Hướng dẫn lập trình OpenGL căn bản

Tác giả: Lê Phong

Tài liệu này được viết với mục đích hướng dẫn lập trình OpenGL ở mức căn bản. Người đọc đã phải nắm được một số kiến thức thiết yếu về đồ họa 3D.

Tài liệu được viết dựa vào các chương 1, 2, 3, 4 và 13 trong OpenGL redbook

http://glprogramming.com/red

có lược bỏ đi những kiến thức chưa cần thiết và tổ chứ lại, diễn giải lại ý cho rõ ràng hơn.

Người đọc được đề nghị tham khảo trực tiếp trong sách đó.

Chương 1: Giới thiệu về OpenGL

1. OpenGL là gì

OpenGL là bộ thư viện đồ họa có khoảng 150 hàm giúp xây dựng các đối tượng và giao tác cần thiết trong các ứng dụng tương tác 3D.

Những thứ OpenGL không hỗ trợ

- bản thân OpenGL không có sẵn các hàm nhập xuất hay thao tác trên window,
- OpenGL không có sẵn các hàm cấp cao để xây dựng các mô hình đối tượng, thay vào đó, người dùng phải tự xây dựng từ các thành phần hình học cơ bản (điểm, đoạn thẳng, đa giác).

Rất may là một số thư viện cung cấp sẵn một số hàm cấp cao được xây dựng nên từ OpenGL. GLUT (OpenGL Utility Toolkit) là một trong số đó và được sử dụng rộng rãi. Trong tài liệu này, chúng ta sẽ sử dụng chủ yếu là OpenGL và GLUT.

Những thứ OpenGL hỗ trợ là các hàm đồ họa

- xây dựng các đối tượng phức tạp từ các thành phần hình học cơ bản (điểm, đoạn, đa giác, ảnh, bitmap),
- sắp xếp đối tượng trong 3D và chọn điểm thuận lợi để quan sát,
- tính toán màu sắc của các đối tượng (màu sắc của đối tượng được quy định bởi điều kiện chiếu sáng, texture của đối tượng, mô hình được xây dựng hoặc là kết hợp của cả 3 yếu tố đó),
- biến đổi những mô tả toán học của đối tượng và thông tin màu sắc thành các pixel trên màn hình (quá trình này được gọi là resterization).

2. Cấu trúc lệnh trong OpenGL

OpenGL sử dụng tiền tố `gl` và tiếp theo đó là những từ được viết hoa ở chữ cái đầu để tạo nên tên của một lệnh, ví dụ `glClearColor()`. Tương tự, OpenGL đặt tên các hằng số bắt đầu bằng `GL_` và các từ tiếp sau đều được viết hoa và cách nhau bởi dấu '_', ví dụ: `GL_COLOR_BUFFER_BIT`.

Bên cạnh đó, với một số lệnh, để ám chỉ số lượng cũng như kiểu tham số được truyền, một số hậu tố được sử dụng như trong bảng sau

Hậu tố	*Kiểu dữ liệu*	*Tương ứng với kiểu trong C*	*Tương ứng với kiểu trong OpenGL*

B	8-bit integer	signed char	Glbyte
S	16-bit integer	Short	Glshort
I	32-bit integer	int or long	GLint, Glsizei
F	32-bit floating-point	Float	GLfloat, Glclampf
D	64-bit floating-point	Double	GLdouble, GLclampd
Ub	8-bit unsigned integer	unsigned char	GLubyte, GLboolean
Us	16-bit unsigned integer	unsigned short	GLushort
Ui	32-bit unsigned integer	unsigned int or unsigned long	GLuint, GLenum, GLbitfield

Ví dụ: `glVertex2i(1,3)` tương ứng với xác định một điểm (x,y) với x, y nguyên (integer).

Lưu ý: OpenGL có định nghĩa một số kiểu biến, việc sử dụng các định nghĩa này thay vì định nghĩa có sẵn của C sẽ tránh gây lỗi khi biên dịch code trên một hệ thống khác.

Một vài lệnh của OpenGL kết thúc bởi `v` ám chỉ rằng tham số truyền vào là một vector.

Ví dụ: `glColor3fv(color_array)` thì `color_array` là mảng 1 chiều có 3 phần tử là float.

3. OpenGL Utility Toolkit (GLUT)

Để khắc phục một số nhược điểm của OpenGL, GLUT được tạo ra với với nhiều hàm hỗ trợ

- quản lý window
- display callback
- nhập xuất (bàn phím, chuột,…)
- vẽ một số đối tượng 3D phức tạp (mặt cầu, khối hộp,…)

Tên các hàm của GLUT đều có tiền tố là `glut`. Để hiểu rõ hơn về GLUT, người đọc tham khảo ở

http://glprogramming.com/red/appendixd.html

4. Một số ví dụ đơn giản

Để khai báo sử dụng OpenGL và GLUT, chúng ta download ở đây

http://www.opengl.org/resources/libraries/glut/glut_downloads.php#windows

và chép các file sau vào trong cùng thư mục của project.

- glut.h
- glut32.dll
- glut32.lib

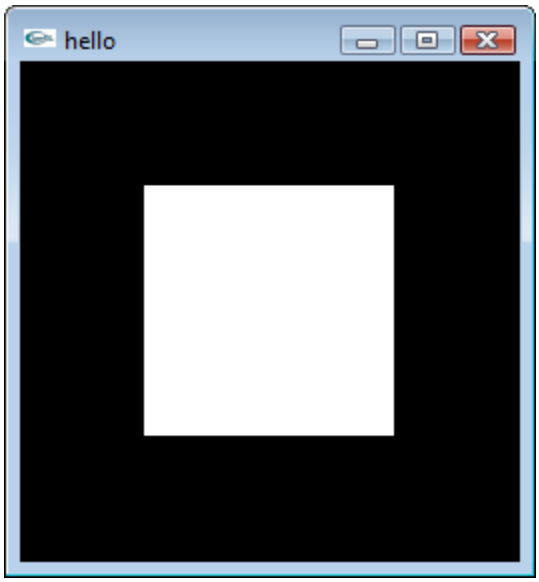
4.1. Ví dụ 1

Chúng ta sẽ vẽ một hình chữ nhật màu trắng trên nền đen.

```
#include "glut.h"

/*  hàm thực hiện các thao tác vẽ theo yêu cầu của chương trình  */
void display(void)
{
/*  xóa mọi pixel  */
    glClear (GL_COLOR_BUFFER_BIT);

/*  vẽ hình chữ nhật có điểm trái-trên và phải-dưới
 *  (0.25, 0.25, 0.0) and (0.75, 0.75, 0.0)
 */
    glColor3f (1.0, 1.0, 1.0);  /*  thiết lập màu vẽ: màu trắng  */
    glBegin(GL_POLYGON);   /*  bắt đầu vẽ đa giác  */
        glVertex3f (0.25, 0.25, 0.0);  /*  xác định các đỉnh của đa giác  */
        glVertex3f (0.75, 0.25, 0.0);
        glVertex3f (0.75, 0.75, 0.0);
        glVertex3f (0.25, 0.75, 0.0);
    glEnd();  /*  kết thúc vẽ đa giác  */

/*
 *  thực hiện quá trình đẩy ra buffer
 */
    glFlush ();
}

/*  hàm thực hiện các khởi tạo */
void init (void)
{
/*  chọn màu để xóa nền (tức là sẽ phủ nền bằng màu này)    */
    glClearColor (0.0, 0.0, 0.0, 0.0); /* màu đen */

/*  thiết lập các thông số cho view  */
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
}

/*  hàm main của chương trình */
```

```
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB); /* khởi tạo chế độ vẽ
single buffer và hệ màu RGB */
    glutInitWindowSize (250, 250); /* khởi tạo window kích thước 250 x 250 */
    glutInitWindowPosition (100, 100);  /*  khởi tạo window tại ví trí
(100,100) trên screen */
    glutCreateWindow ("rectangle"); /*  tên của window là 'rectangle' */
    init ();  /* khởi tạo một số chế độ đồ họa */
    glutDisplayFunc(display); /*  thiết lập hàm vẽ là hàm display() */
    glutMainLoop();     /*  bắt đầu chu trình lặp thể hiện vẽ  */
    return 0;
}
```

Kết quả khi chạy chương trình



4.2. Ví dụ 2

Chúng ta sẽ vẽ hình chữ nhật tương tự như trong ví dụ 1, hơn nữa, hình chữ nhật này sẽ quay quanh tâm của nó.

