An Introduction to 3D Computer Graphics, Stereoscopic Image, and Animation in OpenGL and C/C++

Fore June

Chapter 10 Create Stereoscopic Images with OpenGL

10.1 Multirendering using Accumulation Buffer

The accumulation buffer in OpenGL is one of the few buffers that help users processing graphics data more conveniently. It has the same spatial resolution as the frame buffer but the accumulation buffer has greater depth resolution. It is a higher precision buffer that can be used to accumulate intermediate rendering results. We can alter the viewing and projection matrices to provide multiple samples. These multiple samples result in multiple images, which are typically added into the accumulation buffer. Very often, the resulting accumulated images are scaled to produce an average filtered image. The following are the properties of the OpenGL accumulation buffer.

1. We may think of the accumulation buffer as a special color buffer that stores color values in floating point numbers and accumulates values.
2. Images are not rendered into it directly. Rather, images rendered into one of the color buffers are added to the contents of the accumulation buffer after rendering. Special graphics effects such as antialiasing, motion blur, and depth-of-field can be created by accumulating images generated with different transformation matrices.
3. It helps us to do image processing in a convenient way.

The function that operates on the accumulation buffer is **glAccum**(), which works as follows.

void **glAccum** (GLenum *op*, GLfloat *mult*);

op Specifies the accumulation buffer operation. Valid symbolic constants include GL_ACCUM, GL_LOAD, GL_ADD, GL_MULT, and GL_RETURN.

mult Specifies a floating-point value used in the accumulation buffer operation. The first parameter *op* determines how this value is used.

The operations specified by the parameter *op* are as follows:

GL_ACCUM
The operation obtains integer values of R, G, B, and A from the buffer currently selected for reading. Each component value is divided by $2^n - 1$, where n is the number of bits allocated to each color component in the currently selected buffer. This results in a floating-point number in the range $[0, 1]$; this number is multiplied by the parameter *mult* and added to the corresponding pixel component in the accumulation buffer, thereby updating the accumulation buffer.

GL_LOAD
The operation is similar to GL_ACCUM, except that it does not use the current value in the accumulation buffer to calculate the new value. Here, the integer values of R, G, B, and A from the currently selected buffer are divided by $2^n - 1$, multiplied by *mult*, and then stored in the corresponding accumulation buffer location, overwriting the current value.

GL_ADD
The operation adds *mult* to each of the R, G, B, and A components in the accumulation buffer.

GL_MULT
The operations multiplies each of the R, G, B, and A components in the accumulation buffer by *mult* and returns the scaled component to its corresponding accumulation buffer location.

GL_RETURN
The operation transfers accumulation buffer values to the color buffer or buffers currently selected for writing. It multiplies each R, G, B, and A component by *mult*, then multiplies the product by $2^n - 1$, clamps the result to the range $[0, 2^n - 1]$, and stores the result in the corresponding display buffer location. The only fragment operations that are applied to this transfer are pixel ownership, scissor, dithering, and color writemasks.

We can 'clear' the accumulation buffer with specified color components using the functions **glClearAccum**() and **glClear**() (with accumulation buffer enabled). The function **glClearAccum** (float R, float G, float B, float A) set the R, G, B, and A values when the accumulation buffer is cleared.

The following code section shows an example of using this function to display four images at the same time:

```
glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_ACCUM );
......

glClear(GL_ACCUM_BUFFER_BIT);
for ( int i = 0; i < 4; i++ ) {
   glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);//clear screen
   draw_image( i );            //construct i-th image
   glAccum(GL_ACCUM, 0.25);    //scale each image by 0.25
}
glClear( GL_COLOR_BUFFER_BIT );//clear screen
glAccum(GL_RETURN, 1.0);        //render the accumulated image
```

