Computer Graphics CS537
The Flying Camera Brothers

George Kamberov

Stevens Institute of Technology, Hoboken, NJ 07030, USA

READINGS: Angel 3d Edition, Chapter 4.10 and 4.11; Chapter 5 (in particular 5.3 and 5.7). The Red Book, Chapter 3. The Maths Reading modules.

There are two basic approaches to implement a moving camera: (i) Keep the world static and move the camera appropriately; (ii) Keep the camera static and move the world. The first approach seems to be more straight forward, since the second approach involves "inverting" the actual camera transformations. Most OpenGL-based implementations use the `gluLookAt` function to position the camera.

Approach i

```
void display(void )
{
   //Prepare for new rendering
   /**
    * Clear color, z-buffer, stencil buffer, etc
   **/
   //-------------Flying camera----------------/
   /**
    * Set camera optics/projection
   **/
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
         .
         .
         .
         .
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();

  /**Handle the flying camera movement.
```

```
   * The current eye/COP position i.e. the VRP is
   * stored in the array e[3]
   * The viewing direction (the VPN) is stored
   * in the array n[3].
   * The current reference point (the "at point")
   * is stored in the array a[3],
   *          a[i]=e[i]-n[i]
   * VUP is stored in the array v[3]. (Angel Chapter 5.3.)
   *
  **/
  gluLookAt(e[0],e[1],e[2],a[0],a[1],a[2],v[0],v[1],v[2]);
 //-------------EOFlying camera--------------------------/


    /**
     * Draw the world
    **/
}
```

The trick in using this approach is to compute correctly the current VRP, VUP, and ”look at” direction, or equivalently the current VRP, VUP, and ”at point”. The ”hard part” is to implement the pitch/roll/yaw ops. Each one of them is simply a rotation in a (camera) coordinate plane. The most important and often confused things to keep in mind are: (i) the viewing coordinate system $(\mathbf{u}, \mathbf{v}, \mathbf{n})$ is a right handed coordinate system; (ii) the axis $\mathbf{n}$ points in direction **opposite** to the look at direction, so to slide towards the looked at object on needs to decrement the $\mathbf{n}$ coordinate. To recall the Figures from the Maths modules, see Figure 1.

The basic navigation operations amount to changes of the viewing coordinate system $(\mathbf{u}', \mathbf{v}', \mathbf{n}')$. For example, pitching ”up” by $|\alpha|$ degrees (notice the absolute value here), that is, rolling about the current $\mathbf{u}$, is the same as a counterclockwise rotation by $|\alpha|$ degrees in the $\mathbf{vn}$ coordinate plane, and so changes the viewing coordinate system to

$$\begin{aligned}
\mathbf{u}' &= \mathbf{u} \\
\mathbf{v}' &= \cos(\alpha)\mathbf{v} - \sin(\alpha)\mathbf{n} \\
\mathbf{n}' &= \sin(\alpha)\mathbf{v} + \cos(\alpha)\mathbf{n}.
\end{aligned}$$

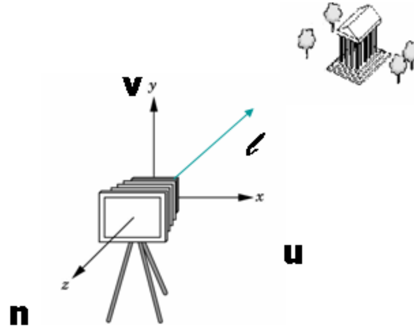

Fig. 1. The viewing coordinate system.

An α degrees clockwise (negative α means counterclockwise) yaw is the same as a rotation by α degrees in the **un** coordinate plane, and so it results in a new viewing coordinate system

$$\begin{aligned}
\mathbf{u}' &= \cos(\alpha)\mathbf{u} - \sin(\alpha)\mathbf{n} \\
\mathbf{v}' &= \mathbf{v} \\
\mathbf{n}' &= \sin(\alpha)\mathbf{u} + \cos(\alpha)\mathbf{n}.
\end{aligned}$$

Rolling (about the viewing axis **n**) is simply a rotation around the **n** and you should have no trouble deriving the necessary formulae following the ideas used to derive pitch and yaw.

