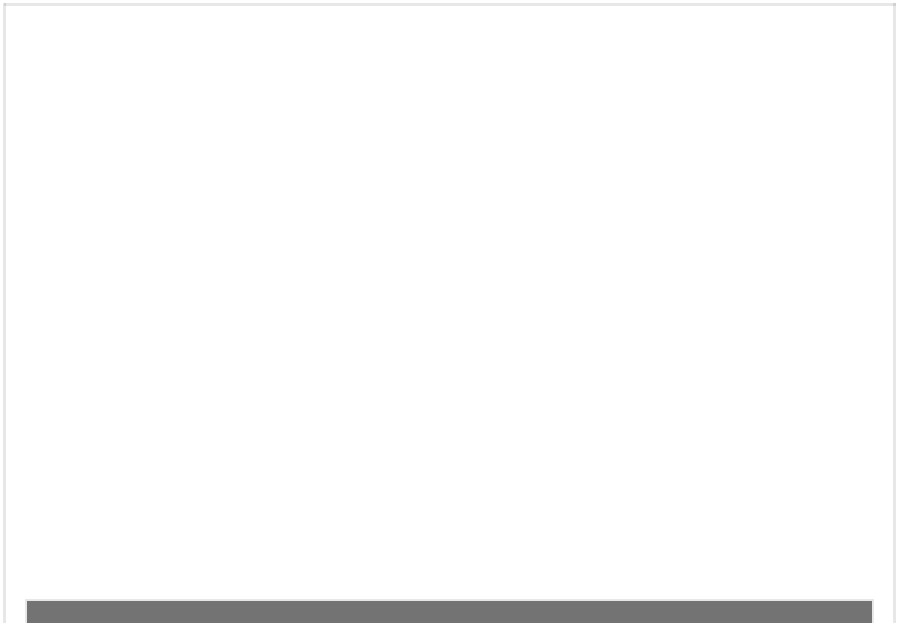
Rendering 3D Anaglyph in OpenGL

It's quite easy (and fun!) to render 3D anaglyphs with OpenGL. I expect you to know what anaglyphs are and how to view them. If not, read this earlier article. The focus of this tutorial is to provide you enough background and code snippets for the task, so that you may have fun rendering anaglyphs with your own programs.

We will focus on producing a red-cyan anaglyph from a given 3D scene. The scene will be rendered twice: Once by setting up the camera for the left eye which will be subsequently filtered to let only red color pass, and other time for the right eye, which will then be filtered so that 'green plus blue' (cyan) components pass. To implement this idea, you'll need to understand the role of *parallax* in stereoscopic vision and the concept of projection and viewing as they apply to OpenGL. This tutorial assumes familiarity with OpenGL projection and modelview transforms. If you know all that stuff, go on ahead, else just skim through this chapter from the redbook and you'll be prepared.

What is parallax? When you look at a 3D anaglyph without the glasses, you will find that the edges of the objects appear displaced in the red and cyan components of the picture. Observe it in the cylinder below:



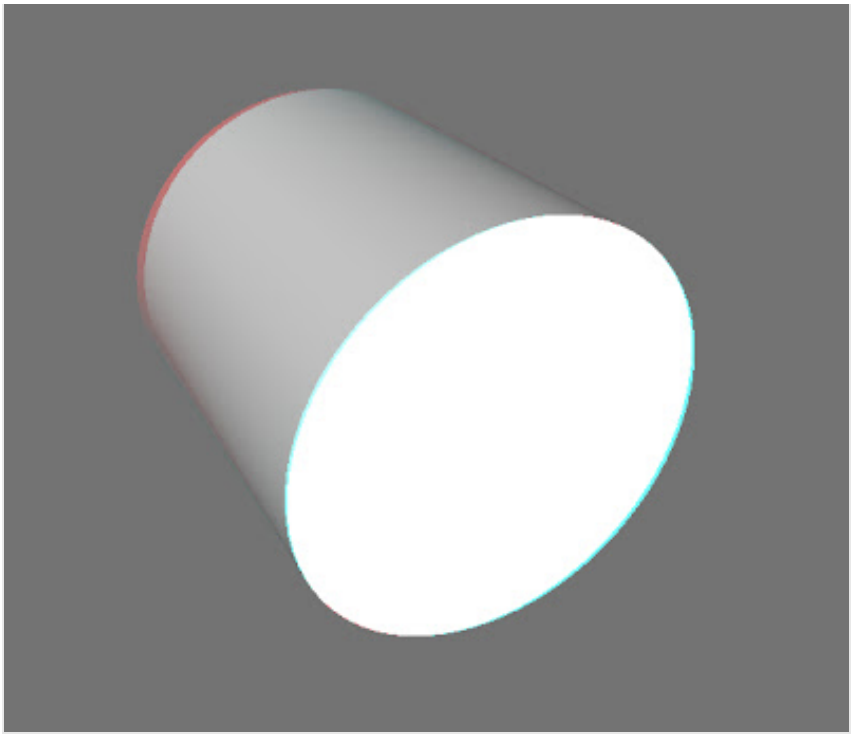


Fig. 1: Sample anaglyph - a cylinder

The human visual system needs depth cues from a flat image (photograph or display-screen) in terms of how much an object shifts laterally between the left eye and the right eye. When we say parallax, we mean exactly this kind of displacement in the image. To see the effect of parallax in the above image, use red/cyan glasses (red on left eye) and try hovering mouse pointer over the left end of the cylinder. It will look sunken into the screen whereas the front of the cylinder will appear more or less at the same depth as the screen. In rendering an anaglyph, all that we're trying to achieve is to get the right kind of parallax for the objects in the scene and the rest is automatically done in the brain, for free!

Parallax is not just qualitative, it has a numeric value and can be positive, negative or zero. In the application, parallax is created by defining two cameras corresponding to the left and right eyes separated by some distance (called *interocular distance* or simply eye-separation) and having a plane at a certain depth along the viewing direction (called *convergence distance*) at which the parallax is zero. Objects at the convergence depth will appear to be

at the same depth as the screen. Objects closer to the camera than the convergence distance will seem to be out-of-screen and objects further in depth than the convergence distance will appear inside the screen. Following figure illustrates this situation:

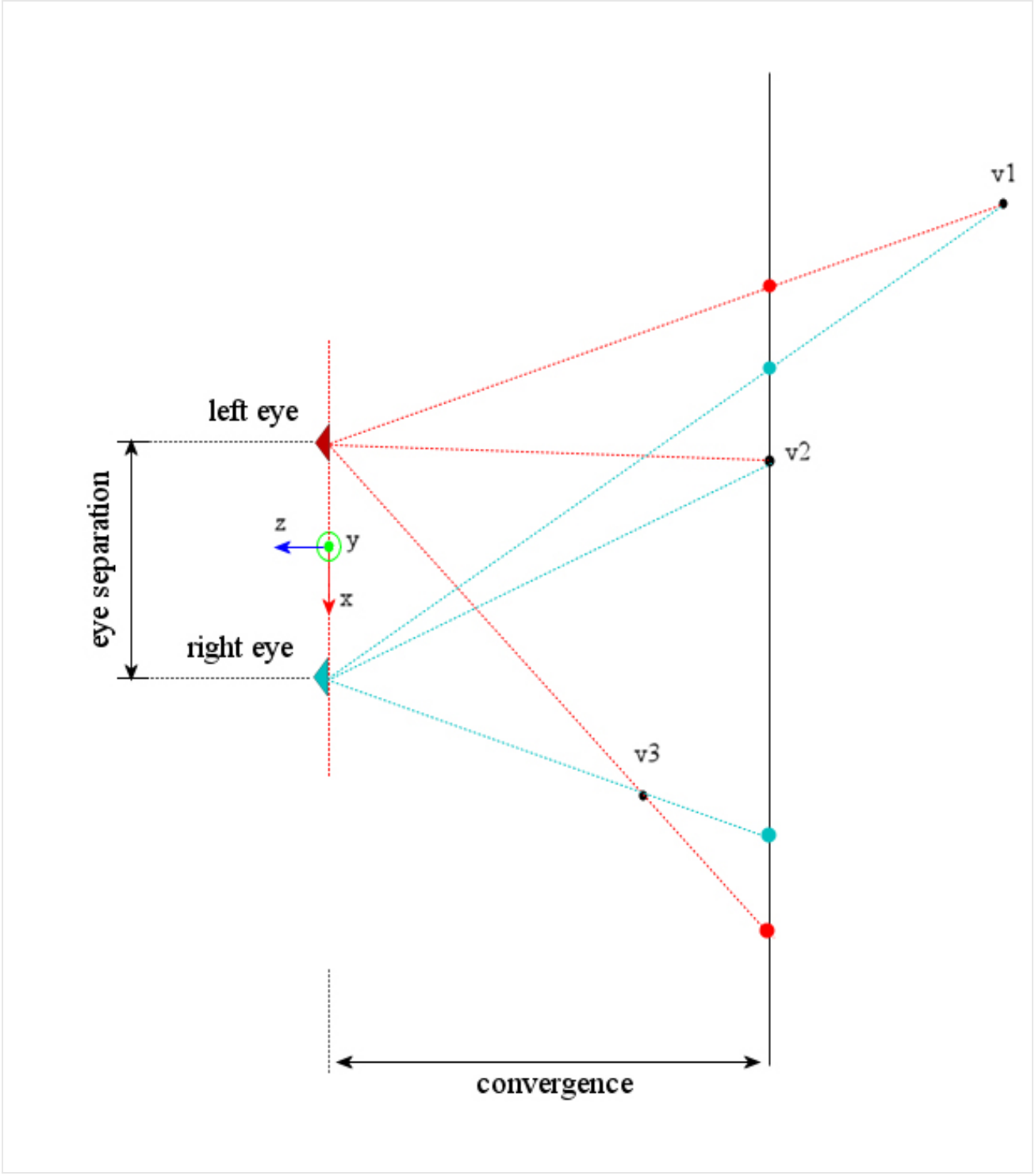


Fig. 2: Parallax resulting from vertices at different depths

In this figure, there are two virtual cameras, one for the left eye (with some negative offset from the origin on the X-axis) and the other for the right eye (with some positive offset from the origin on the X-axis). They are separated by the same amount as the offset between human eyes, which averages around 65 mm. Then there is the depth of zero-parallax called convergence depth. You can imagine it as a plane along x-y axes and located at depth of convergence along the negative Z-axis. For illustration of different kinds of parallax three vertices - v1, v2 and v3 are shown.

For each vertex, consider a line from the vertex to each camera and observe where they intersect with the convergence plane. The gap between the points on the convergence plane for left and the right cameras is the measure of parallax generated by the vertex:

- The vertex v1 which is at a greater depth compared to the convergence creates a parallax. Observe that the red and the cyan dots on the convergence plane are oriented the same way as the two cameras: Red dot is towards the left camera and cyan dot is towards the right camera. This is called positive parallax, and the vertex v1 will appear inside the screen upon being rendered.
- The vertex v2 which is at the same depth as the convergence plane creates zero parallax. There are no separate red and cyan points on the convergence plane here, as there were in the previous case. The vertex v2 will appear at the same depth as the screen when rendered.
- The vertex v3, which is located at a distance less than the convergence distance also causes parallax. But the red and cyan points projected on the convergence plane are oriented opposite to the orientation of the cameras. The point corresponding to the left camera is on the right and the point corresponding to the right camera is on the left. This is called negative parallax and the vertex v3 will seem to appear out of the screen when rendered.

Let's look at another anaglyph which illustrates above characteristics of parallax generated by vertices at different depths:

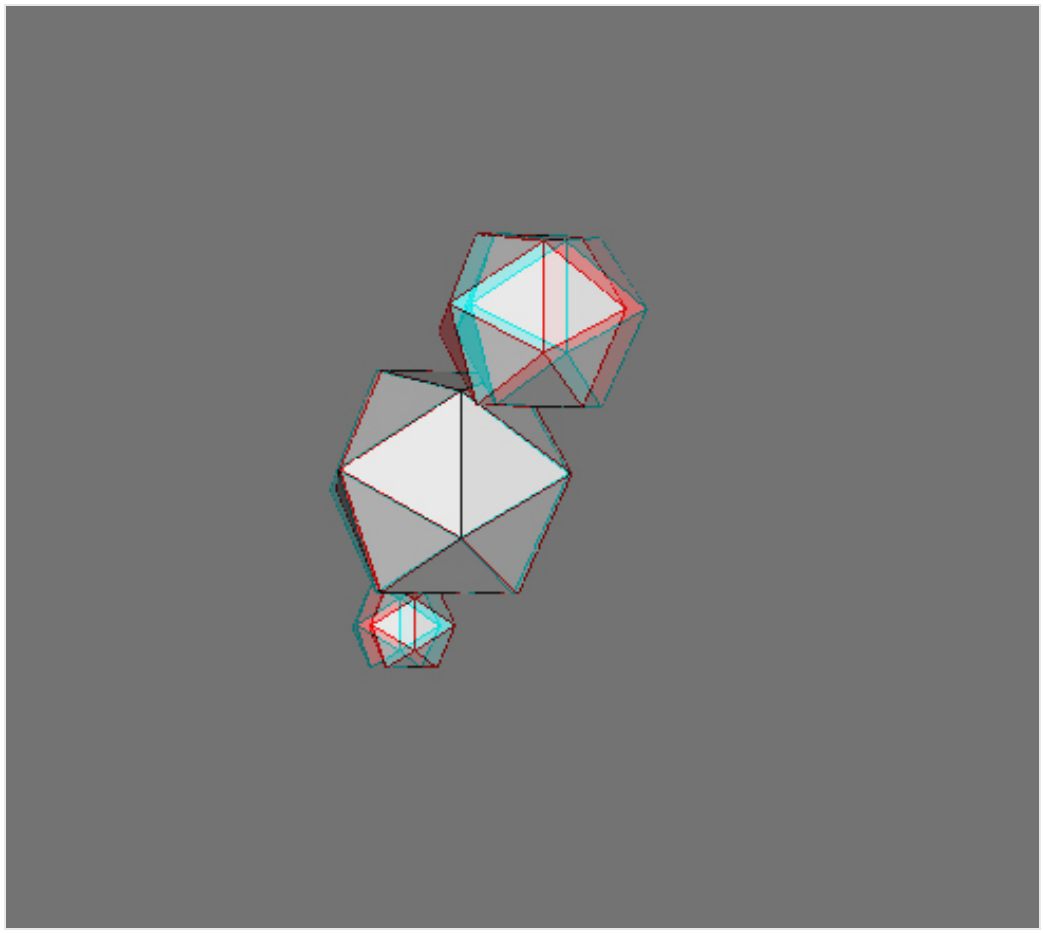


Fig. 3: Icosahedrons showing negative, zero and positive parallax

In the figure, there are three icosahedrons. The one the bottom is furthest in the scene and the one at the top is closest to the camera. The icosahedron in the centre is approximately at the same depth as the screen. Notice how the parallax generated for the near and far icosahedrons is of opposite alignment. The closest icosahedron has red edges to the right and cyan edges to the left. This illustrates negative parallax (for glasses with red filter on left eye) and the icosahedron appears slightly out of the screen when viewed from the red-cyan colored glasses. The small icosahedron at the bottom of the figure has red edges to the left and cyan edges to the right. This alignment is same as the colored glasses used to view them. The parallax created is positive and the icosahedron appears inside the screen behind the icosahedron in the centre

which is rendered with (almost) no parallax and is at the same depth as the screen.

As I mentioned before, parallax can be measured qualitatively. I will now proceed to obtain an equation for parallax introduced in a vertex at a certain depth. While this is not crucially important in setting up the OpenGL for rendering anaglyphs, it is important when you plan the scene and overall range of usable parallax in your interactive application. Consider the following diagram, in which the vertex V is located at depth w and lies beyond the convergence distance:

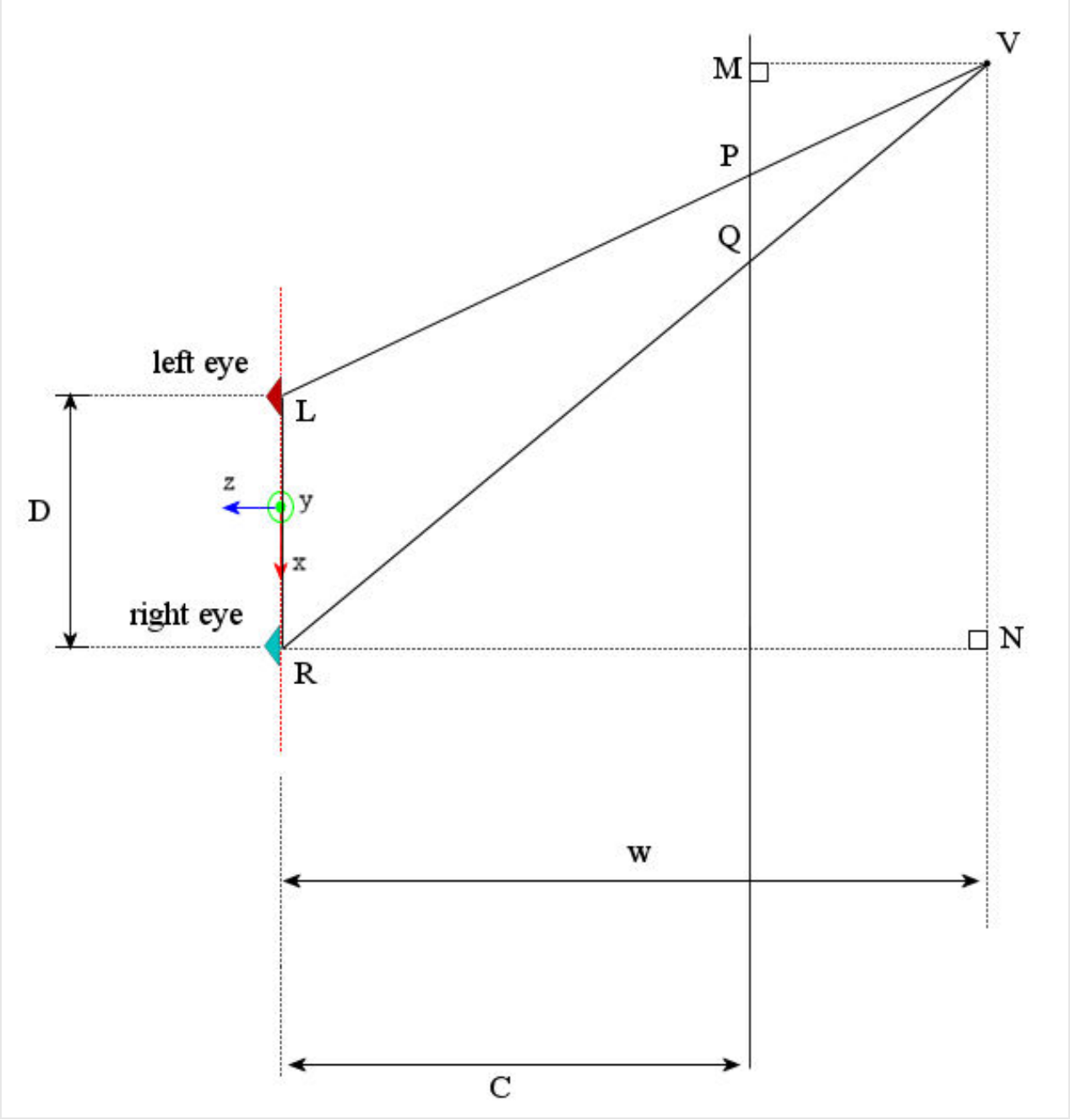


Fig. 4: Measuring the amount of parallax for a vertex beyond convergence distance

The eye separation is D and the convergence distance is C. The line joining the left camera L and vertex V meets the convergence plane at P. Similarly the line joining the right camera R and the vertex V meets the convergence plane at Q. The parallax p associated with the vertex V is the distance PQ. Now consider ΔLVR, wherein by use of the intercept theorem we have:

$$PQLR=VQVR$$

And by the similarity $\Delta QVM \sim \Delta VRN$, we have

$$VQVR = VMRN = w - Cw = 1 - C/w$$

Thus,

$$PQLR = pD = 1 - C/w$$

Or

$$p = D(1 - C/w)$$

Similarly for a vertex that is closer than the convergence distance as shown in the figure:

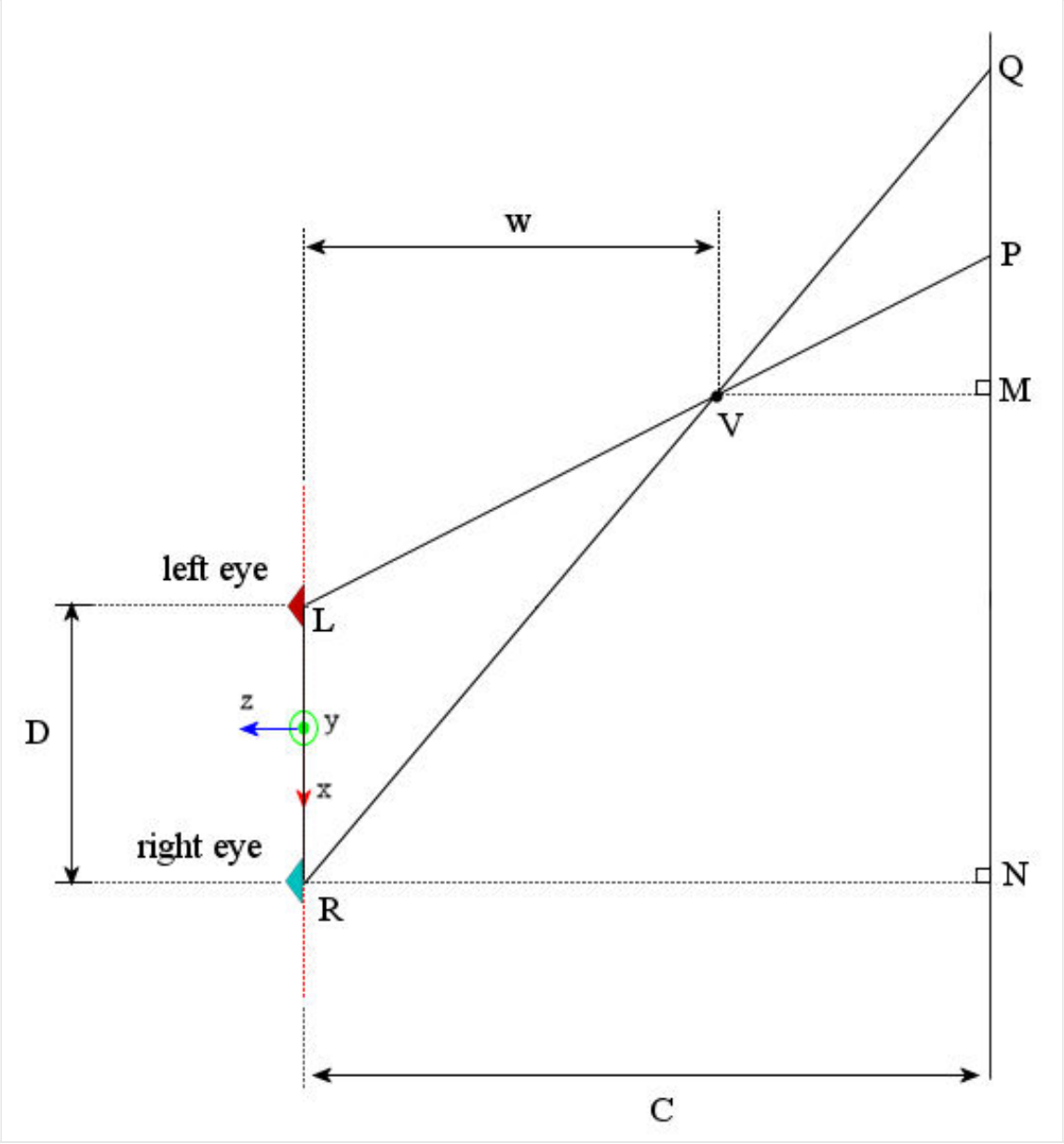


Fig. 5: Measuring the amount of parallax for a vertex closer than the convergence distance

The parallax p can be evaluated again by applying the intercept theorem:

$$PQLR = QVVR = QVQR - QV = QVQR1 - QVQR$$

Now in ΔQNR since $VM \parallel RN$, we have

$$QVQR = VMRN = C - wC = 1 - w/C$$

Thus,

$$PQLR = pD = 1 - w/Cw/C = C/w - 1$$

Or,

$$p = -D(1 - C/w)$$

This equation is the same as that for a vertex further than the convergence distance, with a negative sign. The negative sign implies that the projections of the vertex are on the convergence plane are on opposite side as the corresponding camera. If we disregard the sign in the equation, a negative parallax can be understood as the vertex being closer than the convergence distance. A plot of $p = D(1 - C/w)$ is shown below:

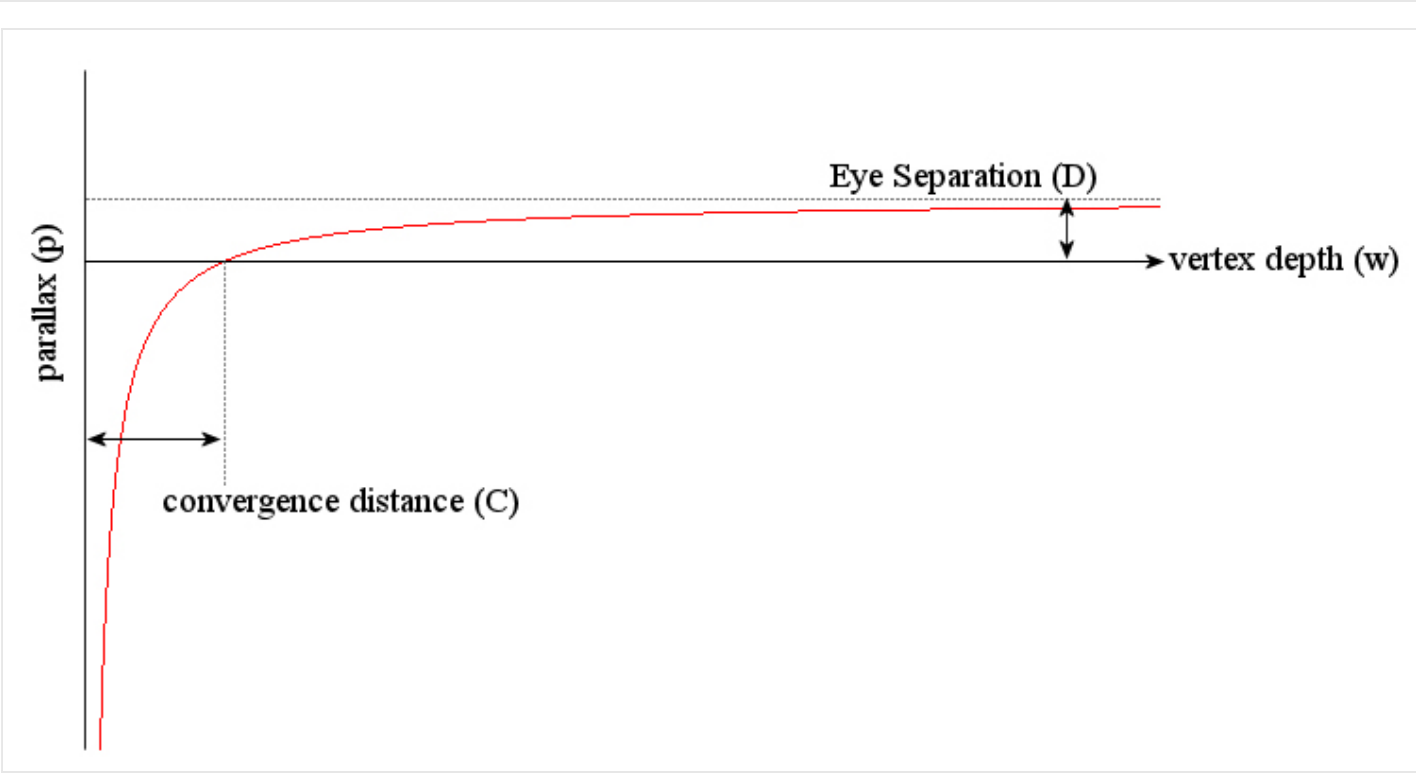


Fig. 6: Variation of parallax with vertex depth for a given convergence distance and eye-separation

