{12} & m_{16} \end{bmatrix}$$

Lưu ý rằng bạn có thể tăng hiệu quả tối đa bằng việc sử dụng danh sách hiển thị để lưu trữ một cách thường xuyên ma trận đã được sử dụng (và nghịch đảo của chúng) hơn là tính toán lại chúng (Xem "Thiết kế khoa học danh sách hiển thị" trong Chương 7). (Cài đặt OpenGL thường phải tính toán nghịch đảo ma trận mô hình quan sát để mặt phẳng pháp tuyến và mặt phẳng cắt có thể được biến đổi một cách chính xác đến tọa độ mắt.)

Chú ý: Nếu bạn đang lập trình trong C và bạn khai báo một ma trận là m[4][4], thì phần tử m[i][j] ở cột thứ i và hàng j của ma trận biến đổi OpenGL. Đây là sự đảo ngược trong quy định chuẩn của C với m[i][j] là hàng i và cột j. Để tránh nhầm lẫn, bạn nên khai báo ma trận của bạn là m[16].

void **glLoadMatrix***{fd}(const TYPE *m);*

Thiết lập mười sáu giá trị của ma trận hiện thời được chỉ ra bởi m.

void **glMultMatrix***{fd}(const TYPE *m);*

Nhân ma trận được xác định bởi mười sáu giá trị được trỏ bởi m với ma trận hiện thời và lưu kết quả vào ma trận hiện thời.

Chú ý: tất cả phép nhân ma trận với OpenGL xảy ra như sau: Giả sử ma trận hiện thời là C và ma trận được chỉ ra với **glMultMatrix*()** hay bất kì lệnh biến đổi nào là M. Sau phép nhân, ma trận cuối cùng luôn la CM. Do phép nhân ma trận không có tính giao hoán, thứ tự khác nhau sẽ tạo kết quả khác nhau.

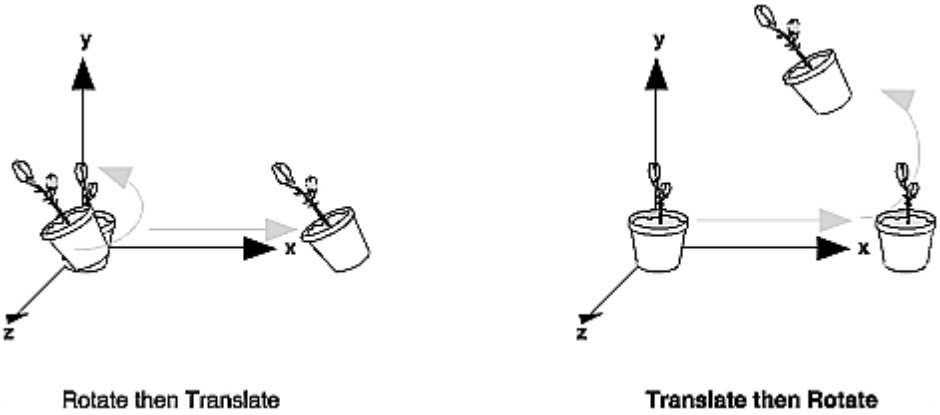
Các biến đổi quan sát và mô hình

Những biến đổi quan sát đối tượng được liên quan một cách chặt chẽ trong OpenGL và thực tế được kết hợp thành một ma trận mô hình quan sát. ()Xem "Một ví dụ đơn giản: Vẽ một hình hộp lập phương.") Một trong những vấn đề khó nhất đối với người mới học về bề mặt đồ họa máy tính là hiểu những tác động về sự kết hợp của các biến đổi ba chiều. Như bạn đã thấy, có hai cách lựa chọn để quan niệm về sự biến đổi- bạn muốn di chuyển máy quay phim theo một hướng, hay di chuyển đối tượng theo hướng ngược lại. Mỗi cách biến đổi đều có ưu và nhược điểm , nhưng trong một vài trường hợp, cách tự nhiên hơn phù hợp với kết quả biến đổi đã dự định trước. Nếu bạn có thể tìm thấy một cách tiếp cận tự nhiên cho ứng dụng cụ thể của bạn, nó dễ dàng hơn để hình dung những biến đổi cần thiết và sau đó viết mã tương ứng để xác định thao tác ma trận. Phần đầu tiên của phần này trình bày quan niệm như thế nào về các biến đổi, sau đó các lệnh cụ thể được đưa ra. Bây giờ, chúng ta chỉ sử dụng các lệnh thao tác ma trận mã bạn vừa được thấy. Cuối cùng, luôn nhớ rằng bạn

phải gọi **glMatrixMode()** với GL_MODELVIEW như đối số ưu tiên của nó để thực hiện các biến đổi mô hình hay đối tượng.

Quan niệm về các biến đổi

Hãy bắt đầu trường hợp đơn giản của hai phép biến đổi: Phép quay 45^0 quanh gốc tọa độ ngược chiều kim đồng hồ quanh trục z và một phép tịnh tiến xuống trục x. Giả sử rằng đối tượng bạn đang vẽ thì nhỏ hơn so với phép tịnh tiến (để bạn có thể thấy kết quả của phép tịnh tiến), và vị trí bắt đầu được đặt tại gốc tọa độ. Nếu bạn quay đối tượng trước và sau đó tịnh tiến, đối tượng được quay xuất hiện trên trục x. Nếu bạn tịnh tiến nó theo trục x trước, và sau đó quay quanh gốc, đối tượng trên đường y=x, như chỉ ra trong Hình 3-4. Nói chung thứ tự của phép biến đổi rất quan trọng. Nếu bạn thực hiện biến đổi A và sau đó biến đổi B, bạn hầu như luôn thu được hình ảnh khác nếu bạn thực hiện chúng trong thứ tự ngược lại.



Hình 3-4: Quay trước hay tịnh tiến trước

Bây giờ chúng ta hãy nói về thứ tự bạn chỉ ra một chuỗi biến đổi. Tất cả các biến đổi quan sát và mô hình được biểu diễn bằng ma trận 4 x 4. Mỗi **glMultMatrix*()** tiếp theo hay lệnh biến đổi nhân ma trận **M** 4 x 4 với ma trận mô hình quan sát hiện thời **C** để tạo ra **CM**. Cuối cùng các đỉnh v được nhân với ma trận mô hình quan sát hiện thời. Quá trình này nghĩa là lệnh biến đổi cuối cùng gọi trong chương trình thực sự là cái đầu tiên áp dụng với các đỉnh: **CMv.** Do đó, một cách nhìn vào nó để nói rằng bạn phải xác định ma trận theo thứ tự ngược lại. Giống như nhiều thứ khác, khi bạn quên việc sử dụng quan niệm chính xác về điều này, nhìn lại sẽ coi như là đi lên. Xét đoạn mã sau đây, chúng vẽ ra một điểm bằng việc sử dụng ba biến đổi:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glMultMatrixf(N); /* apply transformation N */
glMultMatrixf(M); /* apply transformation M */
glMultMatrixf(L); /* apply transformation L */
glBegin(GL_POINTS);
glVertex3f(v); /* draw transformed vertex v */
glEnd();
```

Với mã này, ma trận mô hình quan sát lần lượt chứa **I, N, NM** và cuối cùng **NML**, với **I** là ma trận đơn vị. Đỉnh được biến đổi là **NMLv**. Do đó, phép biến đổi đỉnh là **N(M(Lv)-** tức là, **v** đầu tiên được nhân với **L**, kết quả **Lv** là được nhân với **M**, và kết quả **MLv** được nhân với **N**. Chú ý rằng các biến đổi đến đỉnh **v** xảy ra thực sự theo thứ tự ngược lại hơn là chúng đã được chỉ ra. (Thực ra, chỉ một

phép nhân đơn của một đỉnh với ma trận mô hình quan sát xảy ra, trong ví dụ này, ma trận **N, M,** và **L** vừa được nhân trước thành ma trận đơn trước khi được áp dụng với **v.)**

Hệ tọa độ cố định, tổng quát

Do đó, nếu bạn muốn nghĩ dưới dạng hệ tọa độ cố định, tổng quát- trong đó các phép nhân ma trận ảnh hưởng tới vị trí, hướng, tỉ lệ co dãn mô hình của bạn - bạn phải nghĩ về những phép nhân theo thứ tự ngược lại từ việc chúng xuất hiện như thế nào trong mã. Việc sử dụng ví dụ đơn giản chỉ ra bên trái của Hình 3-4 (một sự phép quay quanh gốc tọa độ và tịnh tiến dọc theo trục x), nếu bạn muốn đối tượng xuất hiện trên các trục sau các phép biến đổi, phép quay phải thực hiện trước, tiếp theo là phép tịnh tiến.

Để thực hiện điều này, bạn cần đảo ngược thứ tự các phép toán, do vậy, đoạn mã trông như sau (với R là ma trận quay và T là ma trận tịnh tiến):

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glMultMatrixf(T); /* translation */
glMultMatrixf(R); /* rotation */
draw_the_object();
```

Di chuyển một hệ tọa độ cục bộ

Một cách khác để thấy các phép nhân ma trận là quên đi về hệ tọa độ cố định, tổng quát trong đó mô hình của bạn được biến đổi và thay vào đó tưởng tượng hệ tọa độ cục bộ được gắn chặt với đối tượng bạn đang vẽ. Tất cả phép biến đổi xảy ra liên quan đến sự thay đổi hệ tọa độ này. Với cách này, các phép nhân ma trận bây giờ xuất hiện theo thứ tự tự nhiên trong mã. (Bất kể bạn đang suy luận gì, mã là giống nhau, nhưng bạn quan niệm chúng khác nhau như thế nào). Để thấy điều này trong ví dụ phép tịnh tiến-quay, bắt đầu hình dung đối tượng với hệ tọa độ gắn chặt với nó. Phép tịnh tiến di chuyển đối tượng và hệ tọa độ của nó theo trục x. Sau đó, phép quay xảy ra quanh (bây giờ - tịnh tiến) gốc tọa độ, nên đối tượng quay tại vị trí của nó trên các trục. Cách tiếp cận này là những gì bạn nên sử dụng cho các ứng dụng như cánh tay robot với khớp nối, ở đó có các khớp nối vai, khuỷu tay, cổ tay và trên mỗi ngón tay. Để xác định đâu là nơi đầu ngón tay nối với với cơ thể, bạn sẽ bắt đầu tại vai, đi xuống khuỷu tay và tiếp tục như thế, áp dụng các phép quay thích hợp và tịnh tiến tại mỗi điểm nối. Nghĩ về nó theo sự đảo ngược sẽ khá rắc rối.

Cách thứ hai có thể là khá phức tạp, tuy nhiên, trong trường hợp xuất hiện việc co dãn và đặc biệt là khi co dãn không đều (co dãn khác nhau theo những trục khác nhau). Sau co dãn đều, phép tịnh tiến di chuyển một đỉnh bởi một phép nhân với những gì chúng đã làm trước đó, do hệ tọa độ được kéo dài. Co dãn không đều cùng với phép quay có thể tạo ra các trục của hệ tọa độ cục bộ không thẳng góc. Như đề cập trước đây, bạn thường đưa ra những lệnh biến đổi quan sát trong chương trình trước mọi biến đổi mô hình. Cách này, một đỉnh trong một mô hình được biến đổi đầu tiên thành hướng mong muốn và sau đó được biến đổi bởi phép tính quan sát. Khi các phép nhân ma trận phải được chỉ ra theo thứ tự đảo ngược, những lệnh quan sát cần đến trước tiên. Tuy nhiên, chú ý rằng, bạn không cần chỉ ra biến đổi hoặc quan sát hoặc đối tượng nếu bạn đã thỏa mãn với điều kiện mặc định. Nếu không có biến đổi quan sát, máy quay phim ở bên trái theo vị trí mặc định tại gốc tọa độ, nhìn về hướng âm trục z, nếu không có biến đổi mô hình, mô hình không di chuyển và nó vẫn có vị trí xác định, hướng, và kích thước. Khi những lệnh để thực hiện biến đổi mô hình có thể được sử dụng để thực hiện biến đổi quan sát, biến đổi mô hình được biến đổi trước, thậm chí nếu những biến đổi quan sát thực sự được đưa ra đầu tiên. Thứ tự cho sự thảo luận này cũng phù hợp cách mà nhiều lập trình viên nghĩ khi lập kế hoạch cho mã của họ: Thông thường, họ viết tất cả mã cần thiết để tạo ra cảnh,

chúng bao gồm các biến đổi vị trí và hướng đối tượng liên quan một cách chính xác với nhau. Tiếp theo, họ quyết định vị trí họ muốn quan sát liên quan với cảnh họ tạo ra, và sau đó họ viết biến đổi quan sát phù hợp.

Biến đổi mô hình

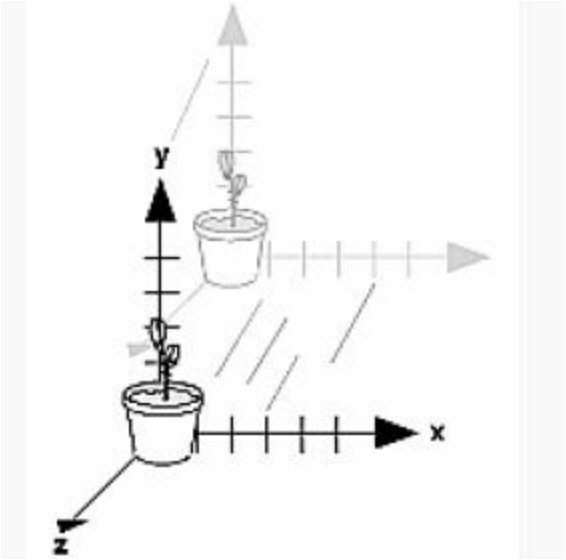
Ba thủ tục OpenGL cho biến đổi đối tượng là **glTranslate*()**, **glRotate*()**, và **glScale*()**. Vì bạn có thể nghi ngờ, các thủ tục này biến đổi một đối tượng (hay hệ tọa độ, nếu bạn nghĩ về nó theo cách đó) bằng việc di chuyển, quay, kéo dài, co ngắn, hay ánh xạ nó. Tất cả 3 lệnh này tương đương với việc tạo ra ma trận tịnh tiến, quay hay co dãn phù hợp, và sau đó gọi **glMultMatrix*()** với ma trận như đối số. Tuy nhiên, ba thủ tục này có thể nhanh hơn sử dụng **glMultMatrix*()**. OpenGL tính toán một cách tự động các ma trận cho bạn. (Xem mục lục F nếu bạn quan tâm về vấn đề này). Trong lệnh tóm tắt mà theo đó, mỗi phép nhân ma trận được mô tả dưới dạng những gì nó thực hiện với đỉnh của đối tượng hình học bằng việc sử dụng tiếp cận hệ thống tọa độ cố định và dưới dạng những gì nó thực hiện với hệ tọa độ cục bộ mà được gắn với một đối tượng.