Example 10-1 Blending Colors of 3 Triangles

In this example, we make use of the accumulation buffer to blend the colors of three partially overlapped triangles. The function **glRotatef**() is used to slightly offset each triangle so that they only partially overlap with each other. The following is the complete listing of the program.

```
/*
  accumtest.cpp :
  Partially overlap three triangles with blending colors
  using accumulation buffer.
*/
#include <GL/glut.h>

//initialization
void init( void )
```

```
{
  glClearColor( 1.0, 1.0, 1.0, 0.0 ); //get white background color
  glMatrixMode( GL_PROJECTION );
  glLoadIdentity();
  gluOrtho2D( 0.0, 400.0, 0.0, 400.0 );
}

void draw_triangle()
{
  glBegin( GL_TRIANGLES );
    glVertex2i( 100, 100 );
    glVertex2i( 100, 300 );
    glVertex2i( 300, 300 );
  glEnd();

}

void setColor( int i )
{
  switch ( i ){
  case 0:
    glColor3f( 1, 0, 0 );  break;
  case 1:
    glColor3f( 0, 1, 0 );  break;
  case 2:
    glColor3f( 0, 0, 1 );  break;
  default:
    glColor3f( 0, 0, 0 );  break;
  }
}

void display( void )
{
  glClear(GL_ACCUM_BUFFER_BIT);
  for ( int i = 0; i < 3; i++ ) {
    glClear(GL_COLOR_BUFFER_BIT);//clear screen
    glPushMatrix();
    //slightly displace each triangle
    glRotatef ( i*5.0, 0, 0, 1 );
    setColor ( i );
    draw_triangle();
    glPopMatrix();
    glAccum(GL_ACCUM, 0.33);     //accumulate color data
  }
  glClear(GL_COLOR_BUFFER_BIT);  //clear screen
  glAccum(GL_RETURN, 1.0);//render data saved in accumulation buffer

  glFlush();                     //send all output to screen
}

int main(int argc, char** argv)
{
  glutInit(&argc, argv);       //initialize toolkit
  //set display mode with accumulation buffer operations
  glutInitDisplayMode (GLUT_RGB | GLUT_ACCUM);
  glutInitWindowSize( 400, 400);      //set window size on screen
  glutInitWindowPosition( 500, 250 ); //set window position on screen
  glutCreateWindow(argv[0]);          //open screen window
  init();
  glutDisplayFunc (display);          //points to display function
```

```
    glutMainLoop();                    //go into perpetual loop
    return 0;
}
```

The main accumulating task is done in the call-back function **display**(), where the command **glAccum**(GL_ACCUM, 0.33) accumulates the data in the accumulation buffer. Inside the for-loop, whenever the function **draw_triangle**() is called, data are written into the color buffer and at the same time the normalized color data are multiplied by 0.33 and added to the original data in the accumulation buffer. This is essentially a color-blending process. When the program exits the for-loop, the color buffer is cleared by the command **glClear**(GL_COLOR_BUFFER_BIT). The next statement **glAccum**(GL_RETURN, 1.0) sends the data to the color buffer without further modification as the scaling factor is 1.0 (second parameter of glAccum) and thus render them on the screen. Figure 10-1 below shows the output of this code.

Figure 10-1 Color Blending using Accumulation Buffer

10.2 OpenGL Stereo

The above section describes the general method of using the accumulation buffer to superimpose different images and render them simultaneously. This is the general technique of creating stereo pairs. We create one image for the left eye and one for the right eye and accumulate both images in the buffer for superimposition. Nowadays, some system may offer special hardware to facilitate the process. In general, there are four types of OpenGL-based stereo viewing systems:

1. **Hardware with OpenGL stereo support.** This makes creating stereo pairs a lot easier. We may initialize the GLUT library for stereo operation with a code section like the following:

   ```
   if (stereo)
     glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB|GLUT_DEPTH|GLUT_STEREO);
   else
     glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
   ```

   In stereo mode, this defines two buffers, namely GL_BACK_LEFT and GL_BACK_RIGHT. We need to select the appropriate buffer for operations. For example, the following code clears the two buffers:

```
glDrawBuffer(GL_BACK_LEFT);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
if (stereo) {
  glDrawBuffer(GL_BACK_RIGHT);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
}
```

2. **Hardware without OpenGL stereo support**. This is used by software that does not know about stereo. These are the low-cost stereo devices targeting consumers, mostly 3D games. They work by using library wrappers that intercept the 3D information sent to OpenGL by the application and convert it to left and right views before displaying the data needed to work with the hardware. These drivers mostly do not work for applications that know about stereo; this may cause a lot of confusion for people trying to use them with true stereo programs.