Để tránh trường hợp hình bị 'giựt' khi chuyển động, chúng ta sẽ không dùng single buffer như ở ví dụ1 mà sẽ dùng double buffer. Ý tưởng của double buffer là

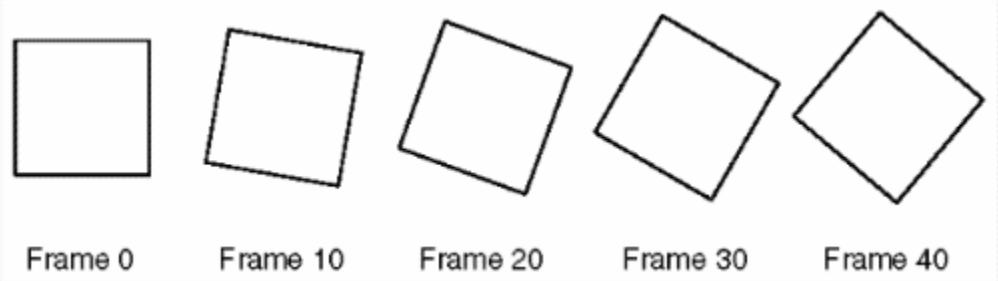
- trong khi buffer 1 đang được dùng để trình diễn frame *t* trên screen thì chương trình sẽ dùng buffer 2 để chuẩn bị cho frame *t+1*,
- khi đến lượt trình diễn frame *t+1* thì chương trình chỉ cần thể hiện buffer 2 và đưa buffer 1 về đằng sau để chuẩn bị cho frame *t+2*.

Do đó mà thời gian chuyển tiếp giữa 2 frame liên tiếp sẽ rất nhỏ và mắt người không phát hiện ra được, dẫn đến việc trình diễn các frame liên tiếp sẽ rất mượt.

```
#include "glut.h"

static GLfloat spin = 0.0; /* góc quay hiện tại của hình chữ nhật */

void init(void)
{
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glShadeModel (GL_FLAT);
}

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glPushMatrix();
   glRotatef(spin, 0.0, 0.0, 1.0);  /* xoay một góc spin quanh trục z */
   glColor3f(1.0, 1.0, 1.0); /* thiết lập màu vẽ cho hcn (màu trắng) */
   glRectf(-25.0, -25.0, 25.0, 25.0); /* vẽ hcn */
   glPopMatrix();
   glutSwapBuffers();  /* thực hiện việc hoán đổi 2 buffer */
}

void spinDisplay(void)
{
   spin = spin + 2.0;  /* xoay thêm 2 deg cho mỗi lần lặp */
   if (spin > 360.0)
      spin = spin - 360.0;
   glutPostRedisplay();  /* thông báo cho chương trình rằng: cần phải thực
hiện việc vẽ lại */
}

/* các thao tác cần làm khi cửa sổ bị thay đổi kích thước */
void reshape(int w, int h)
{
   glViewport (0, 0, (GLsizei) w, (GLsizei) h); /* thay đổi viewport */
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glOrtho(-50.0, 50.0, -50.0, 50.0, -1.0, 1.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}

/* các thao tác xử lý chuột */
void mouse(int button, int state, int x, int y)
{
   switch (button) {
      case GLUT_LEFT_BUTTON:  /* khi nhấn chuột trái */
         if (state == GLUT_DOWN)
            glutIdleFunc(spinDisplay); /* khi chương trình đang trong trạng
thái idle (không phải xử lý gì cả) thì sẽ thực hiện hàm spinDisplay */
         break;
      case GLUT_MIDDLE_BUTTON:  /* khi nhấn nút giữa */
         if (state == GLUT_DOWN)
            glutIdleFunc(NULL);
```

```
            break;
         default:
            break;
      }
}

/* hàm main của chương trình */
int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);  /* khai báo việc sử dụng
chế độ double buffer */
   glutInitWindowSize (250, 250);
   glutInitWindowPosition (100, 100);
   glutCreateWindow ("spinning rectangle");
   init ();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape); /* đăng ký hàm reshape cho sự kiện cửa sổ bị
thay đổi kích thước */
   glutMouseFunc(mouse); /* đăng ký hàm mouse cho sự kiện về chuột */
   glutMainLoop();
   return 0;
}
```

Chương trình sẽ chạy như sau nếu chúng ta click chuột trái vào hình chữ nhật



Chương 2: Vẽ các đối tượng hình học

1. Một số thao tác cơ bản

1.1. Xóa màn hình

Trong OpenGL có 2 loại buffer phổ biến nhất

- *color buffer*: buffer chứa màu của các pixel cần được thể hiện
- *depth buffer* (hay còn gọi là z-buffer): buffer chứa chiều sâu của pixel, được đo bằng khoảng cách đến mắt. Mục đích chính của buffer này là loại bỏ phần đối tượng nằm sau đối tượng khác.

Mỗi lần vẽ, chúng ta nên xóa buffer

```
glClearColor(0.0, 0.0, 0.0, 0.0); /* xác định màu để xóa color buffer
(màu đen) */
glClearDepth(1.0); /* xác định giá trị để xóa depth buffer */
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);  /* xóa color
buffer và depth buffer */
```

1.2. Xác định màu

Khi vẽ một đối tượng, OpenGL sẽ tự động sử dụng màu đã được xác định trước đó. Do đó, để vẽ đối tượng với màu sắc theo ý mình, cần phải thiết lập lại màu vẽ. Thiết lập màu vẽ mới dùng hàm `glColor3f()`, ví dụ

```
glColor3f(0.0, 0.0, 0.0);          // black
glColor3f(1.0, 0.0, 0.0);          // red
glColor3f(0.0, 1.0, 0.0);          // green
glColor3f(1.0, 1.0, 0.0);          // yellow
glColor3f(0.0, 0.0, 1.0);          // blue
glColor3f(1.0, 0.0, 1.0);          // magenta
glColor3f(0.0, 1.0, 1.0);          // cyan
glColor3f(1.0, 1.0, 1.0);          // white
```

2. Vẽ các đối tượng hình học

OpenGL không có sẵn các hàm để xây dựng các đối tượng hình học phức tạp, người dùng phải tự xây dựng chúng từ các đối tượng hình học cơ bản mà OpenGL hỗ trợ: điểm, đoạn thẳng, đa giác.

Khai báo một điểm, dùng hàm `glVertexXY` với X là số chiều (2, 3, hoặc 4), Y là kiểu dữ liệu (như đã nói ở chương 1).