Tịnh tiến

*void **glTranslate**{fd}(TYPEx, TYPE y, TYPEz);*

Nhân ma trận hiện thời với một ma trận mà di chuyển (tịnh tiến) một đối tượng bởi những giá trị đã cho x, y và z (hay di chuyển hệ tọa độ cục bộ bằng một lượng tương tự).

Hình3-5 chỉ ra kết quả của **glTranslate*()**.



Hình 3-5: Tịnh tiến một đối tượng.

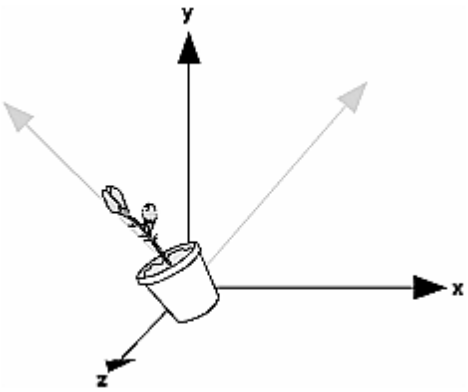
Chú ý rằng việc sử dụng (0.0, 0.0, 0.0) như đối số của **glTranslate*()** là phép tính đơn vị - tức là, nó không có ảnh hưởng đến một đối tượng hay hệ tọa độ cục bộ của nó.

Quay

*void **glRotate**{fd}(TYPE angle, TYPE x, TYPE y, TYPE z);*

Nhân ma trận hiện thời với một ma trận mà quay một đối tượng (hay hệ tọa độ cục bộ) theo hướng ngược chiều kim đồng hồ quanh một trục nối từ gốc đến điểm (x, y, z). Biến angle chỉ ra góc của phép quay theo độ.

Kết quả của **glRotatef (**45.0, 0.0, 0.0, 1.0**)**, là một phép quay 45 độ quanh trục z, được chỉ trên Hình



3-6

Hình 3-6: Quay một đối tượng.

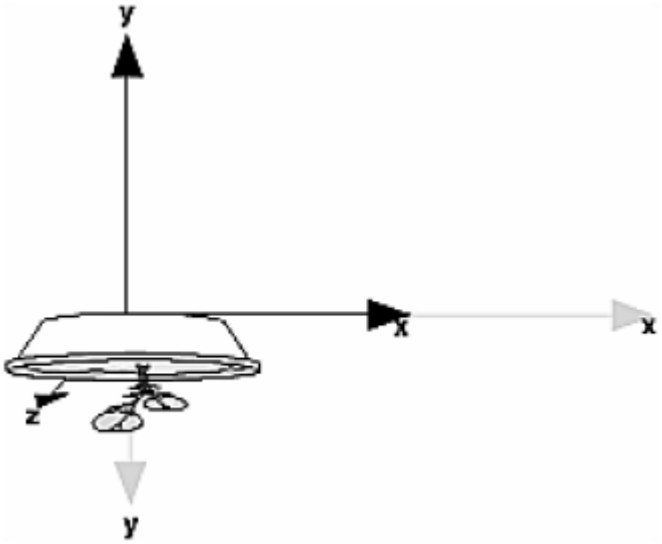
Chú ý rằng một đối tượng nằm xa trên trục quay thì quay sõ ràng hơn (quỹ đạo lớn hơn) một vật vẽ gần trục. Ngoài ra, nếu đối số góc là 0, lệnh **glRotate*()** không có tác dụng.

Co dãn

*void **glScale**{fd}(TYPEx, TYPE y, TYPEz);*

Nhân ma trận hiện thời với một ma trận mà kéo dãn, co lại, hay phản xạ một đối tượng dọc trên các trục. Mỗi tọa độ x, y và z của tất cả các điểm trong đối tượng được nhân với đối số tương ứng x, y và z. Với phương pháp hệ tọa độ cục bộ, trục tọa độ cục bộ được kéo dãn, co lại hay phản xạ bởi các yếu tốt x, y và z, và đối tượng kết hợp được biến đổi với chúng.

Hình 3-7 chỉ ra kết quả của **glScalef(**2.0, -0.5, 1.0**).**



Hình 3-7: Co dãn và phản xạ một đối tượng.

glScale*() chỉ là một trong ba biến đổi mô hình mà thay đổi kích cỡ bề ngoài của một đối tượng: Co dãn với giá trị lớn hơn 1.0 kéo dãn một đối tượng, và sử dụng các giá trị nhỏ hơn 1.0 sẽ co nó lại.

Co dãn với một giá trị -1.0 là phản xạ đối tượng qua một trục. Giá trị đơn vị cho co dãn là (1.0, 1.0,1.0). Nói chung bạn nên hạn chế việc sử dụng **glScale*()** cho trường hợp đó. Việc sử dụng **glScale*()** giảm hiệu suất của việc tính toán ánh sáng, vì vecto pháp tuyến phải chuẩn hóa lại sau phép biến đổi.

Chú ý: Một phép co dãn giá trị 0 kéo theo toàn bộ tọa độ đối tượng dọc theo trục đó là 0. Nó không phải là một ý tưởng hay cho việc thực hiện này, bởi vì một phép toán như vậy không thể được hoàn thành. Nói một cách theo toán học, ma trận không được chuyển đổi, và những ma trận ngược được yêu cầu các phép toán ánh sáng chính xác (Xem Chương 5). Tuy nhiên, đôi khi tọa độ che lấp tạo một sự nhạy cảm; Việc tính toán tô bóng trên bề mặt phẳng là một ứng dụng điển hình.(Xem "Tô bóng" trong Chương 14.) Nói chung, nếu một hệ tọa độ bị sập, nên sử dụng ma trận chiếu hơn là ma trận mô hình quan sát.

Mã ví dụ về biến đổi mô hình

Ví dụ 3-2 là một phần của chương trình tô vẽ một tam giác bốn lần, như chỉ ra trong Hình 3-8. Đây là bốn tam giác được biến đổi.

- Tam giác khung dây liền nét được vẽ mà không biến đổi mô hình.
- Tam giác tương tự được vẽ lại một lần nữa, nhưng với đường đứt nét và tịnh tiến (về phía trái, dọc theo chiều âm trục x).
- Một tam giác được vẽ với đường đứt nét kéo dài, với chiều cao của nó (trục y) bằng một nửa và chiều rộng của nó tăng (trục x) lên 50%.
- Một tam giác được quay, được tạo đường chấm, được vẽ.

Hình 3-8: Ví dụ biến đổi mô hình.

Ví dụ 3-2: Sử dụng các biến đổi mô hình : model.c

```
glLoadIdentity();
glColor3f(1.0, 1.0, 1.0);
draw_triangle(); /* solid lines */
glEnable(GL_LINE_STIPPLE); /* dashed lines */
glLineStipple(1, 0xF0F0);
glLoadIdentity();
glTranslatef(-20.0, 0.0, 0.0);
draw_triangle();
glLineStipple(1, 0xF00F); /*long dashed lines */
glLoadIdentity();
glScalef(1.5, 0.5, 1.0);
draw_triangle();
glLineStipple(1, 0x8888); /* dotted lines */
glLoadIdentity();
glRotatef (90.0, 0.0, 0.0, 1.0);
draw_triangle ();
glDisable (GL_LINE_STIPPLE);
```

Chú ý cách sử dụng của **glLoadIdentity()** để tách biệt với kết quả của các biến đổi mô hình; việc khởi tạo những giá trị ma trận ngăn cản những biến đổi liên tiếp từ việc có một kết quả tích lũy. Mặc dù vậy sử dụng **glLoadIdentity()** được lặp lại có tác dụng mong muốn, nó có thể không hiệu quả, do bạn có thể phải xác định biến đổi quan sát hay mô hình. (Xem "Thao tác với ngăn xếp ma trận") để thấy một cách hay hơn cho biến đổi tách biệt.

Chú ý: Đôi khi, người lập chương trình muốn thử kích hoạt quay đối tượng liên tục bằng việc áp dụng nhiều lần một ma trận phép quay có các giá trị nhỏ. Vấn đề của kĩ thuật này là do những lỗi được làm tròn, tích của hàng nghìn phép quay nhỏ lệch dần khỏi giá trị bạn thực sự muốn (nó có thể thậm chí một cái gì đó chứ không phải là phép quay). Thay vì việc sử dụng kĩ thuật này, tăng góc và đưa ra lệnh quay mới với góc mới tại mỗi bước cập nhật.

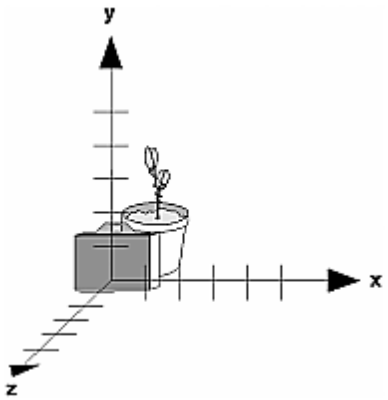
Các biến đổi quan sát

Một biến đổi quan sát thay đổi vị trí và hướng của điểm nhìn. Nếu bạn nhớ lại mô phỏng máy quay phim, biến đổi quan sát đặt vị trí tại chân máy quay phim, trỏ máy quay phim về phía mô hình. Khi bạn di chuyển máy quay phim đến một số vị trí và quay nó cho tới khi nó trỏ theo hướng mong muốn, các biến đổi quan sát nói chung bao gồm phép tịnh tiến và phép quay. Cũng nhớ rằng để đạt được bố cục cảnh nào đó trong ảnh hay ảnh chụp cuối cùng, bạn có thể hoặc di chuyển máy quay phim hoặc di chuyển tất cả đối tượng theo hướng ngược lại. Vì vậy, một biến đổi mô hình mà quay một đối tượng theo hướng ngược chiều kim đồng hồ thì tương đương với một biến đổi quan sát mà quay máy quay phim xuôi theo chiều kim đồng hồ. Cuối cùng, hãy nhớ rằng các lệnh biến đổi quan sát phải được gọi trước khi mọi biến đổi mô hình được thực hiện, sao cho các biến đổi mô hình ảnh hưởng tới đối tượng trước. Bạn có thể tạo ra một biến đổi quan sát theo cách bất kì như được miêu tả tiếp theo đây. Bạn cũng có thể chọn vị trí và hướng mặc định của điểm nhìn, chúng ở tại gốc tọa độ, nhìn về chiều âm của trục z.

- Sử dụng một hay nhiều lệnh biến đổi mô hình (đó là **glTranslate*()** và **glRotate*()**). Bạn có thể nghĩ kết quả của những biến đổi này như việc di chuyển vị trí máy quay phim hay di chuyển tất cả đối tượng trong thế giới thực, liên quan đến vị trí đứng im của máy quay phim.
- Sử dụng thủ tục **gluLookAt()** của Utility Library để xác định hướng nhìn. Thủ tục này tóm lược một chuỗi các lệnh quay và tịnh tiến.
- Tạo thủ tục tiện ích riêng của bạn mà tóm lược phép quay và phép tịnh tiến. Một vài ứng dụng có thể yêu cầu thủ tục tùy chỉnh mà cho phép bạn chỉ ra biến đổi quan sát thuận tiện. Ví dụ bạn có thể muốn chỉ ra góc quay cuộn, nghiêng, và đỉnh của một mặt phẳng trong đường bay, hay bạn có thể muốn xác định phép biến đổi theo tọa độ cực cho một máy quay phim mà quay quanh một đối tượng.

Sử dụng glTranslate*() and glRotate*()

Khi bạn sử dụng những lệnh biến đổi mô hình để mô phỏng những biến đổi quan sát, bạn đang cố gắng di chuyển điểm nhìn theo cách mong muốn trong khi giữ đối tượng đứng im. Do điểm nhìn được đặt vị trí tại gốc tọa độ và do các đối tượng thường được xây dựng phần lớn là dễ dàng ở đó (xem Hình 3-9), nói chung, bạn phải thực hiện một số phép biến đổi sao cho các đối tượng có thể được thấy. Chú ý rằng, như được chỉ ra trong hình, máy quay phim khởi tạo nhìn theo trục z âm. (Bạn đang nhìn phía sau của máy quay phim.)

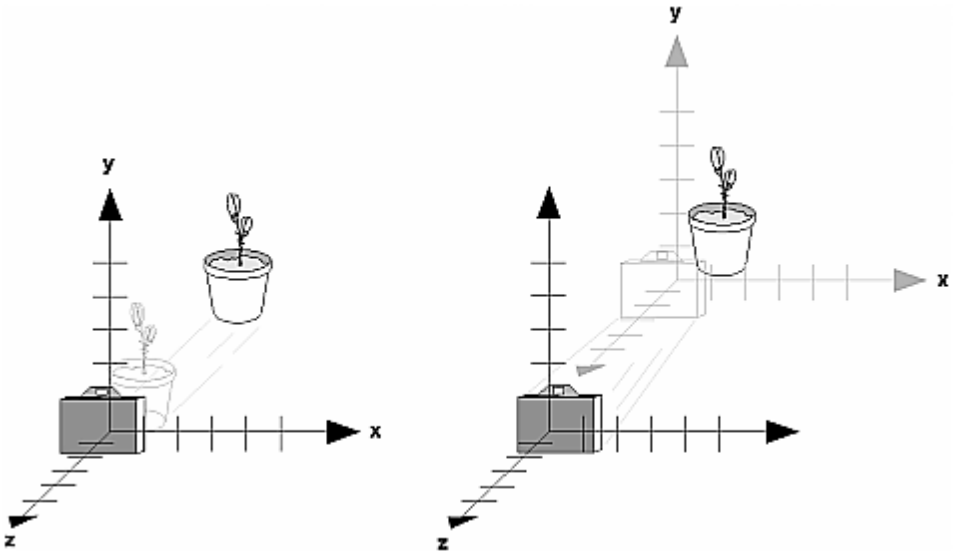


Hình 3-9: Đối tượng và điểm nhìn tại gốc tọa độ.