3. **Hardware without OpenGL support**. This is used by software that knows how to directly produce stereo for a given system without any special hardware support. This may involve using a low-cost stereo device like one of those mentioned in 2. This may suffer from the problem of platform-dependent coding.

4. **Simulated OpenGL stereo support**. This is for software that knows about stereo. This approach allows one to use real stereo applications (as in case 1) on low cost hardware (as in case 2) without the application having to do all the work when porting the application from one platform to another one.

10.3 3D Stereo Rendering Using OpenGL

10.3.1 Using glColorMask

In this section, we discuss the general technique to render 3D stereo images without any special hardware support. The following are the general steps of using OpenGL to create 3D stereo images.

1. Use the function **glColorMask**() to handle colors of different situations of the scene. This function has the following prototype.

   void glColorMask(GLboolean red, GLboolean $green$, GLboolean $blue$, GLboolean $alpha$);

   The function specifies whether red, $green$, $blue$, and $alpha$ can be written into the frame buffer. The default values are all GL_TRUE meaning that all R, G, B, and A components can be written into the buffer. For instance, if we want to filter out the blue component, we simply set the $blue$ variable to GL_FALSE.

2. Create the scene with all the surfaces colored with pure white.

3. Render the scene twice with different colors, once for each eye.
   Suppose we use red color for the left eye, and blue color for the right eye. Then we can do the following to create the two colored scenes.

   a) Call

   glColorMask (GL_TRUE, GL_FALSE, GL_FALSE, GL_FALSE);

before rendering the left-eye scene so that the scene is drawn with red color only.

b) Call

glColorMask (GL_FALSE, GL_FALSE, GL_TRUE, GL_FALSE);

before rendering the right-eye scene so that the scene is drawn with blue color only.

4. Render the scene twice and use the accumulation buffer to superimpose the two images. (If the OpenGL hardware supports stereo buffers, then the above steps can be implemented directly without using the accumulation buffer.)

10.3.2 Using Accumulation Buffer

If the hardware of our system does not support stereo buffers, we may use the accumulation buffer to merge the two images created for the left and right eyes. Alternatively, we may use the color blending techniques that we have discussed in Chapter 5 to combine the images. However, the accumulation buffer technique is more flexible and provides more image processing functions. The following are the typical steps of using this method:

1. Initialize the accumulation buffer in glut:

   glutInitDisplayMode (GLUT_DOUBLE | GLUT_ACCUM | GLUT_RGB | GLUT_DEPTH);

2. Set the clear-colour for the accumulation buffer:

   glClearAccum (0.0, 0.0, 0.0, 0.0);

3. Clear the accumulation buffer when necessary:

   glClear (GL_ACCUM_BUFFER_BIT);

4. Copy the current drawing buffer to the accumulation buffer. We usually do this after the left eye image has been drawn:

   glAccum (GL_LOAD, 1.0);

5. Add the current drawing buffer, which now contains the right eye image, to the accumulation buffer:

   glAccum (GL_ACCUM, 1.0);

6. Copy the accumulation buffer content to the current drawing buffer:

   glAccum (GL_RETURN, 1.0);

10.3.3 Toe-in Projection Method

We have discussed in the previous chapter the principles of the toe-in method in creating stereo image pairs. Figure 9-9 of the chapter shows the setup of this method, in which both the left and right eyes point towards a single focal point. Here we discuss the implementation of this method. We shall use the function **gluPerspective**() to do the projection of the two images as both the left and right eyes will have the same values of *viewing angle* (field of view), *aspect ratio, near distance*, and *far distance*.

As we did in Chapter 7, we use a *Point3* class for handling positions, and use a *Vector3* class for handling vectors in 3D space. Their declarations are straight forward and similar;

each class defines the public data members **x, y, z** to describe the coordinates of a point or the components of a vector, (x, y, z). Details of these classes are defined and discussed in Chapter 12. Here, for clarity, we just use a simplified version of each of those classes.