You can control the camera by sliding it along the look at and the current **u** directions with the arrow keys and by controlling the pitch, yaw, and roll with the X/x, C/c, and Z/z keys. The small/caps toggle is used to distinguish positive (clockwise) from negative (counterclockwise) rotations.

```
void keyboard(unsigned char key, int x , int y)
{
  GLfloat xTemp, yTemp, zTemp;

  GLfloat cosine, sine;
 /**
```

```
 *
 * The values of the variables cosine and sine are plus/minus
 * the cosine and the sine of a small predetermined constant
 * angle.
 * Note that you have to adjust the signs depending on whether
 * the rotations are clockwise or counter-clockwise
 * (or equivalently depending on the case of the key cap/small)
**/

 // SET cosine and sine appropriately

 switch(key)
 {
 case 'x': // pitch up
 case 'X': // pitch down
   xTemp = v[0];
   yTemp = v[1];
   zTemp = v[2];
   v[0]=cosine*xTemp - sine*n[0];
   v[1]=cosine*yTemp - sine*n[1];
   v[2]=cosine*zTemp - sine*n[2];
   n[0]=sine*xTemp + cosine*n[0];
   n[1]=sine*yTemp + cosine*n[1];
   n[2]=sine*zTemp + cosine*n[2];
   break;
 case 'c': // yaw counter-clockwise in the un plane
           // (u towards n)
 case 'C': // yaw clockwise in the un plane
           // (from n towards u)
   xTemp = u[0];
   yTemp = u[1];
   zTemp = u[2];
   u[0]=cosine*xTemp - sine*n[0];
        .
        .
   n[0]=sine*xTemp + cosine*n[0];
        .
        .
   break;
```

```
  case 'z': // roll clockwise in the uv plane
  case 'Z': // roll counterclockwise in the uv plane
       .
       .
       .
    break;
  default:
    break;
  }
  glutPostRedisplay();
}
```

Sliding the camera is considerably easier to implement and I will leave it to you but be careful: In OpenGL the viewing coordinate system is right-handed, and $\mathbf{n}$ points "towards the back of the camera", i.e., away from the objects. As indicated earlier this means that one has to move the eye in direction of the negative VPN in order to move forward towards an object. So moving a step forward towards the object is achieved by

$$\mathbf{e} = \mathbf{e} - s\mathbf{n},$$

while

$$\mathbf{e} = \mathbf{e} + s\mathbf{n}$$

slides backwards. on the other hand sliding to "the right" is achieved by the assignment

$$\mathbf{e} = \mathbf{e} + s\mathbf{u}$$

and to slide to the left

$$\mathbf{e} = \mathbf{e} - s\mathbf{u}.$$

Here s is a small positive constant equal to the step size.

To add to the confusion especially when dealing with other people's code keep in mind that in many older systems the viewing coordinate system was left-handed to accommodate for the fact that in screen coordinates the y axis points downward.

All this may look too messy, but in fact it is pretty straightforward if one keeps his/her wits. Just be careful and double check everything. And of course there is the alternative Method ii.

Method ii

One can achieve the effect of a moving camera by repositioning the objects and keeping the camera position and orientation constant. Note that the movement of the world objects will be opposite to the desired camera motion. For example, to achieve the effect of a camera rotation by 45 degrees counterclockwise around the z-axis you simply rotate the objects by 45 degrees clockwise around the z axis. You have to worry about interpreting the keyboard/mouse input used to control the camera in appropriate displacement/traslation parameters of the objects. A simple OpenGL implementation of this approach is shown below

```
void display(void)
{
    //Prepare for new rendering

    /**
     * Clear color, z-buffer, stencil buffer, etc
    **/

    //-------------Flying camera----------------------------/
    /**
     * Set camera optics/projection = lens opening, zoom, etc
    **/
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45., W/H, 1, 400);

    /**
     * Position the camera
     * Method: Set a permanent camera position and orientation
     *         Move the scene appropriately so that the image
     *         will appear as if the scene was kept at the same
     *         position and orientation but the camera is
     *         moved
    **/
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0,0,0,0,0,1,-1,0,0);
    //Move the world w/r to camera
    glRotatef(roll, 0,0,1);
```

```
    glRotatef(pitch, 0,-1,0);
    glRotatef(heading, 0,0,1);
    retranslated(-X,-Y,-Z);
    //------------EOFlying camera----------------------/
    /**
     * Draw the world
    **/
}
```