Trường hợp đơn giản nhất, bạn có thể di chuyển điểm nhìn ra đằng sau, xa so với đối tượng, điều này có kết quả giống như di chuyển đối tượng về phía trước, hay xa so với điểm nhìn. Nhớ rằng phía trước mặc định là trục z âm; nếu bạn quay điểm nhìn, phía trước sẽ có một ý nghĩa khác. Vì vậy, đặt khoảng cách là 5 đơn vị giữa điểm nhìn và đối tượng bằng cách di chuyển điểm nhìn, như được chỉ trong Hình 3-10, sử dụng

```
glTranslatef (0.0,0.0,-5.0);
```

Thủ tục này di chuyển các đối tượng trong cảnh là -5 đơn vị dọc theo trục z. Điều này cũng tương đương với di chuyển máy quay phim +5 đơn vị dọc theo trục z.



Hình 3-10: Phân biệt điểm nhìn và đối tượng

Bây giờ giả sử bạn muốn quan sát những đối tượng từ một phía. Bạn nên đưa ra lệnh quay trước hay sau lệnh tịnh tiến? Nếu bạn đang sử dụng hệ tọa độ cố định, tổng quát, đầu tiên tưởng tượng cả hai đối tượng và máy quay phim ở gốc tọa độ. Bạn có thể quay đối tượng trước và sau đó di chuyển nó ra xa máy quay phim sao cho mặt mong muốn được nhìn thấy. Vì bạn biết rằng với phương pháp hệ tọa độ cố định, các lệnh phải được đưa ra theo thứ tự ngược lại mà chúng mang theo kết quả, bạn biết rằng bạn cần viết mã lệnh tịnh tiến trước và tiếp theo là lệnh quay. Bây giờ, chúng ta hãy sử dụng phương pháp hệ tọa độ cục bộ. Trong trường hợp này, tập trung vào việc di chuyển đối tượng và hệ

tọa độ cục bộ của nó xa so với gốc tọa độ; sau đó, lệnh quay được thực hiện bằng việc sử dụng hệ tọa độ tịnh tiến ngay lập tức. Với cách này, những lệnh được đưa ra theo thứ tự chúng được áp dụng, nên mỗi lần lặp lại, lệnh tịnh tiến thực hiện trước. Do đó, chuỗi các lệnh biến đổi để tạo ra kết quả mong muốn là

```
glTranslatef(0.0, 0.0, -5.0);
glRotatef(90.0, 0.0, 1.0, 0.0);
```

Nếu bạn đang lo lắng việc duy trì chuỗi kết quả của phép nhân ma trận liên tiếp, thử sử dụng cả hai phương pháp hệ tọa độ cố định và cục bộ và thấy cái nào tạo nhiều ý nghĩa hơn cho bạn. Chú ý rằng với hệ tọa độ cố định, phép quay luôn xảy ra quanh gốc tạo tọa độ chính, trong khi với hệ tọa độ cục bộ, phép quay xảy ra gốc tọa độ của hệ tọa độ cục bộ. Bạn có thể thử sử dụng thủ tục tiện ích **gluLookAt()** được miêu tả trong phần tiếp theo.

Sử dụng thủ tục tiện ích gluLookAt()

Thông thường, lập trình viên xây dựng một cảnh quanh gốc tọa độ hay một số vị trí thuận tiện khác, sau đó họ muốn quan sát nó từ một điểm tùy ý để có một góc nhìn tốt. Đúng như tên gọi, thủ tục tiện ích **gluLookAt()**được thiết kế chỉ cho mục đích này. Nó đưa ra ba bộ đối số, chúng chỉ ra vị trí của điểm nhìn, định nghĩa điểm mà máy quay phim nhắm tới, và chỉ ra hướng đỉnh. Chọn điểm nhìn để tạo ra khung cảnh mong muốn. Điểm chiếu này thường là một nơi nào đó ở giữa cảnh. (Nếu bạn đã xây dựng cảnh của bạn tại gốc tọa độ, điểm chiếu có thể là gốc tọa độ). Có thể phải dùng một chút tiểu xảo để xác định chính xác vecto hướng đỉnh. Ngoài ra, nếu bạn đã xây dựng một số cảnh thế giới thực tại hoặc xung quanh gốc tọa độ và nếu bạn lấy trục y dương để chỉ hướng đỉnh, thì đó chính là vecto hướng đỉnh của bạn trong thủ tục **gluLookAt()**. Tuy nhiên nếu bạn đang thiết kế một mô phỏng đường bay, hướng đỉnh là hướng vuông góc với cánh máy bay, từ máy bay về phía bầu trời khi máy bay ở phía trên bên phải của mặt đất.

Thủ tục **gluLookAt()** là đặc biệt hữu ích khi bạn quay một cảnh quan, Ví dụ. Với một khối quan sát đối xứng cả x và y, điểm xác định (eyex, eyey, eyez) điểm luôn ở trung tâm ảnh trên màn hình, vì vậy bạn có thể sử dụng một chuỗi lệnh để di chuyển điểm này một lượng không đáng kể, bằng cách xoay toàn bộ cảnh.

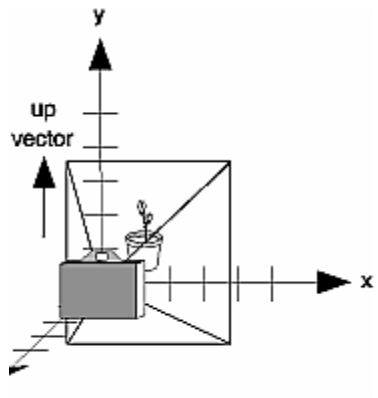
void gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centerx, GLdouble centery, GLdouble centerz, GLdouble upx, GLdouble upy, GLdouble upz);

Định nghĩa một ma trận quan sát và nhân nó với ma trận hiện thời. Điểm nhìn mong muốn được chỉ ra bởi eyex, eyey, và eyez. Các đối số centerx, centery, và centerz chỉ ra điểm bất kì nằm trên hướng nhìn mong muốn, nhưng thông thường chúng là một số điểm ở trung tâm của cảnh đang được quan sát. Các đối số upx, upy, và upz chỉ ra hướng nào là ở trên (tức là, hướng từ đáy đến đỉnh của khối quan sát)

Vị trí mặc định, máy quay phim ở tại gốc tọa độ, nhìn xuống trục z âm, và có trục y dương là hướng đỉnh. Điều này tương tự như việc gọi

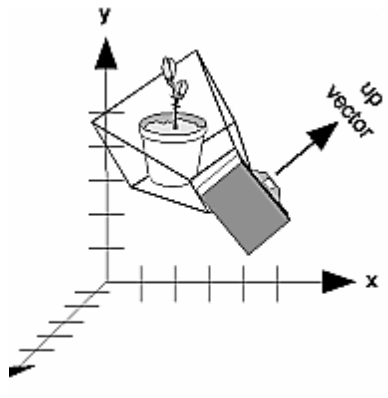
gluLookat (0.0, 0.0, 0.0, 0.0, 0.0, -100.0, 0.0, 1.0, 0.0);

Giá trị z của điểm hướng tới là -100.0, nhưng có thể là một giá trị z âm bất kì, vì hướng nhìn vẫn không đổi. Trong trường hợp này, bạn thực sự không muốn gọi **gluLookAt()**, vì chúng là mặc định (xem Hình 3-11) và bạn đã ở đây! (hướng mở rộng từ máy quay phim đưa ra khối quan sát, chúng chỉ ra phạm vi quan sát của nó.)



Hình 3-11: Vị trí mặc định của máy quay phim.

Hình 3-12 chỉ ra kết quả của thủ tục điển hình **gluLookAt()**. Vị trí máy quay phim(eyex, eyey, eyez) là (4, 2, 1). Trong trường hợp này, máy quay phim đang nhìn thẳng vào mô hình, nên điểm nhìn là (2, 4, -3). Một vecto định hướng (2,2,-1) được chọn để quay điểm nhìn một góc 45 độ.



Hình 3-12: Sử dung gluLookAt() vì vậy để đạt được kết quả này gọi

```
gluLookAt(4.0, 2.0, 1.0, 2.0, 4.0, -3.0, 2.0, 2.0, -1.0);
```

Chú ý rằng **gluLookAt()** là phần của thư viện tiện ích (Utility Library) hơn là thư viện OpenGL cơ bản. Đây không phải bởi vì nó không hữu ích, mà bởi vì nó gói gọn một số lệnh OpenGl cơ bản- đặc biệt **glTranslate*()** và **glRotate*()**. Để thấy điều này, tưởng tượng một máy quay phim đặt tại điểm nhìn tùy ý và theo hướng nhìn, cả hai được xác định với **gluLookAt()** và một cảnh được đặt tại gốc tọa độ. Để quay trở về những gì mà **gluLookAt()** thực hiện, bạn cần biến đổi máy quay phim sao cho nó ở gốc tọa độ và nhìn về trục z âm, vị trí mặc định. Một tịnh tiến đơn giản di chuyển máy quay phim về gốc tọa độ.

Bạn có thể tưởng tượng dễ dàng một chuỗi những phép quay quanh một trong 3 trục của hệ tọa độ cố định mà sẽ hướng máy quay phim sao cho nó hướng về các giá trị z âm.Vì OpenGL cho phép quay quanh trục bất kì, bạn có thể hoàn thành mọi phép quay mong muốn của máy quay phim với lệnh đơn **glRotate*()**.

Chú ý: Bạn có thể có duy nhất một hoạt động biến đổi quan sát. Bạn không thể cố kết hợp kết quả của hai phép biến đổi quan sát, như một máy quay phim có thể có hai chân đỡ. Nếu bạn muốn thay đổi vị trí máy quay phim, bạn hãy gọi **glLoadIdentity()** để chắc chắn xóa đi mọi kết quả còn lưu lại của biến đổi quan sát hiện thời.

Nâng cao

Để biến đổi vecto tùy ý sao cho nó trùng khớp với một vecto tùy ý khác (Ví dụ, trục z âm) bạn cần mất một chút tính toán. Trục mà bạn muốn quay được cho bởi tích có hướng của hai vecto chuẩn hóa. Để tìm ra góc của phép quay, chuẩn hóa hai vecto ban đầu. Cos của góc mong muốn giữa các vecto là tích vô hướng của các vecto chuẩn hóa. Góc quay quanh trục được cho bởi tích có hướng luôn trong khoảng 0 và 180 độ. (Xem Mục lục E về định nghĩa tích có hướng và vô hướng)
Chú ý rằng việc tính toán góc giữa 2 vecto chuẩn hóa bằng cách lấy cos ngược của tích vô hướng không chính xác lắm, đặc biệt cho những góc nhỏ. Nhưng nó sẽ hoạt động đủ tốt để bạn bắt đầu.

Việc tạo một thủ tục tiện tích tùy chọn

Nâng cao

Đối với một vài ứng dụng đặc biệt, bạn có thể muốn xác định thủ tục biến đổi riêng của bạn. Do điều này hiếm khi được thực hiện và trong mọi trường hợp là chủ đề nâng cao, nó hầu như bị bỏ lại như một bài tập cho người đọc. Những bài tập sau gợi ý hai phép biến đổi quan sát tùy chọn mà có thể hữu ích.

Thử điều này

Giả sử rằng bạn đang viết một mô hình hóa máy bay và bạn muốn hiển thị thế giới ở điểm nhìn của phi công. Thế giới đó được miêu tả trong hệ tọa độ với gốc tọa độ trên đường chạy và máy bay ở tọa độ (x, y, z). Giả sử xa hơn rằng máy bay có lộn vòng, lượn. (có một số góc quay của máy bay liên quan đến trọng tâm của nó).

Chỉ ra rằng thủ tục sau có thể coi như biến đổi quan sát:

```
void pilotView{GLdouble planex, GLdouble planey,
GLdouble planez, GLdouble roll,
GLdouble pitch, GLdouble heading)
{
glRotated(roll, 0.0, 0.0, 1.0);
glRotated(pitch, 0.0, 1.0, 0.0);
glRotated(heading, 1.0, 0.0, 0.0);
glTranslated(-planex, -planey, -planez);
}
```

Giả sử rằng ứng dụng của bạn bao gồm phạm vi hoat động của máy quay phim quanh một đối tượng mà tâm của nó đặt tại gốc tọa độ. Trong trường hợp này, bạn muốn chỉ ra biến đổi quan sát bằng sử dụng tọa độ cực.

Hãy để biến khoảng cách xác định bán kính quỹ đạo hay khoảng cách từ máy quay phim đến gốc tọa độ là bao xa (khởi tạo, máy quay phim được di chuyển khoảng cách đơn vị theo trục z dương). Góc phương vị miêu tả góc quay của máy quay phim quanh đối tượng trong mặt phẳng x-y, đo từ trục y dương. Tương tự, phép chiếu thẳng góc là góc quay của máy quay phim trong máy bay y-z, đo từ trục z dương. Cuối cùng vòng xoắn tượng trưng phép quay của khối quan sát quanh trục của hướng nhìn.