Since we have two eye positions to deal with, we define a *Camera* class that can handle different viewpoints and different orientations:

```
class Camera {
public:
   Point3 p;            // View position
   Point3 focus;        // Point at which camera focuses
   Vec3   v;            // View direction vector
   Vec3  up;            // View up direction
   double f;            // Focal Length along v
   double fov;          // Camera aperture (field of view)
   double es;           // Eye separation
   int    w;            // Viewplane width
   int    h;            // Viewplane height

   //constructors
   Camera ()
   {
      // default parameters
      f = 10.0;
      es = f / 30;
      fov = 60;
      w = 400;
      h = 300;
      p = Point3 ( 0, 0, 10 );  //viewpoint
      focus = Point3( 0, 0, 0); //focus
      v = Vec3 ( 0, 0, 1 );     //view direction
      up = Vec3 ( 0, 1, 0 );    //up vector
   }

   void setCamera(double f0, double es0, double fov0, int w0, int h0)
   {
     f  = f0;
     es = es0;
     fov = fov0;
     w  = w0;
     h  = h0;
   }

   void lookAt ()
   {
     gluLookAt(p.x,p.y,p.z, focus.x,focus.y,focus.z, up.x,up.y,up.z);
   }

   void lookAt(const Point3 &eye,const Point3 &focus,const Vec3 &up0)
   {
     p = eye;
     up = up0;
     gluLookAt(p.x,p.y,p.z, focus.x,focus.y,focus.z, up.x,up.y,up.z);
   }
};
```

In the *Camera* class, we define two **lookAt**() functions which work in the same way as the OpenGL **glutLookAt**() function. The first one, which does not take any input parameters, uses the data members to set up the camera orientation. The second one, which takes three input parameters, uses the input parameters and calls **glutLookAt**() to do the setup.

Listing 10-1 below lists the code segment that creates a stereo pair of a wireframe cube and solid teapot using this toe-in method. In the example, the function **drawScene**() does the job of creating the graphics scene; it is independent of the way we render it. The camera is first setup in the **init**() routine using the parameters common to both the left and right eye positions. The two images are rendered in the callback function **display**(). The camera first moves to the left eye position, takes a 'shot' with a red 'filter', scales and saves the data in the accumulation buffer. It then moves to the right eye position and takes a 'shot' using a blue 'filter'. The positions are calculated according to Equation (9.6). That is, the left-eye location E_l, and the right-eye location E_r are given by

$$\begin{aligned} &\text{Left eye position} \quad E_l = (-\tfrac{e}{2}, 0, 0) \\ &\text{Right eye position} \quad E_r = (+\tfrac{e}{2}, 0, 0) \end{aligned} \tag{10.1}$$

where e is the eye separation and we assume that the mid-point between the eyes is at $(0, 0, 0)$. However, in the program we assume the mid-point is at $(0, 0, d)$ implying that the origin $(0, 0, 0)$ is at the center of the projection planes and the mid-point is at a distance d from the center.

The two images are superimposed in the accumulation buffer and rendered with the command "**glAccum**(GL_RETURN,1.0);" at the end. We have used a black background implying that the color components are all zero. Therefore, when we "accumulate" the scene, the background will not contribute any value in the process. Also, the color is filtered by a color-filter, the background will not have any effect as zero component remains to be zero. Figure 10-2 shows the output of this program.