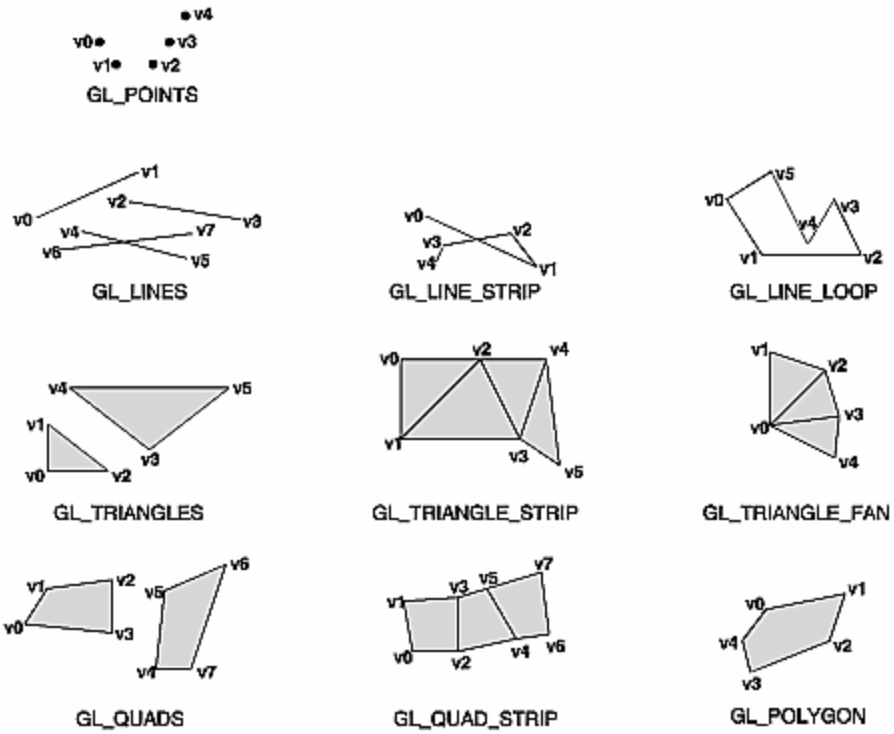
Việc xây dựng các đối tượng hình học khác đều có thể được thực hiện như sau

```
glBegin(mode);
/* xác định tọa độ và màu sắc của các điểm của hình */
glEnd();
```

`mode` có thể là một trong những giá trị sau

Giá trị	*Ý nghĩa*
GL_POINTS	individual points
GL_LINES	pairs of vertices interpreted as individual line segments
GL_LINE_STRIP	series of connected line segments
GL_LINE_LOOP	same as above, with a segment added between last and first vertices
GL_TRIANGLES	triples of vertices interpreted as triangles
GL_TRIANGLE_STRIP	linked strip of triangles
GL_TRIANGLE_FAN	linked fan of triangles
GL_QUADS	quadruples of vertices interpreted as four-sided polygons
GL_QUAD_STRIP	linked strip of quadrilaterals
GL_POLYGON	boundary of a simple, convex polygon

Hình sau minh họa cho các loại mode



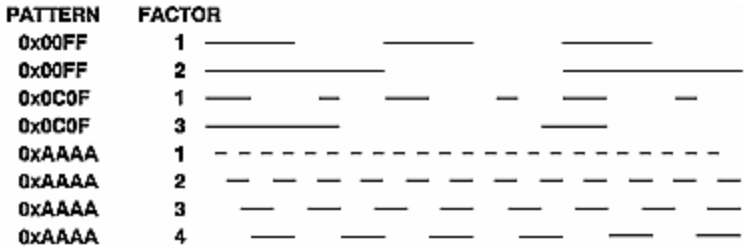
Ví dụ: vẽ hình chữ nhật màu trắng

```
glColor3f (1.0, 1.0, 1.0);  /*  thiết lập màu vẽ: màu trắng  */
glBegin(GL_POLYGON);   /*  bắt đầu vẽ đa giác  */
    glVertex3f (0.25, 0.25, 0.0);  /*  xác định các đỉnh của đa giác  */
    glVertex3f (0.75, 0.25, 0.0);
    glVertex3f (0.75, 0.75, 0.0);
    glVertex3f (0.25, 0.75, 0.0);
glEnd();  /*  kết thúc vẽ đa giác  */
```

Màu sắc thôi chưa đủ, một số tính chất của điểm và đoạn cần quan tâm có thể được thiết lập qua các hàm

- kích thước của một điểm: `void` **`glPointSize`**`(GLfloat size)`
- độ rộng của đoạn thẳng: `void` **`glLineWidth`**`(GLfloat` *`width`*`)`
- kiểu vẽ

```
glEnable(GL_LINE_STIPPLE);  // enable kiểu vẽ
glLineStipple(factor, pattern);  // pattern được cho trong bảng sau,
factor thường là 1
/*  thực hiện các thao tác vẽ  */
...
glDisable (GL_LINE_STIPPLE);  // disable kiểu vẽ
```



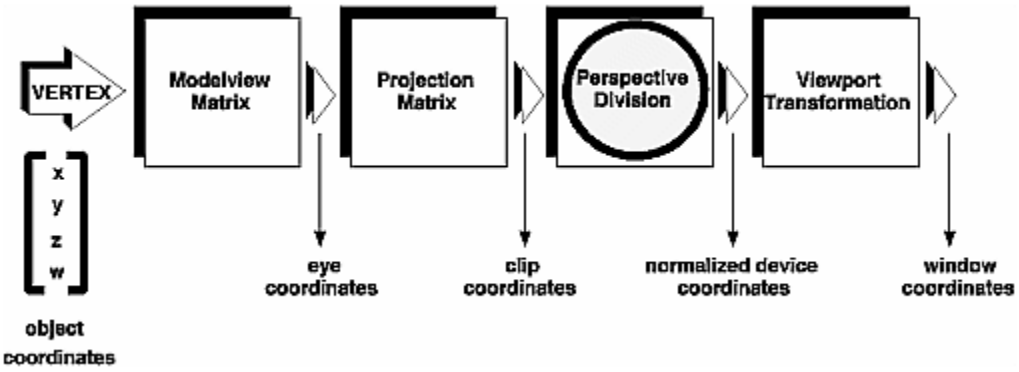
GLUT hỗ trợ sẵn một số hàm để vẽ các đối tượng hình học phức tạp hơn (đề nghị người đọc tự thử qua các hàm này)

```
void glutWireSphere(GLdouble radius, GLint slices, GLint stacks);
void glutSolidSphere(GLdouble radius, GLint slices, GLint stacks);
void glutWireCube(GLdouble size);
void glutSolidCube(GLdouble size);
void glutWireTorus(GLdouble innerRadius, GLdouble outerRadius, GLint nsides,
GLint rings);
void glutSolidTorus(GLdouble innerRadius, GLdouble outerRadius, GLint nsides,
GLint rings);
void glutWireIcosahedron(void);
void glutSolidIcosahedron(void);
void glutWireOctahedron(void);
void glutSolidOctahedron(void);
void glutWireTetrahedron(void);
void glutSolidTetrahedron(void);
void glutWireDodecahedron(GLdouble radius);
void glutSolidDodecahedron(GLdouble radius);
void glutWireCone(GLdouble radius, GLdouble height, GLint slices, GLint
stacks);
void glutSolidCone(GLdouble radius, GLdouble height, GLint slices, GLint
stacks);
void glutWireTeapot(GLdouble size);
void glutSolidTeapot(GLdouble size);
```