Chỉ ra rằng thủ tục sau có thể coi như biến đổi quan sát.

```
void polarView{GLdouble distance, GLdouble twist,GLdouble
elevation, GLdouble azimuth)
{
glTranslated(0.0, 0.0, -distance);
glRotated(-twist, 0.0, 0.0, 1.0);
glRotated(-elevation, 1.0, 0.0, 0.0);
```

```
glRotated(azimuth, 0.0, 0.0, 1.0);
}
```

Biến đổi phép chiếu

Những phần trước miêu tả cách để tạo ra ma trận mô hình quan sát mong muốn sao cho những biến đổi mô hình và quan sát chính xác được áp dụng. Phần này giải thích cách để xác định ma trận chiếu mong muốn, chúng cũng được sử dụng để biến đổi các đỉnh trong cảnh của bạn. Trước khi bạn đưa ra bất kì lệnh biến đổi mong muốn nào trong phần này, nhớ gọi

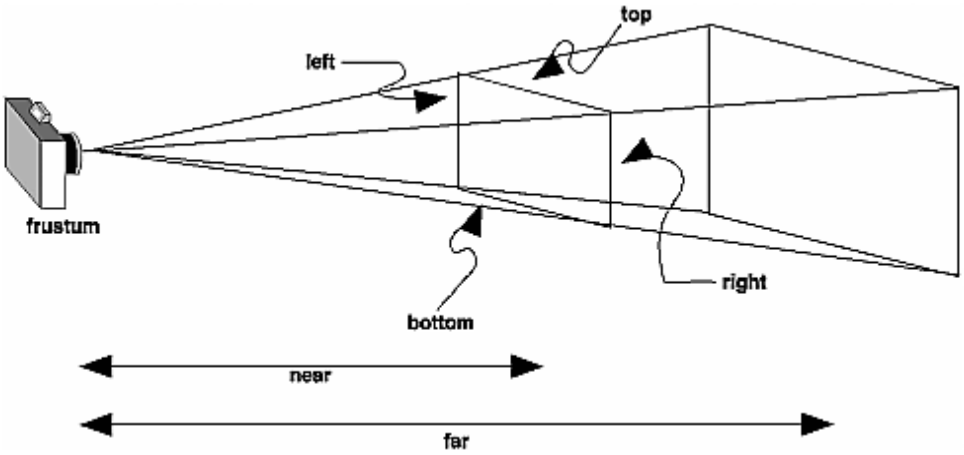
```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
```

sao cho lệnh đó ảnh hưởng ma trận chiếu hơn ma trận mô hình quan sát và để bạn tránh được những biến đổi phép chiếu kép. Do mỗi lệnh biến đổi phép chiếu mô tả hoàn chỉnh một biến đổi cụ thể, thông thường, bạn không muốn kết hợp một biến đổi phép chiếu với một biến đổi khác. Mục đích của sự biến đổi phép chiếu là để xác định *khối quan sát,* chúng được sử dụng theo hai cách. Khối quan sát quyết định một đối tượng được chiếu như thế nào trên màn hình (tức là, bằng việc sử dụng một phép chiếu phối cảnh hay trực giao), và nó xác định những đối tượng hay những phần của đối tượng bị cắt khỏi ảnh cuối cùng. Bạn có thể nghĩ điểm nhìn chúng ta đã nói đến nhìn đến phần cuối của khối quan sát. Tại điểm này bạn có thể muốn đọc lại "Một ví dụ đơn giản: vẽ một hình lập phương" với cái nhìn tổng quan tất cả các phép biến đổi, bao gồm biến đổi hình chiếu.

Phép chiếu phối cảnh

Đặc điểm dễ phân biệt nhất của phép chiếu phối cảnh là việc co ngắn: một đối tượng càng xa máy quay phim, thì hình ảnh của nó càng nhỏ hơn trong ảnh cuối. Điều này xảy ra bởi vì khối quan sát cho một phép chiếu phối cảnh là của một hình chóp cụt (một hình chóp cụt mà đỉnh của nó bị cắt bỏ bởi một mặt phẳng song song với đáy của nó). Những đối tượng mà nằm trong khối quan sát được chiếu thẳng tới đỉnh của hình chóp, nơi đặt máy quay phim hoặc điểm nhìn. Những đối tượng ở gần điểm nhìn sẽ xuất hiện lớn hơn bởi vì chúng chiếm tỷ lệ số lượng lớn của khối quan sát hơn là những đối tượng ở xa, trong phần lớn hơn của hình chóp. Phương pháp chiếu theo cách này thường được sử dụng cho hoạt cảnh, và tất cả ứng dụng khác mà nhìn như trong thực tế bởi vì nó giống như cách nhìn của mắt người (hoặc máy ảnh).

Lệnh để định nghĩa một hình chóp cụt, **glFrustum(),** tính toán một ma trận mà nó thực hiện phép chiếu phối cảnh và nhân ma trận phép chiếu hiện thời (thường là ma trận đơn vị) với nó. Nhắc lại rằng khối quan sát được sử dụng để cắt những đối tượng mà nằm bên ngoài nó, bốn mặt của hình chóp, phía trên của nó, và phần đáy tương ứng với sáu mặt phẳng cắt của khối quan sát, như đã chỉ ra trong Hình 3-13. Những đối tượng hay những phần của đối tượng bên ngoài mặt phẳng này được cắt rất khỏi ảnh cuối. Lưu ý rằn **glFrustum()** không yêu cầu bạn phải định nghĩa khối quan sát cân đối.



Hình 3-13 : Khối quan sát phối cảnh được chỉ ra bởi glFrustum()

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

Tạo một ma trận cho hình chóp cụt quan sát phối cảnh và nhân ma trận hiện thời với nó. Khối quan sát của hình chóp cụt được định nghĩa bởi các biến: left, bottom, -near) và (right, top, -near) chỉ ra tọa độ (x, y, z) của góc dưới bên trái và góc trên bên phải của mặt phẳng cắt gần, near và far cho khoảng cách từ điểm nhìn đến mặt phẳng gần và xa. Chúng nên luôn luôn là dương.

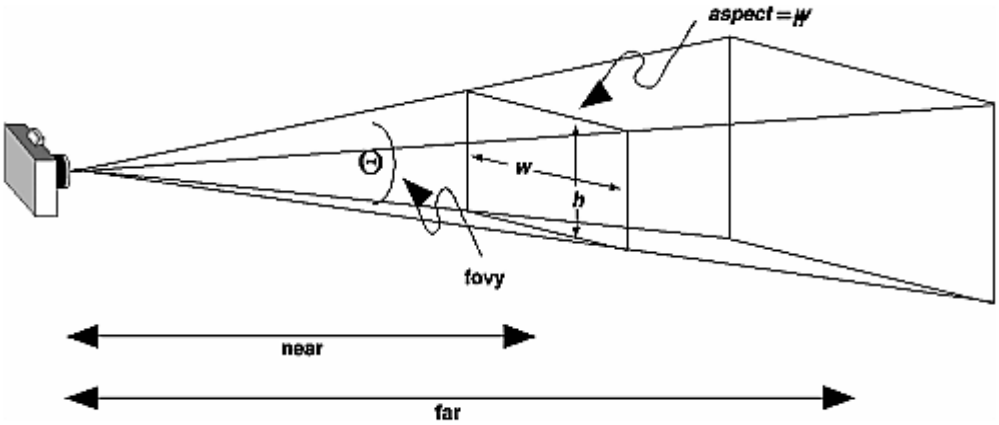
Hình chóp cụt một hướng ngầm định trong không gian ba chiều. Bạn có thể thực hiện pehps quay hay tịnh tiến trên ma trận phép chiếu để thay đổi hướng này, nhưng đây là một thủ thuật và hầu như có thể luôn tránh được.

Nâng cao

Ngoài ra, hình chóp cụt không có tính đối xứng, và trục của nó không cần thiết phải thẳng hàng với trục z

Ví dụ, như bạn có thể sử dụng **glFrustum()** để vẽ một bức tranh nếu như bạn đang nhìn qua một cửa sổ của ngôi nhà. Nơi mà cửa sổ ở phía trên và phía bên phải bạn. Nhà nhiếp ảnh sử dụng như một khối quan sát như thế để tạo ra những phối cảnh giả. Bạn có thể dùng nó để tính toán chính xác hình ảnh với độ phân giải cao hơn thông thường, có thể cho sử dụng trên máy in. Ví dụ, nếu bạn muốn một ảnh có độ phân giải gấp hai lần màn hình của bạn, vẽ bốn lần bức tranh giống nhau, mỗi lần bằng việc sử dụng một hình chóp cụt để bao phủ toàn bộ màn hình với một phần tư hình ảnh. Sau mỗi một phần tư ảnh được tô vẽ, bạn có thể đọc các điểm ảnh sau một tập hợp dữ liệu về ảnh có độ phân giải cao. (Xem Chương 8 để biết thêm thông tin về việc đọc dữ liệu điểm ảnh)

Mặc dù nó là những khái niệm dễ hiểu, **glFrustum()** không phải sử dụng một cách trực giác.Thay vào đó, bạn có thể sử thử thủ tục tiện ích **gluPerspective()**. Thủ tục này tạo ra một khối quan sát cùng dạng với **glFrustum()**, nhưng bạn chỉ ra theo cách khác. Thay vì chỉ ra các góc của mặt phẳng cắt gần, bạn sẽ chỉ ra góc của phạm vi quan sát (&THgr; hay theta, trong Hình 3-14) theo hướng của y và tỷ lệ co của chiều rộng so với chiều cao (x/y). (Với một đơn vị vuông của màn hình, tỉ lệ co là 1.0) .Hai tham số này đủ để xác định một hình chóp cụt không cắt dọc theo hướng nhìn, như đã chỉ ra trong Hình 3-14. Bạn cũng có thể chỉ ra khoảng cách giữa điểm nhìn với mặt phẳng cắt gần và xa, bằng cách cắt hình chóp. Lưu ý rằng **gluPerspective()** bị hạn chế để tạo ra hình chóp cụt mà đối xứng theo cả hai trục x và y dọc theo hướng nhìn. Nhưng đây luôn là những gì bạn muốn.



Hình 3-14 : Khối quan sát phối cảnh chỉ ra bởi gluPerspective()

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

Tạo một ma trận cho một hình cụt quan sát phối cảnh đối xứng và nhân ma trận hiện thời với nó. fovy là góc của phạm vi quan sát trong mặt phẳng x-z, giá trị của nó phải trong miền [0.0,180.0]. aspect là tỉ lệ co của hình cụt, độ rộng của nó được chia cho chiều cao của nó, giá trị near và far là khoảng cách giữa điểm nhìn và các mặt phẳng cắt, dọc theo trục z âm. Chúng nên luôn là giá trị dương.

Chỉ với **glFrustum()**, bạn có thể áp dụng phép quay hay tịnh tiến để thay đổi hướng ngầm định của khối quan sát được tạo bởi **gluPerspective()**. Khi không có biến đổi đó, điểm nhìn vẫn ở gốc tọa độ, và hướng nhìn về phía trục z âm.

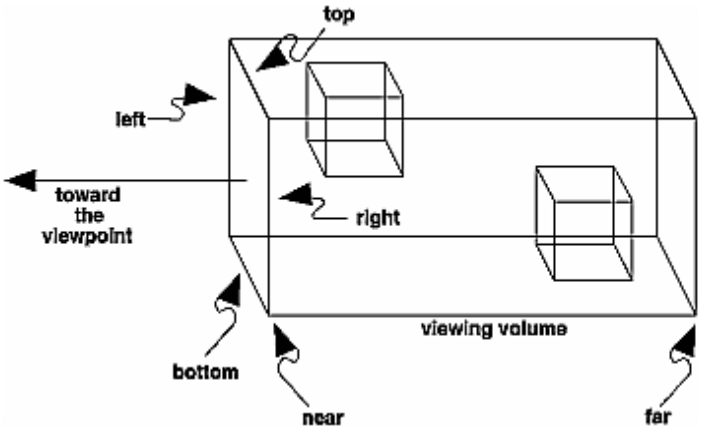
Với **gluPerspective(),** bạn cần chọn những giá trị phù hợp với phạm vi quan sát hay hình ảnh có thể bị méo. Ví dụ, giả sử bạn đang vẽ toàn bộ màn hình, chúng có độ cao 11 inches. Nếu bạn chọn một phạm vi quan sát 90 độ, mắt của bạn phải cách 7,8 inches tính từ màn hình để ảnh xuất hiện không méo. (Đây là khoảng cách mà tạo màn hình góc 90 độ). Nếu mắt bạn ở xa màn hình, như thông thường, thì phối cảnh nhìn sẽ không chính xác. Nếu vùng vẽ của bạn vẽ chiếm diện tích nhỏ hơn toàn bộ màn hình, thì mắt bạn phải gần ảnh hơn. Để có một phạm vi quan sát hoàn hảo thì phải chỉ ra khoảng cách thông thường từ mắt bạn đến màn hình và kích cỡ của cửa sổ, và tính toán góc của của sổ đối diện với kích cỡ và khoảng cách đó. Nó có thể nhỏ hơn bạn suy đoán. Một cách khác để suy nghĩ về phạm vi quan sát khoảng 94 độ với một máy quáy phim 35mm yêu cầu ống kính 20mm, chúng là những ống kính rất rộng. (Xem "Biến đổi gỡ rối" để biết thêm chi tiết về làm thế nào để tính toán phạm vi quan sát mong muốn)

Trong phần trước đề cập inches và mm – chúng có thực sự liên quan đến với OpenGL không? câu trả lời là không. Biến đổi phép chiếu và các biến đổi khác vốn không thuộc về đơn vị. Nếu bạn muốn nghĩ các mặt phẳng cắt gần và xa được đặt tại 1.0 và 20.0 mét, inch, km, hay hải lý, nó là tùy bạn. Quy tắc duy nhất là bạn bạn sử dụng đơn vị quy ước đo đạc. Sau đó ảnh kết quả được vẽ theo tỉ lệ.

Phép chiếu trực giao

Với một hình chiều trực giao, khối quan sát là một hình hộp chữ nhật hay còn gọi là hình hộp (xem Hình 3-15). Không giống như phép chiếu phối cảnh, kích cỡ của khối quan sát không thay đổi từ một điểm này tới một điểm khác, nên khoảng cách từ máy quay phim không ảnh hưởng tới độ phóng to của hình ảnh đối tượng. Kiểu phép chiếu này được sử dụng cho các ứng dụng như thiết kế kiến trúc

hoặc thiết kế với sự trợ giúp của máy vi tính, nơi nó quyết định giữ lại kích cỡ thực tế của đối tượng vẫn góc giữa chúng khi được chiếu.



Hình 3-15 : Khối quan sát của phép chiếu trực giao

Lệnh **glOrtho()** tạo ra một phép chiếu trực giao song song với khối quan sát. Như với **glFrustum()**, bạn sẽ chỉ ra những góc của mặt phẳng cắt gần và khoảng cách tới mặt phẳng cắt xa.