Program Listing 10-1: Toein Projection Implementation (toein.cpp)

```
/*
   Create the scene
*/
void drawScene(void)
{
  glPushMatrix();
  glRotatef ( 20, 1, 0, 0 );
  glRotatef ( 45, 0, 1, 0 );
  glLineWidth ( 3 );
  glutWireCube( 4 );
  glPopMatrix();
  glPushMatrix();
  glTranslatef ( 1, 0, 4 );
  glRotatef ( 25, 0, 1, 0 );
  glutSolidTeapot ( 1 );
  glPopMatrix();
}

Camera camera;

void init(void)
{
   glEnable(GL_DEPTH_TEST);
   glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);
   glFrontFace(GL_CW);
   glClearColor ( 0, 0, 0, 0.0 );     //black background
   glClearAccum ( 0.0, 0.0, 0.0, 0.0 );
```

```
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    //setup camera
    double focalLength = 15;
    double eyeSeparation = focalLength / 30.0;
    double fov = 60;
    camera.setCamera ( focalLength, eyeSeparation, fov, 400, 300 );
    Point3 focus ( 0, 0, 0 );
    Vector3 up ( 0, 1,  0 );
    camera.focus = focus;
    camera.up = up;
    double aspectRatio = (double) camera.w / camera.h;
    gluPerspective( camera.fov, aspectRatio, 0.1, 10000.0 );
}

void display(void)
{
    // Set the buffer for writing and reading
    glDrawBuffer(GL_BACK);
    glReadBuffer(GL_BACK);

    // Clear things
    glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glClear ( GL_ACCUM_BUFFER_BIT );
    // Left eye  projection
    glMatrixMode(GL_MODELVIEW);
    glDrawBuffer(GL_BACK_RIGHT);
    glLoadIdentity();
    // setup camera for left eye
    double d = 10;
    Point3 viewPoint ( -camera.es / 2, 0, d );
    camera.p = viewPoint;                //change camera viewpoint
    camera.lookAt();

    // Left eye filter
    glColorMask(GL_TRUE,GL_FALSE,GL_FALSE,GL_TRUE);

    drawScene();
    // Write over the accumulation buffer
    glAccum( GL_LOAD, 1.0 );
    glDrawBuffer(GL_BACK);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    //right side projection
    glMatrixMode(GL_MODELVIEW);
    glDrawBuffer(GL_BACK_LEFT);
    glLoadIdentity();
    viewPoint.x = camera.es / 2;
    camera.p = viewPoint;                //change camera viewpoint
    camera.lookAt();
    // Right eye filter
    glColorMask(GL_FALSE,GL_FALSE,GL_TRUE,GL_TRUE);
    drawScene();
    glFlush();
    // Addin the new image and copy the result back
    glAccum(GL_ACCUM, 1.0);

    // Allow all colors
    glColorMask(GL_TRUE,GL_TRUE,GL_TRUE,GL_TRUE);
```

```
        glAccum(GL_RETURN,1.0);
        glutSwapBuffers();
    }
```

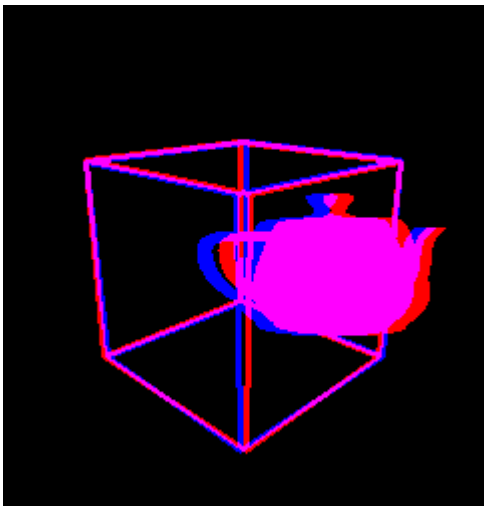

Figure 10-2 Anaglyph Created Using Toe-in Method

10.3.4 Two-center Projection Method

The two-center or the off-axis projection, where the view vector for each eye position remains parallel as shown in Figure 9-10, is the correct method of creating proper anaglyph pairs. We have discussed in Section 9.4.3 the general technique to calculate the projected points on the view plane. However, in our implementation, we do not calculate the projected points directly. We make use of the OpenGL projection functions to do the calculations for us. Since the view vector for each eye remains parallel, we use the OpenGL function **glFrustum**() to describe the perspective projection. Our task is to make the proper setup to use **glFrustum**().

Consider Figure 10-3 that shows the two-center projection (see also Figure 9-13 of Chapter 9). Suppose we let $\overline{AB} = d_{left}$ and $\overline{BD} = d_{right}$. We can make use of similar triangles to find the parameters to call **glFrustum**() for each of the eyes.

In this method, both the left and right eyes use the same projection plane. However, the projected images will be mapped to different near-planes which will define two different frustum as shown in Figure 10-3. In this way, the fields of view for both eyes are the same. Let us first consider the case of the left eye. Referring to the figure, we let

$$\begin{array}{ll} L = left = -\overline{AB} = -d_{left} & R = right = \overline{BD} = d_{right} \\ B = bottom & T = top \\ N = neardistance & F = fardistance \\ f = \text{focal length of camera} & \theta = \text{field of view} \\ e = \text{eye separation} & \rho = \text{aspect ratio} = \dfrac{width(W)}{height(H)} \end{array} \tag{10.2}$$

where W and H are the width and height of the near-plane, and L, R, B, T denote the left, right, bottom and top boundary coordinates of the near-plane respectively. As we usually do, we also assume that the near-plane lies in the x-y plane with y-axis pointing upward. These imply that

$$\begin{aligned} T &= N\tan\frac{\theta}{2} \\ B &= -T \\ H &= T - B = 2T \end{aligned} \tag{10.3}$$