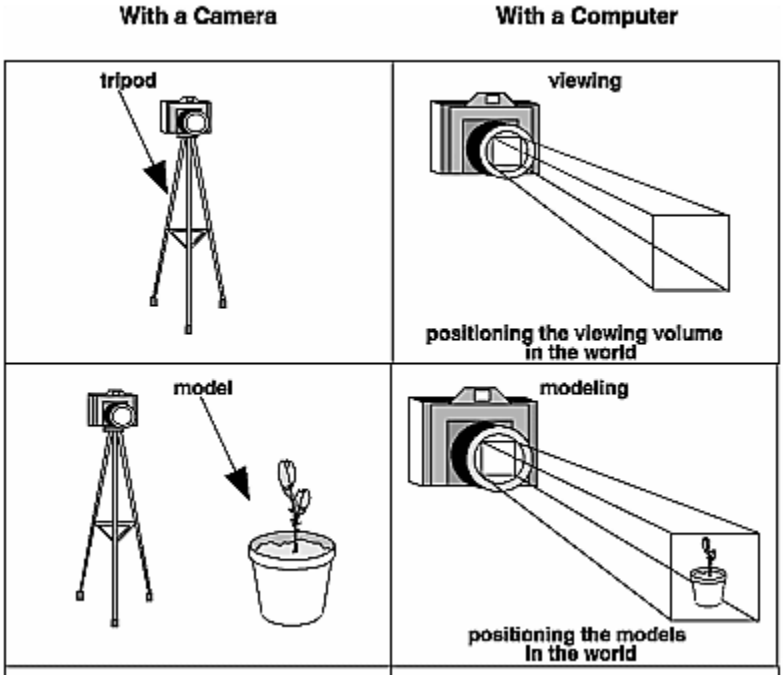
Chương 3: Các phép biến đổi

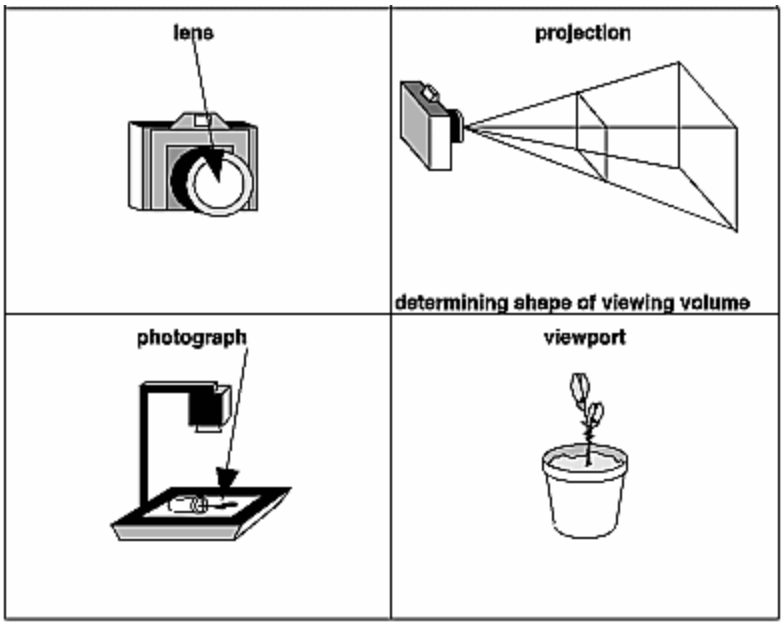
1. Giới thiệu

Trong OpenGL, tiến trình đi từ điểm trong không gian thế giới thực đến pixel trên màn hình như sau



Tương ứng với các thao tác trong chụp ảnh như sau





Trong OpenGL các điểm được biểu diễn dưới hệ tọa độ thuần nhất. Do đó, tọa độ của một điểm 3D được thể hiện bởi $(x,y,z,w)^T$, thông thường w = 1 (chú ý: *cách biểu diễn vector điểm ở đây là dạng cột*). Một phép biến đổi trên một điểm v tương ứng với việc nhân v với ma trận biến đổi M kích thước 4x4: $v' = M.v$.

Trong mỗi bước ModelView và Projection (chiếu), tại mỗi thời điểm, OpenGL đều lưu trữ một ma trận biến đổi hiện hành. Để thông báo với chương trình rằng sẽ thực thi bước ModelView, chúng ta cần phải gọi hàm

```
glMatrixMode(GL_MODELVIEW)
```

Tương tự, để thông báo cho bước Projection, chúng ta gọi hàm

```
glMatrixMode(GL_PROJECTION)
```

Để thiết lập ma trận biến đổi hiện hành bằng ma trận M, chúng ta dùng hàm sau

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

Chú ý: ma trận M có dạng

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

Vì một lí do nào đó chúng ta phải thay đổi ma trận hiện hành, nhưng sau đó chúng ta lại muốn khôi phục lại nó. Ví dụ như chúng ta dời tới một điểm nào đó để vẽ khối hộp, sau đó chúng ta muốn trở lại vị trí ban đầu. Để hỗ trợ các thao tác lưu trữ ma trận hiện hành, OpenGL có một stack cho mỗi loại ma trận hiện hành, với các hàm sau

- đẩy ma trận hiện hành vào trong stack: `void glPushMatrix(void)`
- lấy ma trận hiện hành ở đỉnh stack: `void glPopMatrix(void)`

2. Thao tác trên ModelView

Trước khi thực hiện các thao tác trên ModelView, chúng ta cần gọi hàm

```
glMatrixMode(GL_MODELVIEW);
```

2.1. Một số hàm biến đổi affine

OpenGL hỗ trợ sẵn các hàm biến đổi affine cơ bản như sau

- tịnh tiến
  ```
  void glTranslate{fd}(TYPEx, TYPE y, TYPEz);
  ```
- quay quanh trục nối gốc tọa độ với điểm (x,y,z)
  ```
  void glRotate{fd}(TYPE angle, TYPE x, TYPE y, TYPE z);
  ```
- tỉ lệ (tâm tỉ lệ tại gốc tọa độ)
  ```
  void glScale{fd}(TYPEx, TYPE y, TYPEz);
  ```

Với mục đích tổng quát hơn, việc nhân ma trận M có thể được thực thi bởi hàm

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

Chú ý:

- mọi thao tác biến đổi trên đều có nghĩa là lấy ma trận biến đổi hiện hành nhân với ma trận biến đổi affine cần thực hiện.
- thứ tự thực hiện sẽ ***ngược*** với suy nghĩ của chúng ta, ví dụ thứ tự thực hiện mà chúng ta nghĩ là: quay quanh trục z một góc α, sau đó tịnh tiến đi một đoạn (tr_x, tr_y, tr_z) thì sẽ được thực thi trong OpenGL như sau

  ```
  glTranslatef(trx, try, trz)
  glRotatef(α, 0, 0, 1)
  ```

  (giải thích: nguyên nhân của việc làm ngược này là do tọa độ được biểu diễn bằng vector cột – nhớ lại là $(AB)^T = B^T A^T$)

Ví dụ: chúng ta thực hiện phép quay quanh trục z một góc α và tịnh tiến đi một đoạn theo vector (tr_x, tr_y, tr_z), các bước thực hiện sẽ là

Thao tác	*Ma trận hiện hành*

Khởi tạo ban đầu
 	
`glMatrixMode(GL_MODELVIEW)`
 	
`glLoadIdentity()`
	$\begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & 1 \end{bmatrix}$

Tịnh tiến `glTranslatef(trx, try, trz)`	$\begin{bmatrix} 1 & & & tr_x \\ & 1 & & tr_y \\ & & 1 & tr_z \\ & & & 1 \end{bmatrix}$
Quay `glRotatef(α, 0, 0, 1)`	$\begin{bmatrix} \cos\alpha & -\sin\alpha & & tr_x \\ \sin\alpha & \cos\alpha & & tr_y \\ & & 1 & tr_z \\ & & & 1 \end{bmatrix}$

2.2. Thiết lập view

Giống như chụp hình, thiết lập view là thiết lập vị trí cũng như góc, hướng của camera. GLUT có một hàm giúp thiết lập view một cách nhanh chóng

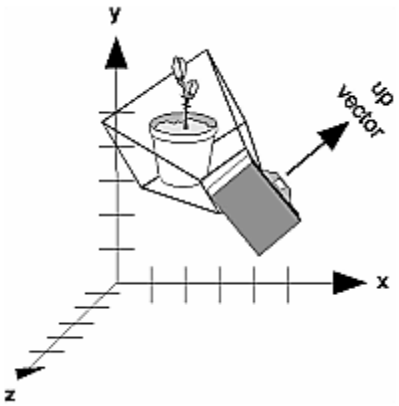
```
void gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble
centerx, GLdouble centery, GLdouble centerz, GLdouble upx, GLdouble
upy, GLdouble upz)
```

trong đó

- (eyex, eyey, eyez) là vị trí đặt của view,
- (centerx, centery, centerz) là điểm nằm trên đường thẳng xuất phát từ tâm view hướng ra ngoài,
- (upx, upy, upz) là vector chỉ hướng lên trên của view