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

Tạo một ma trận cho một phép chiếu song song khối quan sát và nhân ma trận hiện thời với nó (left, bottom, -near) và (right, top, -near) được trỏ tới mặt phẳng cắt gần mà được ánh xạ đến góc dưới bên trái và góc trên bên phải của cổng nhìn cửa sổ, một cách tương ứng (left,bottom, -far) và (right, top, -far) trỏ tới mặt phẳng cắt xa mà ánh xạ đến góc tương ứng của cổng nhìn. Cả hai near và far có thể là dương hoặc âm.

Nếu không có những biến đổi khác thì hướng của phép chiếu song song với trục z, và điểm nhìn hướng về trục z âm. Lưu ý rằng điều này có nghĩa là những giá trị truyền cho far và near được sử dụng như các giá trị z âm nếu các mặt phẳng này đối diện với điểm nhìn, và dương nếu chúng đằng sau điểm nhìn.

Trong trường hợp đặc biệt phép chiếu một ảnh hai chiều lên màn hình 2 chiều, sử dụng thủ tục tiện ích **gluOrtho2D()**. Thủ tục này giống như trong phiên bản ba chiều, **glOrtho(),** ngoại trừ tất cả tọa độ z của các đối tượng trong cảnh được giả sử nằm giữa -1.0 và 1.0. Nếu bạn đang vẽ đối tượng hai chiều bằng việc sử dụng lệnh của đỉnh hai chiều, tất cả tọa độ z là 0; Do đó, không có đối tượng nào bị cắt bởi các giá trị z của chúng.

*void **gluOrtho2D**(GLdouble left, GLdouble right,*
GLdouble bottom, GLdouble top);

Tạo một ma trận đối với phép chiếu các tọa độ hai chiều trên màn hình và nhân ma trận phép chiếu hiện thời với nó. Vùng cắt xen là một hình chữ nhật với góc dưới bên trái là (left, bottom) và góc trên bên phải là (right, top).

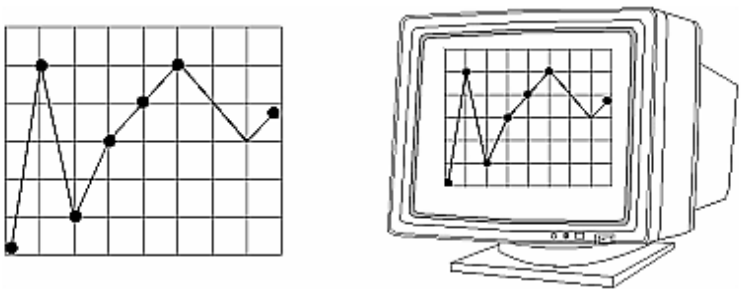
Cắt xén khối quan sát

Sau các đỉnh của đối tượng trong ảnh được biến đổi bởi các ma trận mô hình quan sát và phép chiếu, mọi nguyên thủy mà nằm bên ngoài khối quan sát bị cắt xen. Sáu mặt phẳng được sử dụng được định nghĩa là các mặt và kết thúc của khối quan sát. Bạn có thể chỉ ra thêm các mặt phẳng cắt và xác định vị trí chúng ở bất kì nơi nào bạn chọn. (Xem "Thêm các mặt phẳng cắt" để biết thêm thông tin về chủ

đề nâng cao có liên quan này). Luôn nhớ rằng OpenGL khôi phục lại các cạnh của đa giác mà chúng được cắt.

Biến đổi cổng nhìn

Nhắc lại phần mô phỏng tựa máy quay phim, bạn biết rằng biến đổi quan sát tương ứng với giai đoạn mà kích thước ảnh khai thác được lựa chọn. Bạn muốn có một bức ảnh kích thước nhỏ hay kích thước khổ áp phích quảng cáo? Do đây là đồ họa máy tính, cổng nhìn là vùng cửa sổ hình chữ nhật mà tại đó ảnh được vẽ. Hình 3-16 chỉ ra một cổng nhìn chiếm gần hết màn hình. Cổng nhìn được đo theo tọa độ của cửa sổ, chúng phản ánh vị trí của điểm ảnh trên màn hình liên quan đến góc dưới bên trái của cửa sổ. Hãy nhớ rằng tất cả các đỉnh đã được biến đổi bởi các ma trận mô hình quan sát và phép chiếu theo điểm này, và đỉnh bên ngoài khối quan sát đã được cắt bớt.



Hình 3-16 : Cổng nhìn hình chữ nhật.

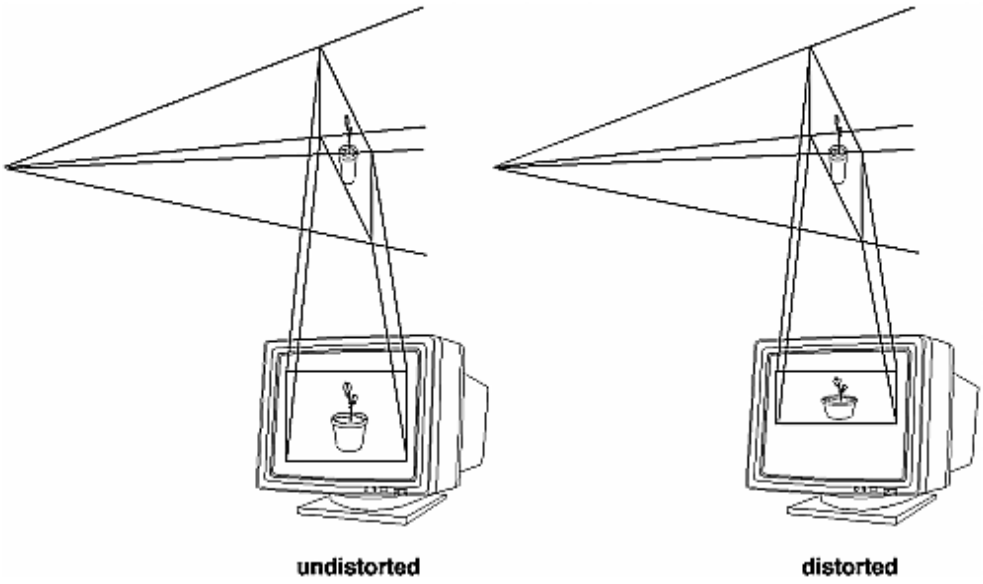
Xác định cổng nhìn

Hệ thống cửa sổ, không phải OpenGL, chịu trách nhiệm mở một cửa sổ trên màn hình. Tuy nhiên, mặc định cổng nhìn được thiết lập là toàn bộ điểm ảnh hình chữ nhật của cửa sổ đã được mở. Bạn sử dụng lệnh **glViewport ()** để chọn một vùng vẽ nhỏ hơn; Ví dụ, bạn có thể chia nhỏ cửa sổ để tạo ra một hiệu ứng phân chia màn hình cho nhiều quan sát trong cùng một cửa sổ.

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

Định nghĩa một điểm ảnh hình chữ nhật trong cửa sổ vào ảnh cuối cùng được ánh xạ. Biến (x, y) chỉ ra góc dưới bên trái của cổng nhìn, và width và height là kích cỡ của cổng nhìn hình chữ nhật. Ngầm định, các giá trị khởi tạo cổng nhìn là (0, 0, winWidth, winHeight). Với winWidth và winHeight là kích cỡ cửa sổ.

Các tỷ lệ co của một cổng nhìn nói chung nên bằng với tỷ lệ co của khối quan sát. Nếu hai tỷ lệ này khác nhau thì hình ảnh được chiếu sẽ bị méo khi được ánh xạ tới cổng nhìn, như thể hiện trong Hình 3-17. Lưu ý rằng những thay đổi tiếp theo với kích thước của cửa sổ không ảnh hưởng rõ ràng đến cổng nhìn. Ứng dụng của bạn sẽ dò các sự kiện thay đổi kích thước cửa sổ và sửa đổi cổng nhìn thích hợp.



Hình 3-17 : Ánh xạ khối quan sát đến cổng nhìn

Trong Hình 3-17, hình bên trái chỉ ra một phép chiếu mà ánh xạ một ảnh vuông thành một cổng nhìn vuông bằng việc sử dụng những thủ tục sau :

```
gluPerspective(fovy, 1.0, near, far);
glViewport(0, 0, 400, 400);
```

Tuy nhiên, trong hình bên phải, cửa sổ đã được thay đổi kích cỡ thành một cổng nhìn hình chữ nhật có cạnh không đều, nhưng phép chiếu là không thay đổi. Ảnh xuất hiện bị nén dọc theo trục x.

```
gluPerspective(fovy, 1.0, near, far);
glViewport (0, 0, 400, 200);
```

Để tránh những biến dạng, sửa đổi tỉ lệ co của các phép chiếu cho phù hợp với cổng nhìn:

```
gluPerspective(fovy, 2.0, near, far);
glViewport(0, 0, 400, 200);
```

Thử điều này

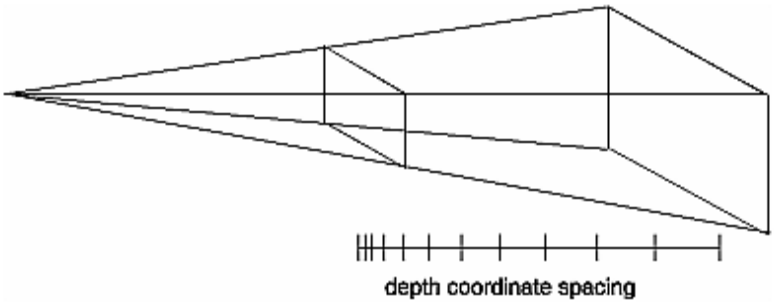
Sửa đổi một chương trình đã có để một đối tượng được vẽ hai lần, với các cổng nhìn khác nhau. Bạn có thể vẽ đối tượng với phép chiếu khác nhau và/hoặc các biến đổi quan sát cho mỗi cổng nhìn. Để tạo ra hai cổng nhìn cùng phía, bạn có thể đưa ra các lệnh này, cùng với các biến đổi mô hình, quan sát, và phép chiếu thích hợp:

```
glViewport (0, 0, sizex/2, sizey);
                    .
                    .
                    .
glViewport (sizex/2, 0, sizex/2, sizey);
```

Tọa độ chiều sâu được biến đổi

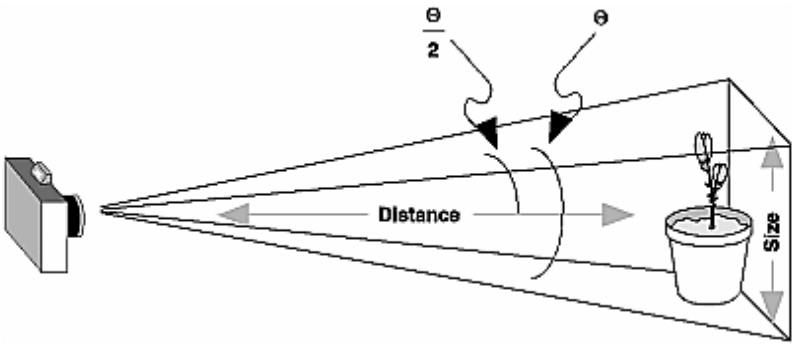
Tọa độ (z) được mã hoá trong khi biến đổi cổng nhìn (và sau đó được lưu trữ trong bộ đệm chiều sâu). Bạn có thể co dãn các giá trị z trong miền mong muốn với lệnh **glDepthRange ()**. (Chương 10 thảo luận về bộ đệm chiều sâu và sử dụng tương ứng với chiều sâu phối hợp). Không giống như tọa độ cửa sổ x và y, OpenGL xét tọa độ cửa sổ z dường như trong miền
0.0 đến 1.0.

*void **glDepthRange**(GLclampd near, GLclampd far);*
Định nghĩa một mã hóa cho tọa độ z mà được thực hiện trong khi biến đổi cổng nhìn. Các giá trị near và far mô tả điều chỉnh các giá trị nhỏ nhất và lớn nhất mà có thể được chứa trong vùng đệm chiều sâu. Ngầm định, chúng là 0.0 và 1.0, một cách tương ứng, chúng làm việc cho hầu hết các ứng dụng. Các biến này được dao động trong phạm vi [0,1].
Trong phép chiếu phối cảnh, sự biến đổi tọa độ chiều sâu (như tọa độ *x* và *y*) là chủ đề phân loại phối cảnh theo tọa độ *w*. Khi biến đổi tọa độ chiều sâu di chuyển càng xa mặt phẳng cắt gần, vị trí của nó sẽ càng thiếu chính xác. (Xem Hình 3-18)



Hình 3-18 : Hình chiều phối cảnh và biến đổi tọa độ chiều sâu.
Bởi vậy, phân loại phối cảnh ảnh hưởng tới độ chính xác của các hoạt động, mà chúng dựa trên sự biến đổi tọa độ chiều sâu, đặc biệt là bộ đệm - chiều sâu, cái mà được sử dụng cho việc khử mặt khuất.
Các biến đổi gỡ rối
Thật quá dễ dàng để có được một máy quay phim đặt đúng hướng, nhưng trong đồ họa máy tính, bạn phải xác định vị trí và hướng với các tọa độ và góc. Khi chúng ta có thể công nhận, mọi thứ quá dễ dàng để có được hiệu ứng màn hình đen nổi tiếng. Mặc dù một vài thứ có thể sai, thông thường bạn có được hiệu ứng này – chúng dẫn đến không có gì tuyệt đối được vẽ trong cửa sổ bạn mở trên màn hình- từ hướng đích sai của máy quay phim và chụp ảnh với mô hình đằng sau bạn. Một vấn đề tương tự phát sinh nếu bạn không chọn một phạm vi quan sát đủ rộng để thấy các đối tượng của bạn nhưng nhưng đủ hẹp để chúng xuất hiện to (lớn) hợp lý.
Nếu bạn đang nỗ lực cho chương trình nhưng chỉ tạo ra một cửa sổ đen, thử những bước dự đoán sau:
1. Kiểm tra những sự kiện hiển nhiên. Chắc chắn hệ thống của bạn được cắm vào. Hãy chắc chắn bạn đang vẽ những đối tượng của bạn với màu khác với màu bạn đang dùng để xóa màn hình. Chắc chắn với bất kì trạng thái bạn đang sử dụng (như là ánh sáng, kết cấu, trộn alpha, các phép toán logic, hoặc khử răng cưa) được bật hay tắt một cách chính xác như mong muốn.
2. Nhớ rằng với các lệnh phép chiếu, tọa độ near và far đánh giá khoảng cách từ điểm nhìn và (ngầm định) bạn đang nhìn theo chiều âm của trục z. Do đó, nếu giá trị near là 1.0 và far là 3.0 , các đối tượng phải có tọa độ z giữa -1.0 và -3.0 để có thể nhìn thấy được. Để đảm bảo rằng bạn đã không cắt mọi thứ ra khỏi cảnh của mình, tạm thời thiết lập mặt phẳng cắt near và far cho một số giá trị kể cả vô lý, như là 0.001 và 1000000.0. Điều này thay đổi diện mạo của các thao tác như là vùng đệm độ sâu va sương mù, như nó có thể vô tình để lộ ra những đối tượng đã bị cắt xén.
3. Xác định vị trí điểm nhìn, hướng bạn đang theo dõi, và vị trí những đối tượng của bạn. Nó có thể giúp cho việc tạo ra một không gian thực ba chiều – bằng cách sử dụng đôi tay của bạn, ví dụ- để tìm hiểu cho những điều này.