Applying similar triangle properties, we can calculate the half-width a of the projection plane of Figure 10-3. First we notice that

$$\frac{2a}{f} = \frac{W}{N} \tag{10.4}$$

Therefore,

$$\frac{a}{f} = \frac{W}{2N} = \frac{\rho \times 2T}{2N} = \rho \times \frac{T}{N} = \rho \times \tan\frac{\theta}{2} \tag{10.5}$$

From (10.5), we obtain

$$a = f \times \rho \times \tan\frac{\theta}{2} \tag{10.6}$$

Once a is known, we can determine the distances b and c of Figure 10-3 by:

$$\begin{aligned} b &= a - \frac{e}{2} \\ c &= a + \frac{e}{2} \end{aligned} \tag{10.7}$$

From similar triangles, we have

$$\frac{d_{left}}{b} = \frac{N}{f} = \frac{d_{right}}{c} \tag{10.8}$$

Also,

$$\begin{aligned} L &= -d_{left} = -b \times \frac{N}{f} \\ R &= d_{right} = c \times \frac{N}{f} \end{aligned} \tag{10.9}$$

Combining (10.7), (10.8), and (10.9), we obtain

$$\begin{aligned} L &= -b \times \frac{N}{f} = -a \times \frac{N}{f} + \frac{e}{2} \times \frac{N}{f} = -\rho \times \frac{H}{2} + \frac{e}{2} \times \frac{N}{f} \\ R &= c \times \frac{N}{f} = a \times \frac{N}{f} + \frac{e}{2} \times \frac{N}{f} = \rho \times \frac{H}{2} + \frac{e}{2} \times \frac{N}{f} \end{aligned} \tag{10.10}$$