Ví dụ:

- (eyex, eyey, eyez) = (4, 2, 1)
- (centerx, centery, centerz) = (2, 4, -3)
- (upx, upy, upz) = (2, 2, -1)



3. Thao tác trên Projection (Phép chiếu)

Trước khi thực hiện các thao tác chiếu, chúng ta gọi 2 hàm

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

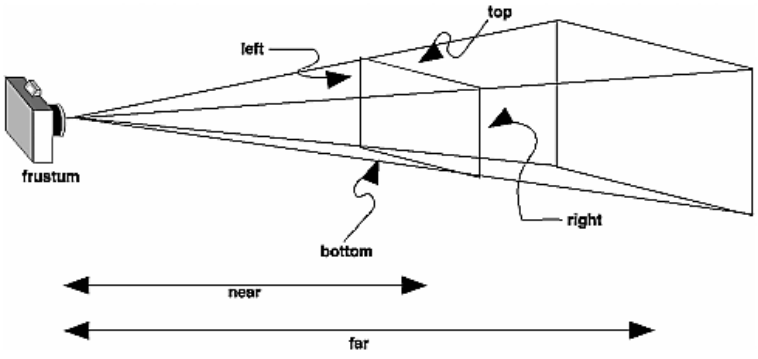
3.1. Chiếu phối cảnh (Perspective Projection)

Đặc điểm của phép chiếu này là đối tượng càng lùi ra xa thì trông càng nhỏ

Để thiết lập phép chiếu này, OpenGL có hàm

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

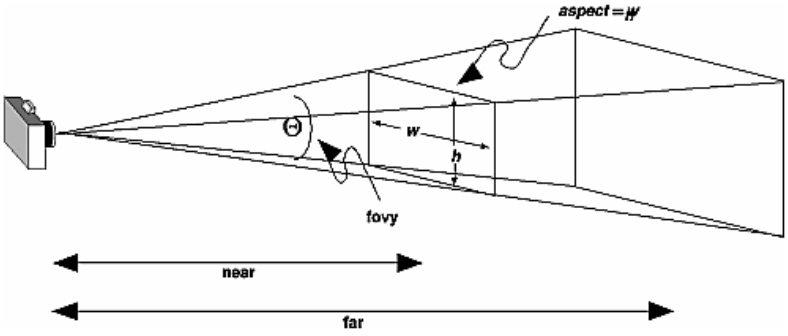
trong đó các tham số được thể hiện như hình dưới đây.



Ngoài ra, để dễ dàng hơn, chúng ta có thể sử dụng hàm

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

trong đó các tham số được miêu tả như hình dưới đây



(aspect = w/h).

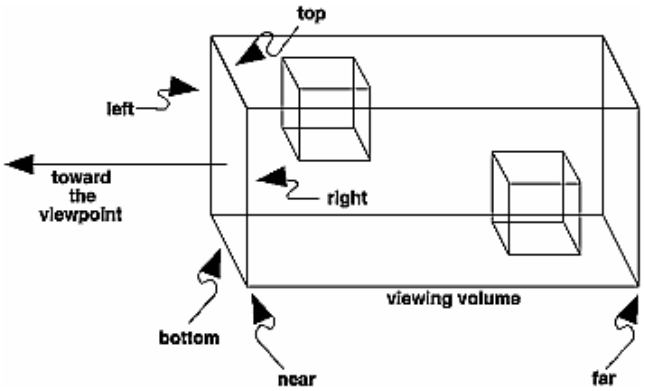
3.2. Chiếu trực giao (Orthogonal Projection)

Trong phép chiếu này, khoảng cách của vật tới camera không ảnh hưởng tới độ lớn của vật đó khi hiển thị.

Để thiết lập phép chiếu này, OpenGL có hàm

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

trong đó các tham số được thể hiện trong hình dưới đây.



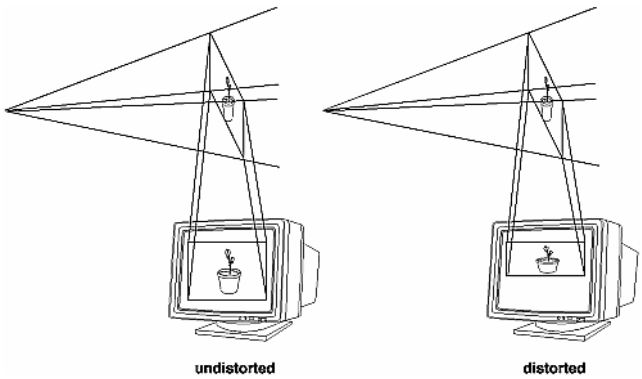
4. Thao tác trên Viewport

OpenGL có hàm để thiết lập viewport

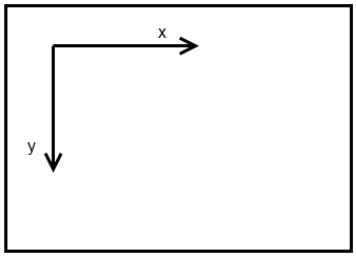
```
void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);
```

trong đó (x,y) là vị trí điểm trái-trên trong cửa sổ vẽ, width, height là chiều rộng và cao của viewport. Mặc định (x,y,width,height) = (0,0,winWidth, winHeight) (chiếm toàn bộ cửa sổ)

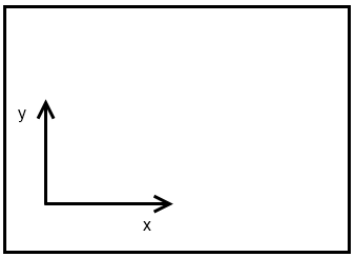
Hình sau minh họa việc thiết lập viewport.



Chú ý: lập trình trong môi trường Windows (ví dụ như dùng MFC), tọa độ trong cửa sổ thông thường được quy định như sau



Tuy nhiên, trong viewport, chúng ta cần phải quên quy ước đó đi, thay bằng



Lưu ý: khi bắt sự kiện mouse thì tọa độ trả về vẫn tuân theo quy tắc của Windows.