4. Chắc chắn rằng bạn biết vị trí bạn đang quay. Bạn có thể quay quanh một số vị trí tùy ý trừ khi bạn đã dịch về gốc đầu tiên. Nó cũng ổn để quay quanh điểm bất kì trừ khi bạn muốn quay quanh gốc tọa độ.
5. Kiểm tra hướng của bạn. Sử dụng **gluLookAt()** để nhắm tới khối quan sát vào những đối tượng của bạn. Hay vẽ các đối tượng của bạn ở chính hoặc gần gốc tọa độ, và sử dụng **glTranslate*()** như một biến đổi quan sát để di chuyển máy quay phim đủ xa theo hướng z để các đối tượng nằm trong khối quan sát. Một khi bạn đã điều khiển cho đối tượng được nhìn thấy, thử thay đổi khối quan sát tăng lên để đạt kết quả chính xác như bạn muốn, sẽ thảo luận trong phần tiếp theo.
Kể cả sau khi bạn đã hướng đúng máy quay phim và bạn có thể thấy các đối tượng, chúng có thể xuất hiện quá nhỏ hoặc quá to. Nếu bạn đang sử dụng **gluPerspective()**, bạn có thể phải cần tới sự thay đổi góc mà định nghĩa phạm vi quan sát bằng cách thay đổi giá trị của tham số đầu tiên trong lệnh này. Bạn có thể sử dụng lượng giác để tính toán phạm vi quan sát mong muốn đã cho sẵn kích cỡ của đối tượng và khoảng cách của nó từ điểm nhìn: tang của một nửa góc là một nửa kích cỡ của đối tượng được chia ra bởi khoảng cách tới đối tượng (xem Hình 3-19). Vì vậy bạn có thể sử dụng thủ tục arctang để tính toán một nửa góc mong muốn.
Ví dụ 3-3 giả sử một thủ tục như thế, **atan2()**, chúng tính toán artang đã cho độ dài của cạnh đối diện và cạnh kề của một tam giác vuông. Kết quả này sau đó cần chuyển đổi từ radian sang độ.



Hình 3-19 : Sử dụng lượng giác học để tính toán phạm vi quan sát
Ví dụ 3-3 : Tính toán phạm vi quan sát

```
#define PI 3.1415926535
double calculateAngle(double size, double distance)
{
     double radtheta, degtheta;
     radtheta = 2.0 * atan2 (size/2.0, distance);
     degtheta = (180.0 * radtheta) / PI;
     return (degtheta);
}
```

Tất nhiên, thông thường bạn không biết chính xác kích cỡ của đối tượng, và khoảng cách chỉ có thể xác định giữa điểm nhìn và một điểm lẻ trong cảnh của bạn. Để thu được một giá trị khá tốt gần đúng, tìm các hộp bao cho cảnh của bạn bằng việc xác định tọa độ x, y, z nhỏ nhất và lớn nhất của tất cả các đối tượng trong cảnh của bạn. Sau đó tính bán kính của một hình cầu biên cho hộp đó, và sử dụng tâm của hình cầu để xác định khoảng cách và bán kính để xác định kích thước.

Ví dụ, giả sử tất cả các tọa độ trong đối tượng của bạn thỏa mãn các phương trình - $1 \leq x \leq 3$, $5 \leq y \leq 7$, và $-5 \leq z \leq 5$. Sau đó, tâm của hộp bao là (1, 6, 0), và bán kính của một khối cầu bao là khoảng cách từ tâm của hộp đến góc bất kỳ - nói (3, 7, 5) - hoặc

$$\sqrt{(3-1)^2 + (7 - 6)^2 + (5 - 0)^2} = \sqrt{30} = 5.477$$

Nếu điểm nhìn là (8, 9, 10), khoảng cách giữa nó và tâm là

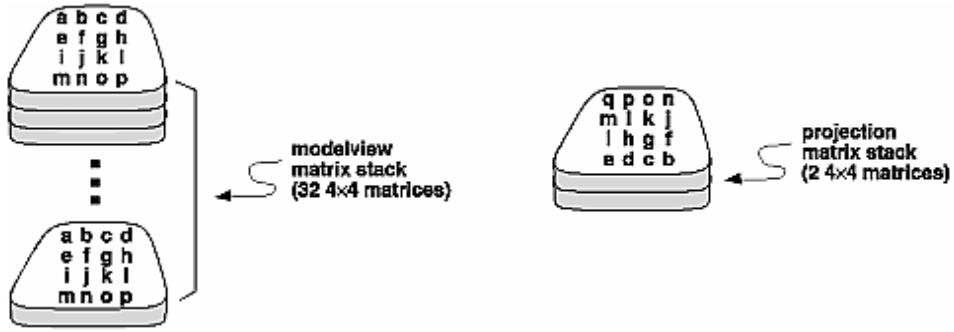
$$\sqrt{(8-1)^2 + (9 - 6)^2 + (10 - 0)^2} = \sqrt{158} = 12.570$$

$$\sqrt{(8-1)^2 + (9 - 6)^2 + (10 - 0)^2} = \sqrt{158} = 12.570$$

Các tang của nửa góc là 5,477 chia cho 12,570, bằng 0,4357, do đó, một nửa góc là 23,54 độ.
Hãy nhớ rằng góc phạm vi quan sát ảnh hưởng đến vị trí tối ưu cho điểm nhìn, nếu bạn đang cố gắng có được một ảnh thực. Ví dụ, nếu tính toán của bạn chỉ ra rằng bạn cần một phạm vi quan sát 179 độ, điểm nhìn phải là một phân số nhỏ của một inch từ màn hình để đạt được tính hiện thực. Nếu tính toán phạm vi quan sát của bạn quá lớn, bạn có thể cần phải di chuyển điểm nhìn xa hơn đối tượng.

Thao tác sắp xếp ma trận

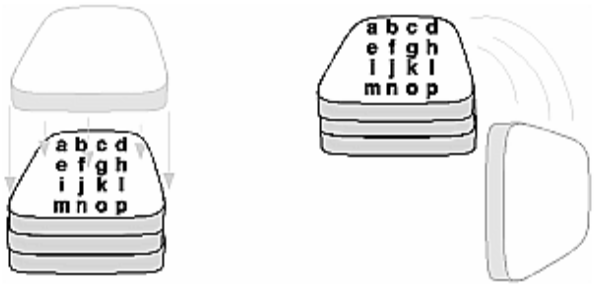
Các ma trận mô hình quan sát và phép chiếu bạn đã được tạo, nạp, và nhân chỉ xác định được đỉnh trong khối của chúng. Mỗi ma trận này thực sự là phần tử cao nhất của ngăn xếp ma trận. (xem Hình 3-20)



Hình 3-20 : Ngăn xếp ma trận mô hình quan sát và phép chiếu

Một ngăn xếp ma trận là hữu ích cho việc xây dựng mô hình có thứ bậc, nơi các đối tượng phức tạp được xây dựng từ những cái đơn giản hơn. Ví dụ, giả sử bạn đang vẽ một xe ô tô bốn bánh, mỗi bánh được gắn vào ô tô với năm cái ốc vít. Bạn có một thủ tục đơn để vẽ một bánh xe và thủ tục khác để vẽ một ốc vít, do tất cả các bánh xe và ốc vít giống nhau. Các thủ tục này vẽ một bánh xe hoặc một ốc vít ở một số vị trí và hướng thích hợp, nói tâm tại gốc tọa độ với trục của nó trùng với trục z. Khi bạn vẽ ô tô, bao gồm cả các bánh xe và những ốc vít, bạn muốn gọi thủ tục vẽ bánh xe bốn lần với những biến đổi khác nhau trong mỗi lần vẽ để định vị các bánh xe chính xác. Khi bạn vẽ mỗi bánh xe, bạn muốn vẽ các ốc vít năm lần, mỗi lần dịch chuyển tương ứng liên quan tới bánh xe.
Giả sử trong một phút tất cả bạn phải làm là vẽ thân ô tô và các bánh xe. Mô tả bằng tiếng Anh về những gì bạn muốn làm có thể là một cái gì đó như thế này:
Vẽ thân ô tô. Ghi nhớ bạn đang ở đâu, và dịch chuyển đến bánh xe trước bên phải. Vẽ bánh xe rồi quẳng đi tịnh tiến cuối sao cho vị trí hiện tại là quay về gốc tọa độ thân ô tô. Ghi nhớ bạn ở đâu, và tính tiến đến bánh xa trước bên trái…

Tương tự, với mỗi bánh xe, bạn muốn vẽ bánh xe, ghi nhớ rằng bạn ở đâu và tịnh tiến liên tục đến mỗi vị trí mà ốc vít được vẽ, quẳng đi các phép biến đổi sau mỗi ốc vít được vẽ.
Do các phép biến đổi được chứa như các ma trận, một ngăn xếp ma trận cung cấp một cơ chế lý tưởng cho việc thực hiện sắp xếp của việc nhớ, tịnh tiến và quẳng đi. Tất cả những thao tác ma trận đã được trình bày trước đây (**glLoadMatrix()**, **glMultMatrix()**, **glLoadIdentity())** và các lệnh tạo các ma trận biến đổi cụ thể) liên quan tới ma trận hiện tại, hoặc mà trận trên đỉnh của ngăn xếp. Bạn có thể điều khiển ma trận trên đỉnh với các lệnh thực hiện thao tác ngăn xếp: **glPushMatrix()**, chúng sao chép của ma trận hiện tại và bổ sung bản sao chép vào đỉnh của ngăn xếp, và **glPopMatrix()**, chúng loại bỏ ma trận trên cùng của ngăn xếp, như đã chỉ ra trong Hình 3-21(Nhớ rằng ma trận hiện tại luôn là ma trận trên cùng). Thực tế, **glPushMatrix()** nghĩa là "ghi nhớ bạn ở đâu " và **glPopMatrix()** nghĩa là " quay trở lại nơi bạn đã ở"



Hình 3-21 : Pushing và Popping ngăn xếp ma trận.
*void **glPushMatrix**(void);*
*Đẩy tất cả các ma trận vào ngăn xếp hiện thời xuống một mức. Ngăn xếp hiện thời được xác định bởi **glMatrixMode()**. Ma trận trên cùng được sao chép, sao cho nội dung của nó được sao lại trong cả ma trận trên cùng và thứ hai tính từ trên xuống. Nếu quá nhiều ma trận được đẩy vào, một lỗi bị sinh ra.*
*void **glPopMatrix**(void);*
*Lấy ma trận trên cùng ra khỏi ngăn xếp, hủy các nội dung của các ma trận bị loại. Mọi thứ là ma trận thứ hai tính từ đỉnh trở thành ma trận trên cùng. Ngăn xếp hiện thời được xác định bởi **glMatrixMode()**. Nếu ngăn xếp chứa một ma trận duy nhất, lời gọi **glPopMatrix()** tạo ra một lỗi.*
Ví dụ 3-4: vẽ một chiếc ô tô, giả sử tồn tại các thủ tục vẽ thân ô tô, một bánh xe và một ốc vít.
Ví dụ 3-4 : Pushing và Popping ma trận

```
draw_wheel_and_bolts()
{
   long i;
   draw_wheel();
   for(i=0;i<5;i++){
      glPushMatrix();
         glRotatef(72.0*i,0.0,0.0,1.0);
         glTranslatef(3.0,0.0,0.0);
         draw_bolt();
      glPopMatrix();
   }
}
draw_body_and_wheel_and_bolts()
```

```
{
   draw_car_body();
   glPushMatrix();
      glTranslatef(40,0,30)/*move to first wheel position*/
    draw_wheel_and_bolts();
   glPopMatrix();
   glPushMatrix();
      glTranslatef(40,0,-30); /*move to 2nd wheel position*/
      draw_wheel_and_bolts();
   glPopMatrix();
   ...              /*draw last two wheels similarly*/
}
```

Mã này giả sử các trục bánh xe và ốc vít là trùng với trục z, với mỗi ốc vít được cách đều nhau 72 độ, 3 đơn vị (có thể inch) từ tâm của bánh xe,và rằng các bánh xe trước là 40 đơn vị ở phía trước và 30 đơn vị ở bên phải và trái so với gốc tọa độ của chiếc xe. Một ngăn xếp hiệu quả hơn là một ma trận riêng lẻ, đặc biệt nếu ngăn xếp được cài đặt trong phần cứng. Khi bạn đẩy vào một ma trận, bạn không cần phải sao chép dữ liệu hiện tại quay lại xử lí chính, và phần cứng có thể được sao chép nhiều hơn một phần tử của ma trận trong một lần. Đôi khi, bạn có thể muốn duy trì một ma trận đơn vị tại đáy của ngăn xếp sao cho bạn không gọi **glLoadIdentity()** nhiều lần.