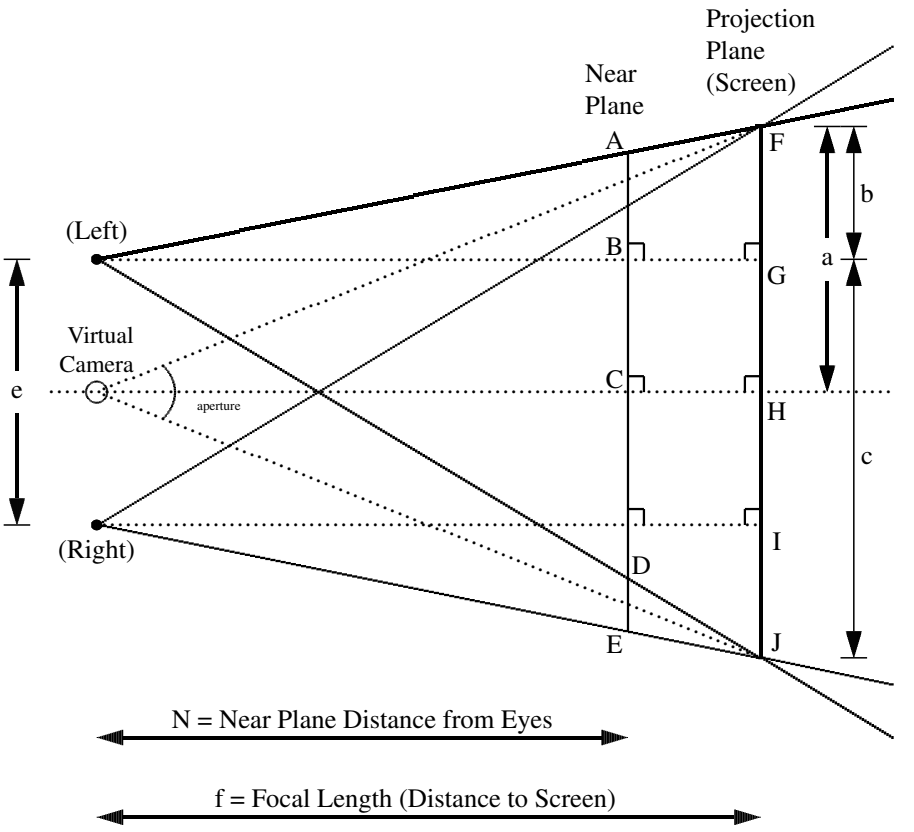

Figure 10-3 Off-axis Projection Calculations

The variables L, R, T, and B are used as the input parameters for the **glFrustum**() function of the left eye. Using the same code notations that we have used in the toein method implementation, we can implement this using a code section like the following.

```
/*
  L = left, R = right, T = top, B = bottom, N = near, F = far,
  f = focal length, e = eye separation, theta2 = (field of view)/2
  a = half-width of projection plane, ratio = aspect ratio
*/
double L, R, T, B, N, F, e, f, theta2, a, b, c, ratio;
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
...          //set N and F to near and far distances
f = camera.f;       // focal length
e = camera.es;      // eye separation
theta2 = (3.1415926/180) * camera.fov / 2;  //theta / 2 in radians
ratio  = (double) camera.w / camera.h;
a = f * ratio * tan ( theta2 );
b = a - e / 2.0;
c = a + e / 2.0;
T = N * tan ( theta2 );
B = -T;
L = -b * N / f;
R =  c * N / f;
glFrustum( L, R, B, T, N, F );
```

```
//viewpoint = (-e/2, 0, f), focus = (-e/2, 0, 0), up=(0, 1, 0)
gluLookAt ( -camera.es/2, 0, f, -camera.es/2, 0, 0, 0, 1, 0 );
```

The above formulas and code are for the left eye. We have assumed that the origin $(0, 0, 0)$ is at the center of the projection plane; the eye is looking along the $-z$ direction and is at a distance f from the projection plane.

We can similarly obtain the equations and code for the right eye. If the center of the eye separation has x-y coordinates $(0, 0)$, then there are some symmetries between the two eyes and the right eye parameters can be obtained directly from those of the left eye. Firstly, we note that the top and bottom boundaries of the near-plane are the same for both eyes. Secondly, we note that the value of the left boundary for the right eye is equal to the negative value of the right boundary for the left eye and the value of the right boundary for the left eye is equal to the negative value of the left boundary for the right eye. Suppose we use the subscript l to denote the left eye and the subscript r to denote the right eye. Then we have

$$\begin{aligned} T_r &= T_l = N\tan\frac{\theta}{2} \\ B_r &= -T_r \\ L_r &= -R_l = -c \times \frac{N}{f} = -\rho \times \frac{H}{2} - \frac{e}{2} \times \frac{N}{f} \\ R_r &= L_l = b \times \frac{N}{f} = \rho \times \frac{H}{2} - \frac{e}{2} \times \frac{N}{f} \end{aligned} \tag{10.11}$$

Listing 10-2 below lists the code segment that creates a stereo pair using this two-center method. The setup of the camera and the functions **init**() and **drawScene**() are very similar to those of the toe-in method of Listing 10-1 that we have explained above. In this example, we use the function **glColorMask**() to filter the colors for the left and right eyes. The output of this program is shown in Figure 10-4 below.