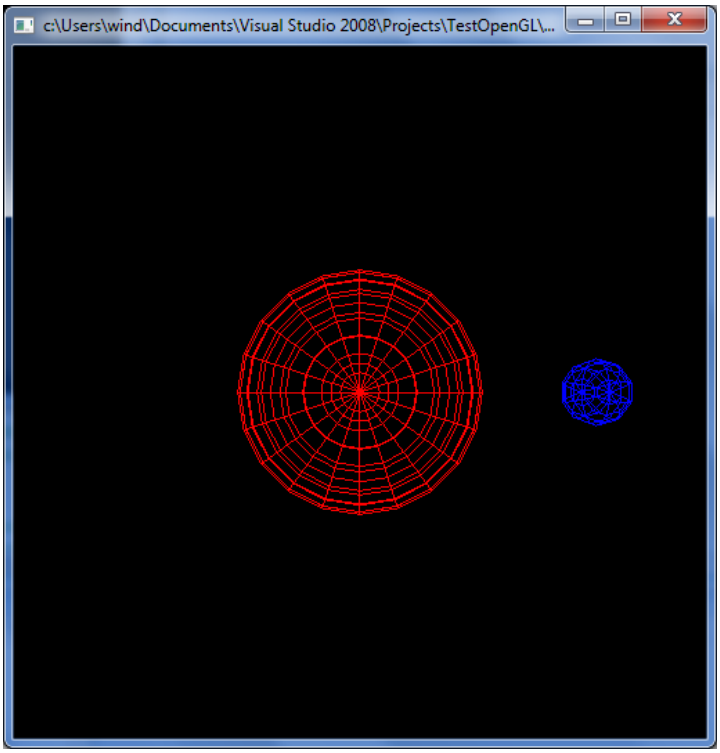
4. Ví dụ

Chúng ta xét ví dụ về xây dựng mô hình Trái Đất quay xung quanh Mặt Trời

```
#include "glut.h"

static int year = 0, day = 0; // thông số chỉ thời gian trong năm và thời
gian trong ngày để xác định vị trí của trái đất trên quỹ đạo và xác định góc
quay của nó quanh tâm

/* Khởi tạo */
void init(void)
{
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glEnable(GL_DEPTH_TEST);  // bật chức năng cho phép loại bỏ một phần của
đối tượng bị che bởi đối tượng khác
   glShadeModel (GL_FLAT);
}

/* hàm vẽ */
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // xóa color buffer và
depth buffer

   glPushMatrix(); // lưu lại ma trận hiện hành
   glColor3f (1.0, 0, 0); // thiết lập màu vẽ là màu đỏ
   glutWireSphere(1.0, 20, 16);   // vẽ mặt trời là một lưới cầu có tâm tại
gốc tọa độ

   /* di chuyển đến vị trí mới để vẽ trái đất */
   glRotatef ((GLfloat) year, 0.0, 1.0, 0.0); // quay một góc tương ứng với
thời gian trong năm
   glTranslatef (2.0, 0.0, 0.0); // tịnh tiến đến vị trí hiện tại của trái
đất trên quỹ đạo quanh mặt trời
   glRotatef ((GLfloat) day, 0.0, 1.0, 0.0); // quay trái đất tương ứng với
thời gian trong ngày
   glColor3f (0, 0, 1.0); // thiết lập màu vẽ là màu blue
   glutWireSphere(0.2, 10, 8);    // vẽ trái đất
   glPopMatrix();  // phục hồi lại ma trận hiện hành cũ: tương ứng với quay
lại vị trí ban đầu

   glutSwapBuffers();
}

/* xử lý khi cửa sổ bị thay đổi */
void reshape (int w, int h)
{
   glViewport (0, 0, (GLsizei) w, (GLsizei) h);  // thay đổi kích thước
viewport
/* xét thao tác trên chiếu */
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ();
   gluPerspective(60.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0); // thực hiện
phép chiếu phối cảnh

/* xét thao tác trên ModelView */
   glMatrixMode(GL_MODELVIEW);
```

```
   glLoadIdentity();
   gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0); // thiết lập view
}

/* xử lý sự kiện keyboard */
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);
   glutInitWindowSize (500, 500);
   glutInitWindowPosition (100, 100);
   glutCreateWindow (argv[0]);
   init ();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutKeyboardFunc(keyboard);
   glutMainLoop();
   return 0;
}
```

Kết quả khi chạy chương trình

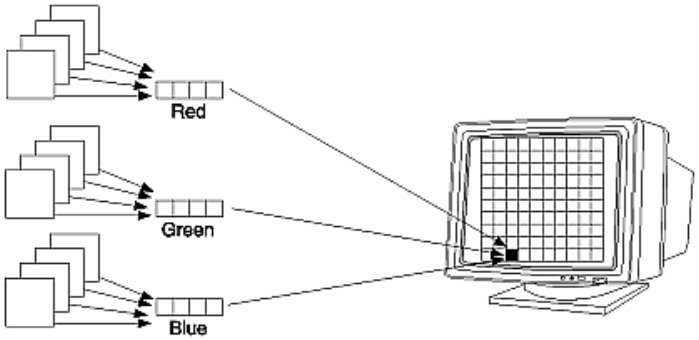

c:\Users\wind\Documents\Visual Studio 2008\Projects\TestOpenGL\...

Chương 4: Tô màu

1. Chế độ màu RGBA

OpenGL hỗ trợ 2 chế độ màu: RGBA và Color-Index. Ở trong tài liệu này, chúng ta chỉ quan tâm đến RGBA.

Trong chế độ màu RGBA, RGB lần lượt thể hiện màu Red, Green, Blue. Còn thành phần A (tức alpha) không thực sự ảnh hưởng trực tiếp lên màu pixel, người ta có thể dùng thành phần A để xác định độ trong suốt hay thông số nào đó cần quan tâm. Ở đây, chúng ta sẽ không bàn đến thành phần A này.



Để thiết lập màu vẽ hiện hành trong chế độ RGBA, chúng ta có thể sử dụng các hàm sau

```
void glColor3{b s i f d ub us ui} (TYPEr, TYPEg, TYPEb);
void glColor4{b s i f d ub us ui} (TYPEr, TYPEg, TYPEb, TYPEa);
void glColor3{b s i f d ub us ui}v (const TYPE*v);
void glColor4{b s i f d ub us ui}v (const TYPE*v);
```

trong đó, nếu các tham số là số thực thì thành phần màu tương ứng sẽ nằm trong đoạn [0,1], ngược lại thì sẽ được chuyển đổi như ở bảng sau

Suffix	Data Type	Minimum Value	Min Value Maps to	Maximum Value	Max Value Maps to
b	1-byte integer	-128	-1.0	127	1.0
s	2-byte integer	-32,768	-1.0	32,767	1.0
i	4-byte integer	-2,147,483,648	-1.0	2,147,483,647	1.0
ub	unsigned 1-byte integer	0	0.0	255	1.0

us	unsigned 2-byte integer	0	0.0	65,535	1.0
ui	unsigned 4-byte integer	0	0.0	4,294,967,295	1.0

2. Thiết lập mô hình shading

Một đoạn thẳng có thể được tô bởi một màu đồng nhất (chế độ *flat*) hay bởi nhiều màu sắc khác nhau (chế độ *smooth*). Để thiết lập chế độ shading phù hợp, chúng ta có thể sử dụng hàm

```
void glShadeModel (GLenum mode);
```

trong đó `mode` là chế độ mong muốn, nhận 1 trong 2 giá trị `GL_SMOOTH` hoặc `GL_FLAT`.

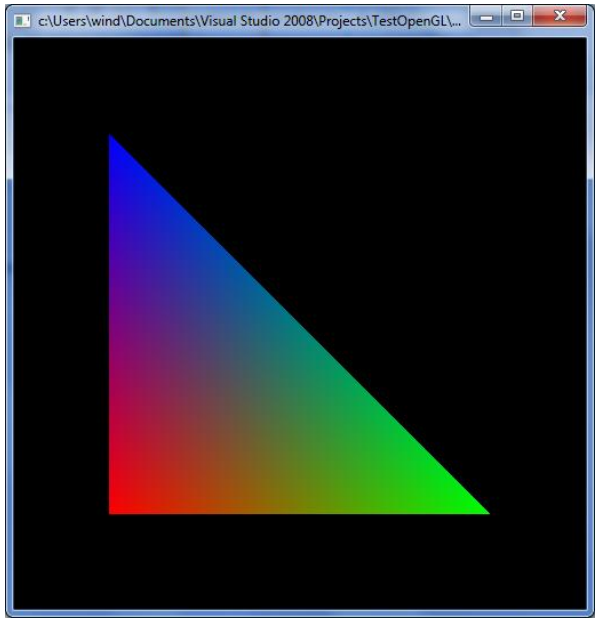
2.1. Chế độ smooth

Thông qua ví dụ sau chúng ta sẽ hiểu được chế độ smooth có tác động như thế nào

```
#include "glut.h"

void init(void)
{
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glShadeModel (GL_SMOOTH);  // thiết lập chế độ shading là smooth
}

void triangle(void)
{
   glBegin (GL_TRIANGLES);  // vẽ tam giác
   glColor3f (1.0, 0.0, 0.0);  // đỉnh thứ nhất màu red
   glVertex2f (5.0, 5.0);
   glColor3f (0.0, 1.0, 0.0); // đỉnh thứ 2 màu green
   glVertex2f (25.0, 5.0);
   glColor3f (0.0, 0.0, 1.0);  // đỉnh thứ 3 màu blue
   glVertex2f (5.0, 25.0);
   glEnd();
}

void display(void)
{
   glClear (GL_COLOR_BUFFER_BIT);
   triangle ();
   glFlush ();
}

void reshape (int w, int h)
```

```
{
   glViewport (0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ();
   if (w <= h)
      gluOrtho2D (0.0, 30.0, 0.0, 30.0*(GLfloat) h/(GLfloat) w);
   else
      gluOrtho2D (0.0, 30.0*(GLfloat) w/(GLfloat) h, 0.0, 30.0);
   glMatrixMode(GL_MODELVIEW);
}

int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize (500, 500);
   glutInitWindowPosition (100, 100);
   glutCreateWindow (argv[0]);
   init ();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutMainLoop();
   return 0;
}
```

Kết quả khi chạy chương trình



2.2. Chế độ flat

Như đã nói ở trên, chế độ flat tô hình đang xét một màu đồng nhất. Khi đó, OpenGL sẽ lấy màu của một đỉnh làm màu tô cho toàn bộ hình.