Ngăn xếp ma trận mô hình quan sát

Như bạn đã thấy trước đây trong "Các biến đổi quan sát và mô hình," ma trận mô hình quan sát bao gồm tích lũy phép nhân các ma trận biến đổi quan sát và mô hình. Mỗi biến đổi quan sát hay mô hình tạo ra một ma trận mới mà chúng được nhân ma trận mô hình quan sát hiện thời; kết quả chúng trở thành ma trận hiện thời mới, biểu diễn biến đổi tổ hợp. Ngăn xếp ma trận mô hình quan sát ma trận chứa ít nhất ba mươi hai ma trận 4x4; khởi tạo, ma trận trên cùng là ma trận đơn vị. Một số cài đặt của OpenGL có thể hỗ trợ hơn ba mươi hai ma trận trên ngăn xếp. Để tìm số lượng ma trận cho phép lớn nhất, bạn có thể sử dụng lệnh truy vấn **glGetIntegerv**(GL_MAX_MODELVIEW_STACK_DEPTH**,** GLint *params).

Ngăn xếp ma trận phép chiếu

Ma trận phép chiếu chứa một ma trận cho biến đổi phép chiếu, chúng mô tả khối quan sát. Nói chung, bạn không muốn tạo các ma trận phép chiếu, mà bạn chỉ ra **glLoadIdentity()** trước khi thực hiện một biến đổi phép chiếu. Vì lí do này, ngăn xếp ma trận phép chiếu chỉ cần hai mức sâu; một số cài đặt OpenGl có thể cho phép nhiều hơn hai ma trận 4x4. Để tìm ngăn xếp chiều sâu, gọi **glGetIntegerv**(GL_MAX_PROJECTION_STACK_DEPTH**,** GLint *params).

Sử dụng ma trận thứ hai trong ngăn xếp là một ứng dụng mà cần để hiểu thị trợ giúp cửa sổ bằng văn bản, cùng với cửa sổ chuẩn của nó chỉ ra một cảnh ba chiều. Do văn bản được đặt hầu hết với phép chiếu trực giao, bạn có thể thay đổi tạm hệ thời đến phép chiếu trực giao, hiển thị trợ giúp và sau đó quay trở lại phép chiếu trước của bạn:

```
glMatrixMode(GL_PROJECTION);
glPushMatrix(); /*save the hiện thời projection*/
glLoadIdentity();
glOrtho(...); /*set up for displaying help*/
display_the_help();
```

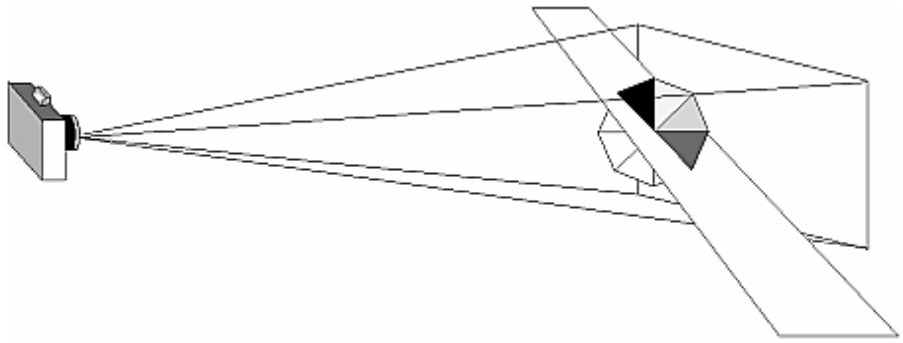
```
glPopMatrix();
```
Lưu ý rằng bạn cũng có thể phải thay đổi ma trận mô hình quan sát sao cho phù hợp

Nâng cao

Nếu bạn biết về toán học, bạn có thể tạo các ma trận phép chiếu tùy chỉnh mà thực hiện các phép biến đổi phép chiếu tùy ý. Ví dụ, OpenGl và Utility Library của nó không xây dựng theo cơ chế cho phối cảnh hai tâm chiếu. Nếu bạn đã thử mô phỏng việc vẽ trong văn bản thiết kế, bạn có thể cần một ma trận phép chiếu như thế.

Bổ sung mặt phẳng cắt

Thêm vào sáu mặt phẳng cắt của khối quan sát (trái, phải, dưới, trên, gần và xa), bạn có thể xác định các mặt phẳng cắt bổ sung để giới hạn thêm khối quan sát, như chỉ ra trong Hình 3-22. Điều này hữu ích cho việc loại bỏ những đối tượng ngoại cảnh -ví dụ, nếu bạn muốn hiển thị mô hình cắt của đối tượng. Mỗi mặt phẳng được xác định bởi hệ số trong phương trình Ax+By+Cz+D = 0. Các mặt phẳng cắt được biến đổi tự động thích hợp bằng các biến đổi mô hình và quan sát. Khối cắt trở thành phần giao nhau của khối quan sát với tất cả các nửa không gian được xác định bởi những mặt phẳng cắt bổ sung. Nhớ rằng các đa giác mà được cắt tự động có các cạnh được phục hồi thích hợp bởi OpenGL.



Hình 3-22 : Bổ sung những mặt phẳng cắt và khối quan sát

*void **glClipPlane**(GLenum plane, const GLdouble *equation);*

*Định nghĩa một mặt phẳng cắt. Các đối số phương trình trở đến bốn hệ số của phương trình mặt phẳng, Ax+By+Cz+D = 0. Tất cả các điểm với tọa độ mắt nhìn (xe, ye, ze, we) mà thỏa mãn (A B C D)M-1 (xe ye ze we)T >= 0 nằm trên một nửa không gian xác định bởi mặt phẳng, với M là ma trận mô hình quan sát hiện thời tại thời điểm **glClipPlane()** được gọi. Tất cả các điểm không nằm trong nửa không gian này bị cắt.*

Bạn cần kích hoạt mỗi mặt phẳng cắt bổ sung bằng cách:

```
glEnable(GL_CLIP_PLANE i);
```

Bạn có thể hủy kích hoạt một mặt phẳng với:

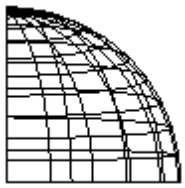
```
glDisable(GL_CLIP_PLANE i);
```

Tất cả cài đặt của OpenGL phải được hỗ trợ ít nhất là sáu mặt phẳng cắt bổ sung, mặc dù một số cài đặt có thể cho phép hơn thế. Bạn có thể sử dụng **glGetIntegerv()** với GL_MAX_CLIP_PLANES để tìm có bao nhiêu mặt phẳng cắt được hỗ trợ.

Lưu ý : Thực hiện cắt xén là kết quả **glClipPlane()** được thực hiện theo hệ tọa độ mắt, không theo hệ tọa độ cắt. Sự khác nhau này dễ nhận thấy nếu ma trận phép chiếu là duy nhất (có nghĩa là, một ma trận phép chiếu san phẳng các hệ tọa độ ba chiều thành các ma trận hai chiều). Cắt xén thực hiện theo hệ tọa độ mắt tiếp tục lấy vị trí trong ba chiều thậm chí khi ma trận phép chiếu là duy nhất.

Ví dụ mã mặt phẳng cắt

Ví dụ 3-5: tô vẽ một lưới khối cầu với 2 mặt phẳng cắt mà cắt đi ba phần tư khối cầu ban đầu, chỉ ra trong Hình 3-23.



Hình 3-23 : Lưới khối cầu bị cắt

Ví dụ 3-5 : Lưới khối cầu với hai mặt phẳng cắt : clip.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
void init(void)
{
glClearColor (0.0, 0.0, 0.0, 0.0);
glShadeModel (GL_FLAT);
}
void display(void)
{
GLdouble eqn[4] = {0.0, 1.0, 0.0, 0.0};
GLdouble eqn2[4] = {1.0, 0.0, 0.0, 0.0};
glClear(GL_COLOR_BUFFER_BIT);
glColor3f (1.0, 1.0, 1.0);
glPushMatrix();
glTranslatef (0.0, 0.0, -5.0);
/* clip lower half -- y < 0 */
glClipPlane (GL_CLIP_PLANE0, eqn);
glEnable (GL_CLIP_PLANE0);
/* clip left half -- x < 0 */
glClipPlane (GL_CLIP_PLANE1, eqn2);
glEnable (GL_CLIP_PLANE1);
glRotatef (90.0, 1.0, 0.0, 0.0);
glutWireSphere(1.0, 20, 16);
glPopMatrix();
glFlush ();
}
void reshape (int w, int h)
{
glViewport (0, 0, (GLsizei) w, (GLsizei) h);
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
gluPerspective(60.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
glMatrixMode (GL_MODELVIEW);
```

```
}
int main(int argc, char** argv)
{
glutInit(&argc, argv);              glutInitDisplayMode (GLUT_SINGLE
| GLUT_RGB);
glutInitCửa sổSize (500, 500);
glutInitCửa sổPosition (100, 100);
glutCreateCửa sổ (argv[0]);
init ();
glutDisplayFunc(display);
glutReshapeFunc(reshape);
glutMainLoop();
return 0;
}
```

Thử điều này

- Thử thay đổi các hệ số mô tả các mặt phẳng cắt trong Ví dụ 3-5.
- Thử gọi một biến đổi mô hình, ví dụ **glRotate*()**, để tác động lên **glClipPlane()**. Tạo mặt phẳng cắt di chuyển độc lập với các đối tượng trong cảnh.

Các ví dụ biên soạn một vài biến đổi

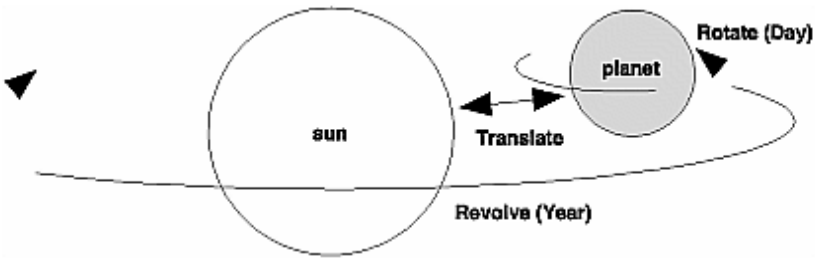
Phần này giải thích cách kết hợp một số biến đổi để đạt được kết quả cụ thể. Hai ví dụ đã thảo luận là hệ mặt trời, trong đó các đối tượng cần quay quanh trục của chúng cũng như quanh quỹ đạo của nhau, và một cánh tay robot, chúng có một số khớp nối mà hệ tọa độ biến đổi một cách hiệu quả giống như chúng di chuyển liên quan với nhau.

Xây dựng một hệ thống mặt trời

Chương trình mô tả trong phần này vẽ một hệ thống mặt trời đơn giản với một hành tinh và một mặt trời, cả hai đều sử dụng thủ tục vẽ khối cầu. Để viết chương trình này, bạn cần sử dụng **glRotate*()** đối với vòng quay hành tinh quanh mặt trời và đối với vòng quay hành tinh quanh trục của nó. Bạn cũng cần **glTranslate*()** để di chuyển hành tinh ra ngoài quỹ đạo của nó, ra xa tâm của hệ mặt trời. Nhớ rằng bạn có thể chỉ ra kích cỡ mong muốn của hai khối cầu bằng cách cung cấp đối số thích hợp cho thủ tục **glutWireSphere().**

Để vẽ hệ mặt trời, đầu tiên bạn cài đặt một biến đổi phép chiếu và một biến đổi quan sát. Ví dụ, **gluPerspective()**va **gluLookAt()** được sử dụng.

Vẽ mặt trời khá dễ, do nó được đặt tại vị trí tọa độ cố định, trung tâm, là nơi vị trí thủ tục khối cầu đặt ở đó. Do đó, vẽ mặt trời không yêu cầu tịnh tiến; bạn có thể sử dụng **glRotate*()** để tạo mặt trời quay quanh trục tùy ý. Để vẽ một hành tinh quay quanh mặt trời như chỉ ra trong Hình 3-24, yêu cầu một số biến đổi mô hình. Hành tinh cần quay quanh trục của nó mỗi ngày một lần. Và mỗi năm, hành tinh hoàn thành một vòng quay quanh mặt trời



Hình 3-24 : Hành tinh và mặt trời

Để xác định thứ tự của các biến đổi mô hình, tưởng tượng những gì xảy ra cho hệ tọa độ cục bộ. Một **glRotate*()** khởi tạo quay hệ tọa độ cục bộ mà lúc đầu trùng với hệ tọa độ chính. Tiếp theo, **glTranslate*()** di chuyển sao hệ tọa độ cục bộ đến một vị trí trên quỹ đạo của hành tinh; khoảng cách di chuyển nên bằng với bán kính của quỹ đạo. Vì thế, khởi tạo **glRotate*()** xác định vị trí thực tế trên quỹ đạo hành tinh (hay đây là thời gian nào trong năm).

glRotate*() thứ hai quay hệ tọa độ cục bộ quanh các trục cục bộ, do đó việc xác định thời gian một ngày cho hành tinh. Một lần bạn đưa ra tất cả các lệnh biến đổi này, hành tinh có thể được vẽ. Tóm lại, đây là các lệnh OpenGl để vẽ mặt trời và hành tinh: toàn bộ chương trình được trình bày trong Ví dụ 3-6.