Program Listing 10-2: Two-center Projection Implementation (twocenter.cpp)

```
void init(void)
{
   glEnable(GL_DEPTH_TEST);
   glClearColor(0.0,0.0,0.0,0.0);   //black background
   glClearAccum(0.0,0.0,0.0,0.0);   // The default
}

void display(void)
{
   /*
     L=left,R=right,T=top,B=bottom,N = near,F = far,f=focal length
     e = eye separation, a = half-width of projection plane,
     ratio = aspect ratio, theta2 = (field of view)/2
   */
   double theta2, near,far;
   double L, R, T, B, N, F, f, e, a, b, c, ratio;

   // Clip to avoid extreme stereo
   near = camera.f / 5;
   far = 1000.0;
   f = camera.f;

   // Set the buffer for writing and reading
```

```
glDrawBuffer(GL_BACK);
glReadBuffer(GL_BACK);

// Clear things
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glClear(GL_ACCUM_BUFFER_BIT);

// Left eye filter
glColorMask(GL_TRUE,GL_FALSE,GL_FALSE,GL_TRUE);

// Create the projection
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
theta2 = ( 3.1415926 / 180 ) * camera.fov / 2;  //theta / 2 in radians
ratio  = camera.w / (double)camera.h;
a = f * ratio * tan ( theta2 );
b = a - camera.es / 2.0;
c = a + camera.es / 2.0;
N = near;
F = far;
T = N * tan ( theta2 );
B = -T;
L = -b * N / f;
R =  c * N / f;
glFrustum( L, R, B, T, N, F );

// Create the model for left eye
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
camera.p = Point3( -camera.es/2, 0, f );   //change camera viewpoint
camera.focus = Point3 ( -camera.es/2, 0, 0 );
camera.lookAt();
drawScene();
glFlush();
glColorMask(GL_TRUE,GL_TRUE,GL_TRUE,GL_TRUE);

// Write over the accumulation buffer
glAccum(GL_LOAD,1.0);

glDrawBuffer(GL_BACK);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

//now handle the right eye
glMatrixMode(GL_PROJECTION);
glLoadIdentity();

// Obtain right-eye parameters from left-eye, no change in T and B
double temp;
temp = R;
R = -L;
L = -temp;
glFrustum( L, R, B, T, N, F );

// Right eye filter
glColorMask(GL_FALSE,GL_FALSE,GL_TRUE,GL_TRUE);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
camera.p = Point3( camera.es/2, 0, f );   //change camera viewpoint
camera.focus = Point3 ( camera.es/2, 0, 0 );
```

```
    camera.lookAt();
    drawScene();
    glFlush();
    glColorMask(GL_TRUE,GL_TRUE,GL_TRUE,GL_TRUE);

    // Add  the new image and copy the result back
    glAccum(GL_ACCUM,1.0);
    glAccum(GL_RETURN,1.0);

    glutSwapBuffers();
}
```

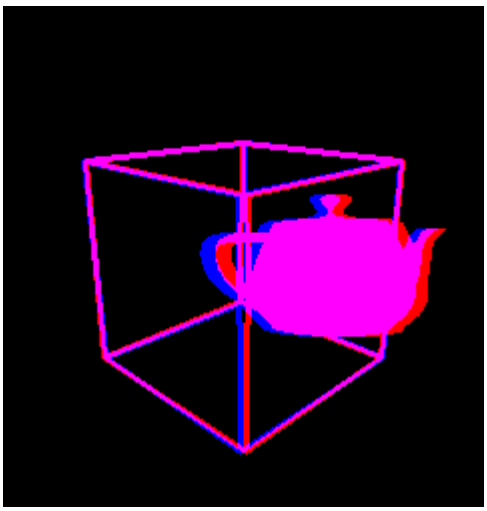

Figure 10-4 Anaglyph Created Using Two-center Method

In the implementation, we have again used a black background. It is not appropriate to use a white background to create a stereo pair. This is because when we use the OpenGL commands to accumulate the colors, all values in the color buffer, including the background values will be added and this could lead to undesired values. More importantly, stereo pairs do not work well with a white background. With a black background, when a region with blue color passes through the red filter of our left eye, it becomes black and 'disappears', and our left eye will not see the region. On the other hand, if a white background is used, the blue region simply becomes a black region and our left eye still can see a black region on a white background. This means that the left eye will see something that is in not supposed to be seen in the real 3D world. As a consequence, this will largely compromise the stereoscopic effect.

In some special cases that we do need a non-black background, we can first start with a black background and use the accumulation buffer to accumulate the objects we need to render. In the process, we use the stencil buffer to keep track of the pixels that the objects have mapped to. The stencil buffer creates a mask for the objects. At the end, we put in the special background we want but will mask off the pixels the objects have occupied. We will present the implementation of this technique in Chapter 14, where we discuss the creation of stereo pairs using extrusion and surface of revolution.

Other books by the same author

Windows Fan, Linux Fan

by *Fore June*

Windws Fan, Linux Fan describes a true story about a spiritual battle between a Linux fan and a Windows fan. You can learn from the successful fan to become a successful Internet Service Provider (ISP) and create your own wealth. See *http://www.forejune.com/*

Second Edition, 2002.
ISBN: 0-595-26355-0 Price: $6.86