- Đối với đoạn thẳng, điểm đó là điểm cuối của đoạn,
- Đối với đa giác, điểm đó được chọn theo quy tắc trong bảng sau

Loại đa giác	Đỉnh được chọn để lấy màu cho đa giác thứ i
single polygon	1
triangle strip	i+2
triangle fan	i+2
independent triangle	3i
quad strip	2i+2
independent quad	4i

Tuy nhiên, cách tốt nhất để tránh lầm lẫn là thiết lập màu tô đúng 1 lần.

Chương 5: Tương tác với người dùng: chọn đối tượng

1. Giới thiệu

Việc cho phép người dùng chọn đối tượng bằng cách click chuột trên cửa sổ là một yêu cầu thiết yếu đối với các ứng dụng tương tác. Để thực hiện được những chức năng như vậy, trong OpenGL có sẵn một chế độ là Selection.

Có 2 công đoạn lớn chúng ta cần phải làm

1) Thực hiện các thao tác vẽ trong chế độ render (đây là điều mà 4 chương trước đã bàn tới)
2) Thực hiện các thao tác vẽ trong chế độ selection (giống hoàn toàn như trong công đoạn 1), kết hợp với một số thao tác đặc trưng trong chế độ selection.

Công đoạn 1 là các thao tác để biến đổi các đối tượng trong không gian về các pixel và sau đó hiển thị lên màn hình. Công đoạn 2, gần như ngược lại, chương trình xác định xem pixel mà người dùng tương tác (ví dụ như nhấn chuột trái) thuộc đối tượng nào.

Để chuyển đổi qua lại giữa các công đoạn (hay chế độ), chúng ta dùng hàm

```
GLint glRenderMode(GLenum mode);
```

trong đó mode là `GL_RENDER` hoặc `GL_SELECT` (mode còn có thể là `GL_FEEDBACK` nhưng ở đây chúng ta sẽ không xét tới).

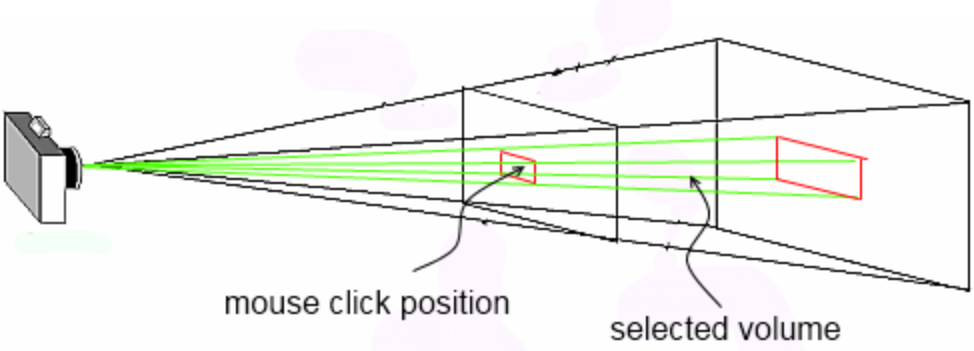
2. Các thao tác trên chế độ selection

Trước tiên chúng ta cần kích hoạt chế độ selection

```
glRenderMode(GL_SELECT)
```

2.1. Xác định vùng chọn

Ví dụ về chọn đối tượng bằng click chuột được cho như hình dưới đây



Việc xác định vùng chọn tương tự như là việc xác định khối nhìn, tức là chúng ta sẽ thao tác trên phép chiếu (projection – chương 3, mục 3).

Thao tác tổng quát như sau

```
glMatrixMode (GL_PROJECTION);
glPushMatrix ();
glLoadIdentity ();
gluPickMatrix (...);
gluPerspective, glOrtho, gluOrtho2D, or glFrustum
   /* ... */
glPopMatrix();
```

Trong đó,

```
void gluPickMatrix(GLdouble x, GLdouble y, GLdouble width, GLdouble
height, GLint viewport[4]);
```

là hàm xác định vùng quan tâm trong viewport (ví dụ như xung quanh vùng click chuột) với:

- `(x, y, width, height)` là tham số xác định quan tâm trên viewport
- `viewport[4]` là mảng 4 phần tử chứa 4 tham số của viewport, có thể dùng hàm `glGetIntegerv(GL_VIEWPORT, GLint *viewport)` để lấy ra.

2.2. Thiết lập đối tượng và danh tính cho đối tượng

Để phân biệt được các đối tượng với nhau, OpenGL cần phải đặt tên cho các đối tượng cần quan tâm. Việc đặt tên này có 3 điều đáng lưu ý

1) tên là một số nguyên,
2) các đối tượng có thể mang cùng tên: đây là các đối tượng được gom vào cùng một nhóm được quan tâm, ví dụ như nhóm các hình cầu, nhóm các hình khối hộp,…
3) tên có thể mang tính phân cấp, thể hiện ở đối tượng được cấu thành từ nhiều thành phần khác nhau. Ví dụ như khi click vào một cái bánh của cái một cái xe hơi, chúng ta cần biết là cái bánh số mấy của cái xe hơi số mấy.

OpenGL có một stack giúp thao tác trên tên các đối tượng, với các hàm

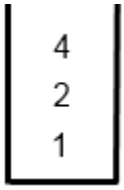
- `void` **`glInitNames`**`(void)` khởi tạo stack (stack lúc này rỗng)
- `void` **`glPushName`**`(GLuint name)` đặt tên của đối tượng cần xét vào trong stack
- `void` **`glPopName`**`(void)` lấy tên nằm ở đỉnh stack ra khỏi stack
- `void` **`glLoadName`**`(GLuint name)` thay nội dung của đỉnh stack

Việc sử dụng stack này giúp cho mục đích thứ 3 – xây dựng tên mang tính phân cấp. Mỗi lần thực thi `glPushName(name)` hoặc `glLoadName(name)` thì chương trình sẽ hiểu là các đối tượng được vẽ ở các dòng lệnh sau sẽ có tên là `name` và chúng là thành phần bộ phận của đối tượng có tên đặt ở ngay dưới đỉnh stack.

Ví dụ: xét đoạn mã giả sau

```
glInitNames();
glPushName(1);
/* vẽ đối tượng thứ nhất */
...
glPushName(2);
/* vẽ đối tượng thứ 2 */
...
glPushName(4);
/* vẽ đối tượng thứ 3 */
...
```

stack sẽ có nội dung sau



thì nghĩa là đối tượng (4) là thành phần của đối tượng (2), và đối tượng (2) là thành phần của đối tượng (1).