```
glPushMatrix();
glutWireSphere(1.0, 20, 16); /* draw sun */
glRotatef ((GLfloat) year, 0.0, 1.0, 0.0);
glTranslatef (2.0, 0.0, 0.0);
glRotatef ((GLfloat) day, 0.0, 1.0, 0.0);
glutWireSphere(0.2, 10, 8); /* draw smaller planet */
glPopMatrix();
```

Ví dụ 3-6 : Hệ hành tinh: planet.c

```
 #include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
static int year = 0, day = 0;
void init(void)
{
glClearColor (0.0, 0.0, 0.0, 0.0);
glShadeModel (GL_FLAT);
}
void display(void)
{
glClear (GL_COLOR_BUFFER_BIT);
glColor3f (1.0, 1.0, 1.0);
glPushMatrix();
glutWireSphere(1.0, 20, 16); /* draw sun */
glRotatef ((GLfloat) year, 0.0, 1.0, 0.0);
glTranslatef (2.0, 0.0, 0.0);
glRotatef ((GLfloat) day, 0.0, 1.0, 0.0);
glutWireSphere(0.2, 10, 8); /* draw smaller planet */
```

```
glPopMatrix();
glutSwapBuffers();
}
void reshape (int w, int h)
{
glViewport (0, 0, (GLsizei) w, (GLsizei) h);
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
gluPerspective(60.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
}
void keyboard (unsigned char key, int x, int y)
{
switch (key) {
case 'd':
day = (day + 10) % 360;
glutPostRedisplay();
break;
case 'D':
day = (day - 10) % 360;
glutPostRedisplay();
break;
case 'y':
year = (year + 5) % 360;
glutPostRedisplay();
break;
case 'Y':
year = (year - 5) % 360;
glutPostRedisplay();
break;
default:
break;
}
}
int main(int argc, char** argv)
{
glutInit(&argc, argv);
glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
glutInitCửa sổSize (500, 500);
glutInitCửa sổPosition (100, 100);
glutCreateCửa sổ (argv[0]);
init ();
glutDisplayFunc(display);
```

```
glutReshapeFunc(reshape);
glutKeyboardFunc(keyboard);
glutMainLoop();
return 0;
}
```

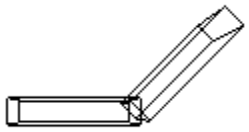
Thử điều này

Thử cho thêm mặt trăng vào với hành tinh. Hoặc một vài mặt trăng và một vài hành tinh.

Gợi ý : sử dụng **glPushMatrix()**và **glPopMatrix()** để lưu và khôi phục vị trí và hướng của hệ tọa độ tại thời điểm thích hợp. Nếu bạn định vẽ một số mặt trăng quanh hành tinh, bạn cần lưu hệ tọa độ trước khi bố trí mỗi mặt trăng và khôi phục hệt tọa độ sau khi mỗi mặt trăng được vẽ. Cố gắng làm nghiêng trục của hành tinh đi một chút.

Xây dựng một cánh tay robot có khớp nối

Phần này nói về một chương trình tạo ra một cánh tay robot có khớp với hai hay nhiều đoạn. Phần cánh tay sẽ được kết nối với điểm trục ở vai, khuỷu tay, hoặc những khớp nối khác. Ví dụ 3-25 chỉ ra một khớp nối đơn của một cánh tay như thế.



Ví dụ 3-25 : cánh tay rotbot

Bạn có thể sử dụng một hình lập phương như một phần của cánh tay robot, nhưng trước tiên bạn phải gọi biến đổi mô hình hợp lý để định hướng từng phần. Do gốc của hệ tọa độ cục bộ khởi tạo tại tâm của hình lập phương, bạn cần di chuyển hệ tọa độ cục bộ đến một cạnh của hình lập phương. Mặt khác, hình lập phương quay quanh tâm của nó hơn là điểm trục.

Sau khi bạn gọi **glTranslate*()** để thiết lập điểm trục và **glRotate*()** để xác định lại trục hình lập phương, dịch chuyển trở lại tâm hình lập phương. Sau đó, hình lập phương được co dãn (dẹt và rộng) trước khi được vẽ. **glPushMatrix()** và **glPopMatrix()** hạn chế tác động của **glScale*()**. Đây là những gì mã của bạn có thể thấy cho phần đầu của cánh tay (toàn bộ chương trình được trình bày trong Ví dụ 3-7)

```
glTranslatef (-1.0, 0.0, 0.0);
glRotatef ((GLfloat) shoulder, 0.0, 0.0, 1.0);
glTranslatef (1.0, 0.0, 0.0);
glPushMatrix();
glScalef (2.0, 0.4, 1.0);
glutWireCube (1.0);
glPopMatrix();
```

Để xây dựng phần thứ hai, bạn cần phải di chuyển hệ tọa độ cục bộ đến điểm trục tiếp theo. Do hệ tọa độ đã được quay trước đó, trục x đã hướng theo chiều dài của cánh tay quay. Do đó, việc dịch chuyển theo trục x di chuyển hệ tọa độ cục bộ đến điểm trục tiếp theo. Mỗi khi nó ở điểm trục đó, bạn có thể sử dụng mã tương tự để vẽ phần thứ hai như đã sử dụng cho phần đầu tiên. Điều này có thể tiếp tục với số lượng các phần không giới hạn (vai, khuỷu tay, cổ tay, những ngón tay)

```
glTranslatef (1.0, 0.0, 0.0);
glRotatef ((GLfloat) elbow, 0.0, 0.0, 1.0);
glTranslatef (1.0, 0.0, 0.0);
```

```
glPushMatrix();
glScalef (2.0, 0.4, 1.0);
glutWireCube (1.0);
glPopMatrix();
```

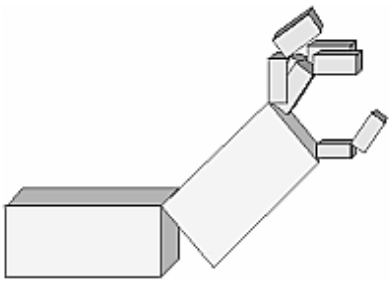
Ví dụ 3-7 : Cánh tay Robot: robot.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
static int shoulder = 0, elbow = 0;
void init(void)
{
glClearColor (0.0, 0.0, 0.0, 0.0);
glShadeModel (GL_FLAT);
}
void display(void)
{
glClear (GL_COLOR_BUFFER_BIT);
glPushMatrix();
glTranslatef (-1.0, 0.0, 0.0);
glRotatef ((GLfloat) shoulder, 0.0, 0.0, 1.0);
glTranslatef (1.0, 0.0, 0.0);
glPushMatrix();
glScalef (2.0, 0.4, 1.0);
glutWireCube (1.0);
glPopMatrix();
glTranslatef (1.0, 0.0, 0.0);
glRotatef ((GLfloat) elbow, 0.0, 0.0, 1.0);
glTranslatef (1.0, 0.0, 0.0);
glPushMatrix();
glScalef (2.0, 0.4, 1.0);
glutWireCube (1.0);
glPopMatrix();
glPopMatrix();
glutSwapBuffers();
}
void reshape (int w, int h)
{
glViewport (0, 0, (GLsizei) w, (GLsizei) h);
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
gluPerspective(65.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef (0.0, 0.0, -5.0);
}
```

```
void keyboard (unsigned char key, int x, int y)
{
switch (key) {
case `s': /* s key rotates at shoulder */
shoulder = (shoulder + 5) % 360;
glutPostRedisplay();
break;
case `S':
shoulder = (shoulder - 5) % 360;
glutPostRedisplay();
break;
case `e': /* e key rotates at elbow */
elbow = (elbow + 5) % 360;
glutPostRedisplay();
break;
case `E':
elbow = (elbow - 5) % 360;
glutPostRedisplay();
break;
default:
break;
}
}
int main(int argc, char** argv)
{
glutInit(&argc, argv);
glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
glutInitCửa sổSize (500, 500);
glutInitCửa sổPosition (100, 100);
glutCreateCửa sổ (argv[0]);
init ();
glutDisplayFunc(display);
glutReshapeFunc(reshape);
glutKeyboardFunc(keyboard);
glutMainLoop();
return 0;
}
```

Thử điều này

- Thay đổi Ví dụ 3-7 để tạo thêm các phần trên cánh tay robot.
- Thay đổi Ví dụ 3-7 để tạo thêm các phần ở vị trí giống nhau. Ví dụ như, cho một cánh tay robot một số ngón tay tại cổ tay, như chỉ ra ở Hình 3-26. Gợi ý: sử dụng **glPushMatrix()** và **glPopMatrix()** để lưu và khôi phục vị trí và hướng của hệ tọa độ tại cổ tay. Nếu bạn định vẽ các ngón tay tại cổ tay, bạn cần lưu ma trận hiện tại trước khi đặt vị trí mỗi ngón tay và khôi phục ma trận hiện tại sau khi mỗi ngón tay được vẽ.



Hình 3-26 : Cánh tay Robot với các ngón tay

Các phép biến đổi nghịch đảo hoặc đối xứng.

Quy trình xử lí đồ họa rất tốt ở việc sử dụng ma trận quan sát và phép chiếu và một cổng nhìn cho việc cắt xén để biến đổi tọa độ thực (hay đối tượng) của một đỉnh vào trong cửa sổ (hay màn hình). Tuy nhiên, có các tình huống mà bạn muốn đảo ngược quá trình đó. Một trường hợp phổ biến là khi một ứng dụng người dùng tậnd ụng chuột để lựa chọn một vị trí trong ba chiều. Chuột sẽ chỉ trả về một giá trị hai chiều là vị trí màn hình của con trỏ. Bởi vậy, ứng dụng sẽ phải đảo ngược quá trình biến đổi để xác định từ vị trí bắt đầu màn hình này trong không gian ba chiều.

Thủ tục thư viện tiện ích **gluUnProject()** thực hiện đảo ngược của các biến đổi này. Cho các tọa độ cửa sổ ba chiều cho một vị trí và tất cả những biến đổi mà tác động lên chúng **gluUnProject()** trả về tọa độ thực từ nơi nó đã bắt đầu.

*int **gluUnProject**(GLdouble winx, GLdouble winy, GLdouble winz, const GLdouble modelMatrix[16], const GLdouble projMatrix[16], const GLint viewport[4], GLdouble *objx, GLdouble *objy, Gldouble *objz);*

*Ánh xạ các tọa độ cửa sổ xác định (winx, winy, winz) thành các tọa độ đối tượng, việc sử dụng các biến đổi được xác định bởi một ma trận mô hình quan sát (modelMatrix), ma trận phép chiếu (projMatrix), và cổng nhìn (viewport). Tọa độ đối tượng kết quả được trả về theo objx, objy, và objz. Hàm trả về GL_TRUE, ám chỉ thành công, hay GL_FALSE, ám chỉ thất bại (ví dụ một ma trận không thể nghịch đảo). Thao tác này không cố gắng để cắt các tọa độ đến cổng nhìn hay khử các giá trị độ sâu mà thuộc bên ngoài **glDepthRange()**.*

Đây là những khó khăn vốn có trong việc cố gắng đảo ngược quá trình biến đổi. Một vị trí màn hình hai chiều có thể được bắt đầu từ nơi bất kì trên toàn bộ dòng trong gian ba chiều, Để kết quả có nghĩa, **gluUnProject()** yêu cầu tọa độ chiều sâu cửa sổ (winz) được cung cấp và winz được xác định theo **glDepthRange()**. Đối với các giá trị ngầm định của **glDepthRange()**, winz bằng 0 sẽ yêu cầu các tọa độ thực của điểm biến đổi tại mặt phẳng cắt gần, trong khi winz bằng 1 sẽ yêu cầu điểm tại mặt phẳng cắt xa.

Ví dụ 3-8 chứng minh **gluUnProject()** bằng việc đọc vị trí chuột và xác định các điểm 3 chiều tại các mặt phẳng cắt gần và mặt phẳng cắt xa tính từ điểm biến đổi. Việc tính toán các tọa độ thực được in để chuẩn hóa đầu ra, nhưng cửa sổ tự kết xuất nó chỉ là màu đen.

Ví dụ 3-8 : Đảo ngược quy trình xử lí đồ họa: unproject.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdlib.h>
#include <stdio.h>
void display(void)
```

```
{
glClear(GL_COLOR_BUFFER_BIT);
glFlush();
}
void reshape(int w, int h)
{
glViewport (0, 0, (GLsizei) w, (GLsizei) h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective (45.0, (GLfloat) w/(GLfloat) h, 1.0, 100.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
}
void mouse(int button, int state, int x, int y)
{
GLint viewport[4];
GLdouble mvmatrix[16], projmatrix[16];
GLint realy; /* OpenGL y coordinate position */
GLdouble wx, wy, wz; /* returned world x, y, z coords */
switch (button) {
case GLUT_LEFT_BUTTON:
if (state == GLUT_DOWN) {
glGetIntegerv (GL_VIEWPORT, viewport);
glGetDoublev (GL_MODELVIEW_MATRIX, mvmatrix);
glGetDoublev (GL_PROJECTION_MATRIX, projmatrix);
/* note viewport[3] is height of cửa sổ in pixels */
realy = viewport[3] - (GLint) y - 1;
printf ("Coordinates at cursor are (%4d, %4d)\n",
x, realy);
gluUnProject ((GLdouble) x, (GLdouble) realy, 0.0,
mvmatrix, projmatrix, viewport, &wx, &wy, &wz);
printf ("World coords at z=0.0 are (%f, %f, %f)\n",
wx, wy, wz);
gluUnProject ((GLdouble) x, (GLdouble) realy, 1.0,
mvmatrix, projmatrix, viewport, &wx, &wy, &wz);
printf ("World coords at z=1.0 are (%f, %f, %f)\n",
wx, wy, wz);
}
break;
case GLUT_RIGHT_BUTTON:
if (state == GLUT_DOWN)
exit(0);
break;
default:
break;
```

```
}
}
int main(int argc, char** argv)
{
glutInit(&argc, argv);
glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
glutInitCửa sổSize (500, 500);
glutInitCửa sổPosition (100, 100);
glutCreateCửa sổ (argv[0]);
glutDisplayFunc(display);
glutReshapeFunc(reshape);
glutMouseFunc(mouse);
glutMainLoop();
return 0;
}
```

gluProject() là thủ tục khác của Utility Library, chúng liên quan tới **gluUnProject().gluProject()** bắt chước những hành động của quy trình biến đổi. Cho các tọa độ thực ba chiều và tất cả những biến đổi mà tác động lên chúng, **gluProject()** trả về tọa độ cửa sổ đã được biến đổi.

*int **gluProject**(GLdouble objx, GLdouble objy, GLdouble objz, const GLdouble modelMatrix[16], const GLdouble projMatrix[16], const GLint viewport[4], GLdouble *winx, GLdouble *winy, Gldouble *winz);*

Ánh xạ các tọa độ đối tượng đã xác định (objx, objy, objz) thành tọa độ cửa sổ, bằng việc sử dụng các biến đổi được xác định bởi một ma trận mô hình quan sát (modelMatrix), ma trận phép chiếu (projMatrix), và cổng nhìn (viewport). Các tọa độ cửa sổ kết quả được trả về theo winx, winy, và winz.

Hàm trả về GL_TRUE, nếu thành công, hoặc GL_FALSE nếu thất bại.