2.3. Truy vấn đối tượng trong vùng chọn

Để có thể truy vấn xem đối tượng nào được chọn, OpenGL xử lý như sau

- trước tiên sẽ đánh dấu mọi đối tượng nào có vùng giao với vùng chọn,
- sau đó, với mỗi đối tượng có vùng giao, tên của nó và giá trị z nhỏ nhất, z lớn nhất của vùng giao sẽ được lưu trong hit records
- mọi truy vấn về đối tượng được chọn sẽ được thực hiện trên hit records.

Như vậy, dựa trên hit records chúng ta biết được các thông tin sau

1) số recods = số lượng đối tượng cần quan tâm nằm trong vùng chọn
2) với mỗi record, chúng ta biết được các thông tin sau
   a. tên của đối tượng (bao gồm tên của tất cả các đối tượng mà nó là thành phần)
   b. z_min và z_max của vùng giao giữa đối tượng với vùng chọn (2 con số này nàm trong [0,1] và cần phải nhân với 2^{31}-1 (`0x7fffffff`)).

Để khởi tạo hit records, chúng ta cần phải gọi hàm

```
void glSelectBuffer(GLsizei size, GLuint *buffer)
```

trong đó buffer chính là mảng chứa hit records.

Chú ý: thủ thục này phải được gọi trước khi chuyển sang chế độ `GL_SELECT`.

4. Ví dụ

Trong ví dụ này, tương tự như ví của của chương 3 mục 4, chúng ta sẽ vẽ mô hình trái đất quay xung quanh mặt trời. Hơn nữa, chúng ta sẽ cho phép thực hiện các thao tác sau

- nếu người dùng click vào mặt trời thì mặt trời sẽ được vẽ bằng solid sphere thay vì là wire sphere
- nếu người dùng click vào trái đất thì trái đất sẽ được vẽ bằng solid sphere thay vì là wire sphere
- nếu người dùng click vào vùng không thuộc đối tượng nào thì cả trái đất và mặt trời sẽ được vẽ bằng hình mặc định ban đầu là wire sphere

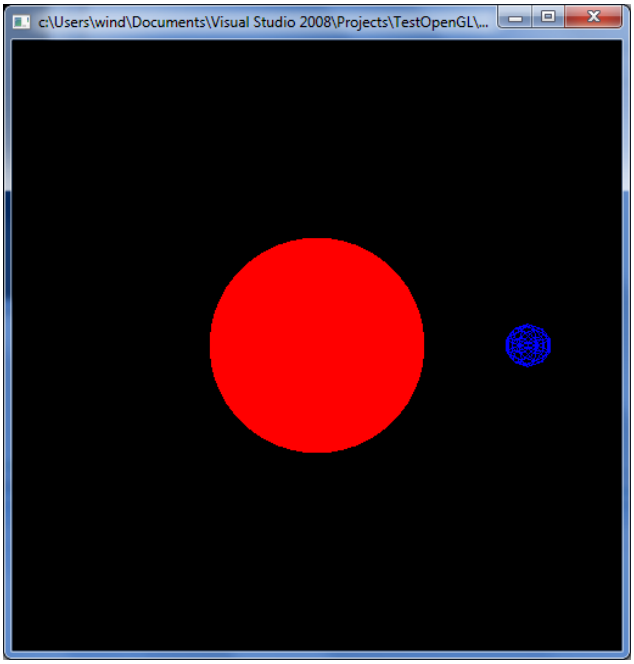
```
#include "glut.h"

#define NON -1
#define SUN 1
#define PLANET 2

static int year = 0, day = 0;
static int ichosen = NON;  // ghi lại xem đối tượng nào đang được chọn, NON
nghĩa là không có đối tượng nào hết

void init(void)
{
      glClearColor (0.0, 0.0, 0.0, 0.0);
      glEnable(GL_DEPTH_TEST);
      glShadeModel(GL_FLAT);
}
```

```
// hàm vẽ tổng quát cho cả chế độ render và chế độ selection
void draw(GLint mode)
{
	glMatrixMode(GL_MODELVIEW);
	glPushMatrix();
	glColor3f (1.0, 0, 0);
	if (mode == GL_SELECT)  // nếu đang là chế độ selection thì đặt tên cho
mặt trời
		glLoadName(SUN);
	if (ichosen == SUN) // nếu đang chọn SUN thì sẽ vẽ mặt trời khác đi
		glutSolidSphere(1.0, 50, 50);
	else
		glutWireSphere(1.0, 20, 16); // ngược lại thì vẽ như bình thường

/* di chuyển đến tọa độ mới để vẽ trái đất */
	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);

	glColor3f (0, 0, 1.0);
	if (mode == GL_SELECT) // nếu đang là chế độ selection thì đặt tên cho
mặt trời
		glLoadName(PLANET);
	if (ichosen == PLANET) // nếu trái đất đang được chọn thì sẽ vẽ khác đi
		glutSolidSphere(0.2, 30, 30);
	else
		glutWireSphere(0.2, 10, 8);

	glPopMatrix();
}

void display()
{
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
	draw(GL_RENDER);
	glutSwapBuffers();
}

// hàm xử lý hit records
void processHits (GLint hits, GLuint buffer[])
{
	unsigned int i, j;
	GLuint *ptr;
	float min_z_min;

	ptr = (GLuint *) buffer;
	ichosen = NON;

	/*  lặp với mỗi hit, trong trường hợp có nhiều hit thì sẽ chọn đối
tượng ở gần mắt nhất  */
	for (i = 0; i < hits; i++) {
		GLuint names = *ptr; ptr++;
		float z_min = (float) *ptr/0x7fffffff; ptr++; // giá trị z_min
của vùng giao đối tượng với vùng chọn
		float z_max = (float) *ptr/0x7fffffff; ptr++; // giá trị z_max
		GLuint name = *ptr;
		ptr++;
```

```
            if ( i == 0 || min_z_min > z_min )  // chọn đối tượng ở gần mắt
hơn
            {
                  min_z_min = z_min;
                  ichosen = name;
            }
      }
      ptr = (GLuint*) buffer;
}

#define BUFSIZE 512
// hàm xử lý thông điệp về mouse
void pick(int button, int state, int x, int y)
{
      GLuint selectBuf[BUFSIZE];
      GLint hits;
      GLint viewport[4];

/* chỉ xử lý khi người dùng click chuột trái */
      if (button != GLUT_LEFT_BUTTON || state != GLUT_DOWN)
            return;

      glGetIntegerv (GL_VIEWPORT, viewport);

      glSelectBuffer (BUFSIZE, selectBuf); // khởi tạo hit records
      (void) glRenderMode (GL_SELECT); // chọn chế độ selection

      glInitNames(); // khởi tạo stack tên
      glPushName(0); // đặt tên cho đối tượng rỗng là 0

/* thiết lập vùng chọn */
      glMatrixMode (GL_PROJECTION);
      glPushMatrix ();
      glLoadIdentity ();
      gluPickMatrix ((GLdouble) x, (GLdouble) (viewport[3]- y), 5.0, 5.0,
viewport); // vùng quan tâm vùng quanh mouse 5x5 pixel
      gluPerspective(60.0, viewport[2]/viewport[3], 1.0, 20.0);
      draw(GL_SELECT); // vẽ trong chế độ selection

      glMatrixMode (GL_PROJECTION);
      glPopMatrix ();
      glFlush ();

      hits = glRenderMode (GL_RENDER); // hits là số hit trong vùng chọn
      processHits (hits, selectBuf); // xử lý hit records
      glutPostRedisplay(); // bắt vẽ lại
}


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);
	glutInitWindowSize (500, 500);
	glutInitWindowPosition (100, 100);
	glutCreateWindow (argv[0]);
	init ();
	glutDisplayFunc(display);
	glutReshapeFunc(reshape);
	glutKeyboardFunc(keyboard);
	glutMouseFunc(pick); // thiết lập hàm pick xử lý thông điệp mouse
	glutMainLoop();
	return 0;
}
```

Kết quả khi click vào mặt trời


c:\Users\wind\Documents\Visual Studio 2008\Projects\TestOpenGL\...