The graph shows that as the vertex moves further and further into the scene, the parallax generated asymptotically approaches the value of eye separation D. The parallax remains positive at all vertex depths greater than the convergence distance C, at which the parallax is zero. For vertices that are closer in the scene that the distance C, the parallax is negative and quickly

approaches $-\infty$. Note that for a vertex at $w=C/2$, the parallax obtained is the same as eye separation. Such large vales of negative parallax can make the viewer's eyes diverge causing strain and should be avoided. The practical value of convergence depth is chosen on the basis of the shot being prepared and the type of effect (out of the screen or inside screen) used. Eye separation is typically kept at 1/30*th* of the convergence distance and objects closer than half the convergence distance are avoided in the scene.

The only remaining task is to discuss how we set up a twin camera in OpenGL. In non-stereo mode, you require only one camera, whose viewing parameters are defined by calling *glFrustum()* or *gluPerspective()*. The frustum obtained looks like the following:

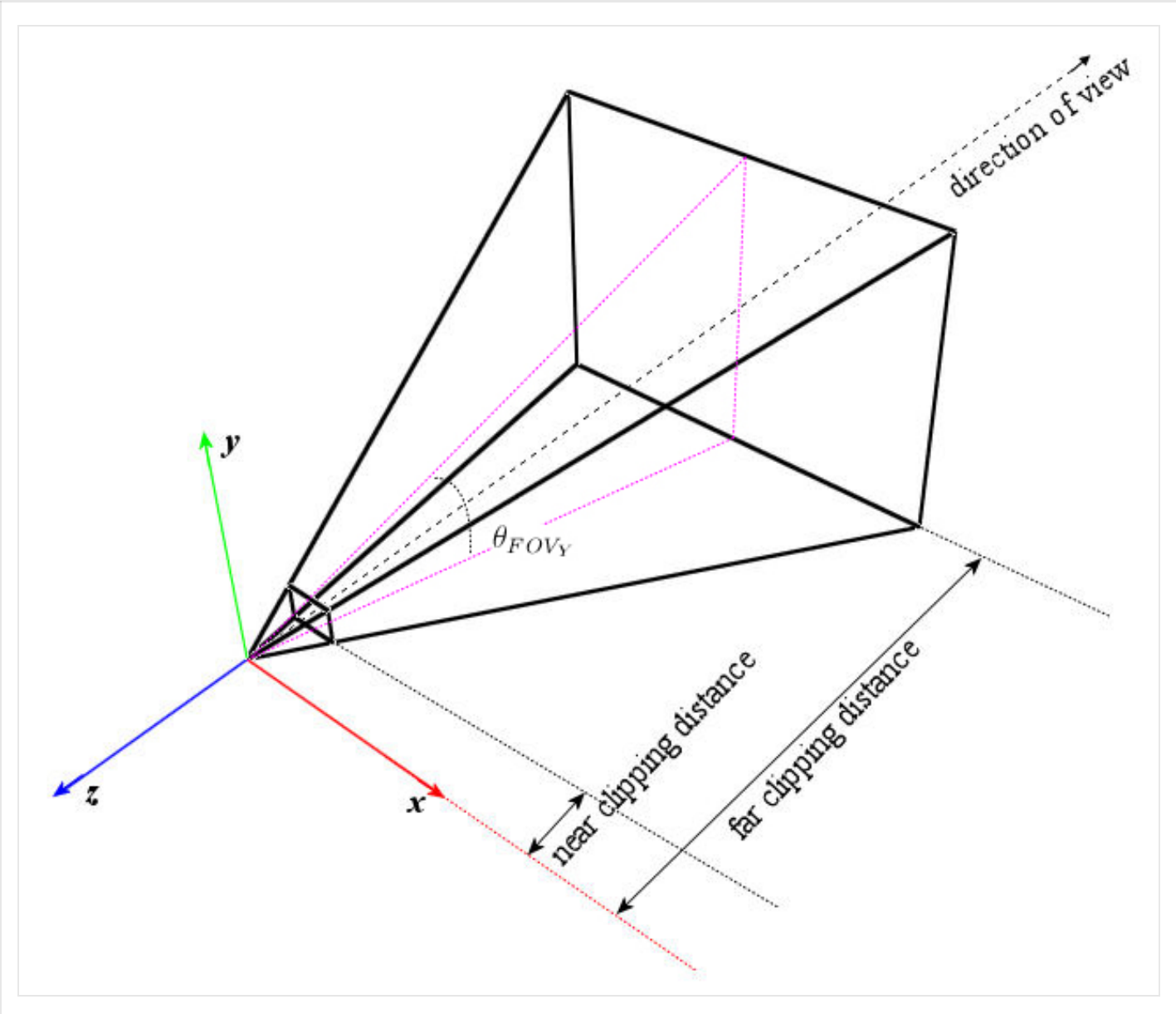


Fig. 7(a): A mono frustum

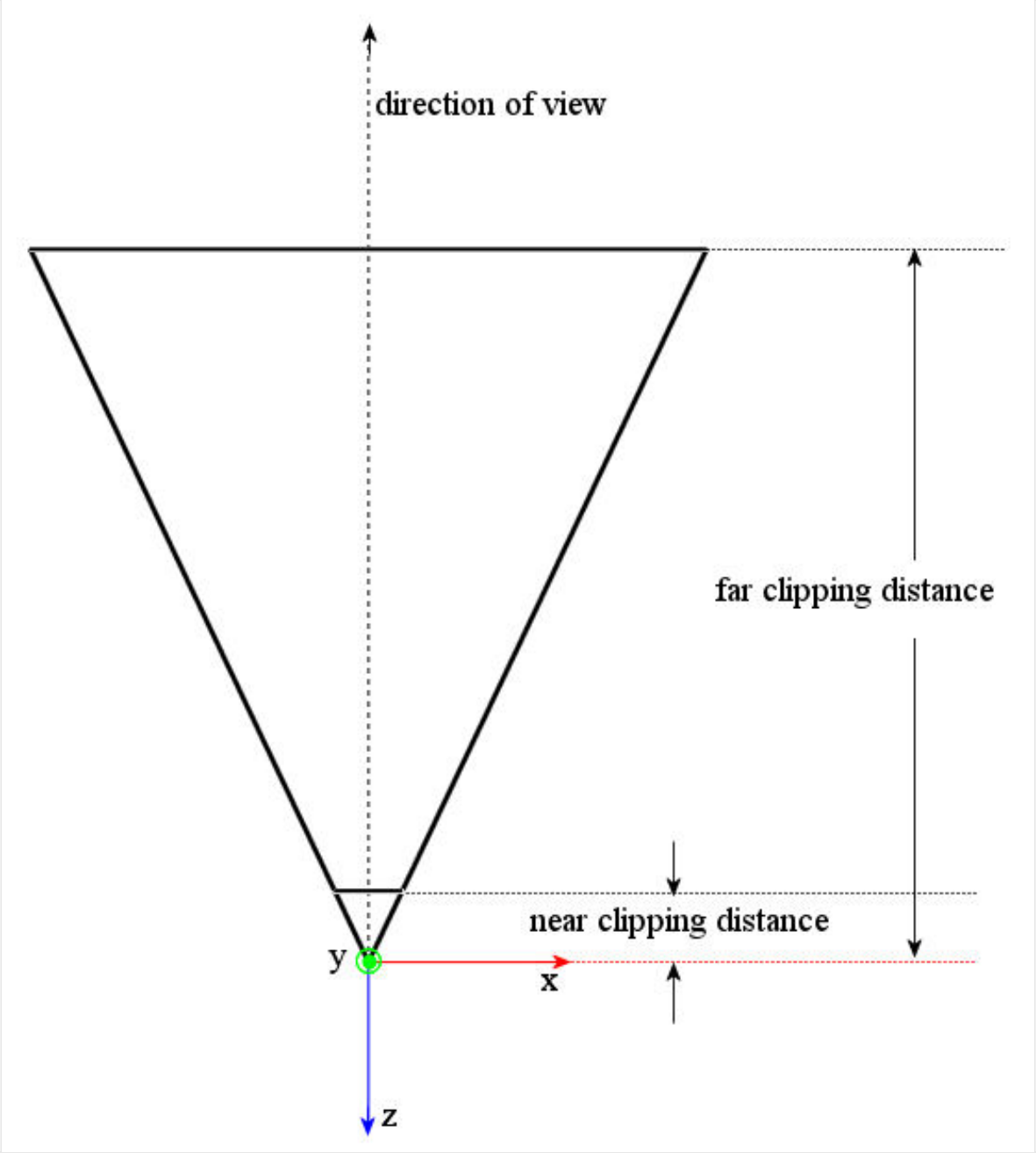


Fig. 7(b): Mono frustum (orthographic view from top)

The twin-camera setup needed for stereoscopic rendering, however, resembles the following:

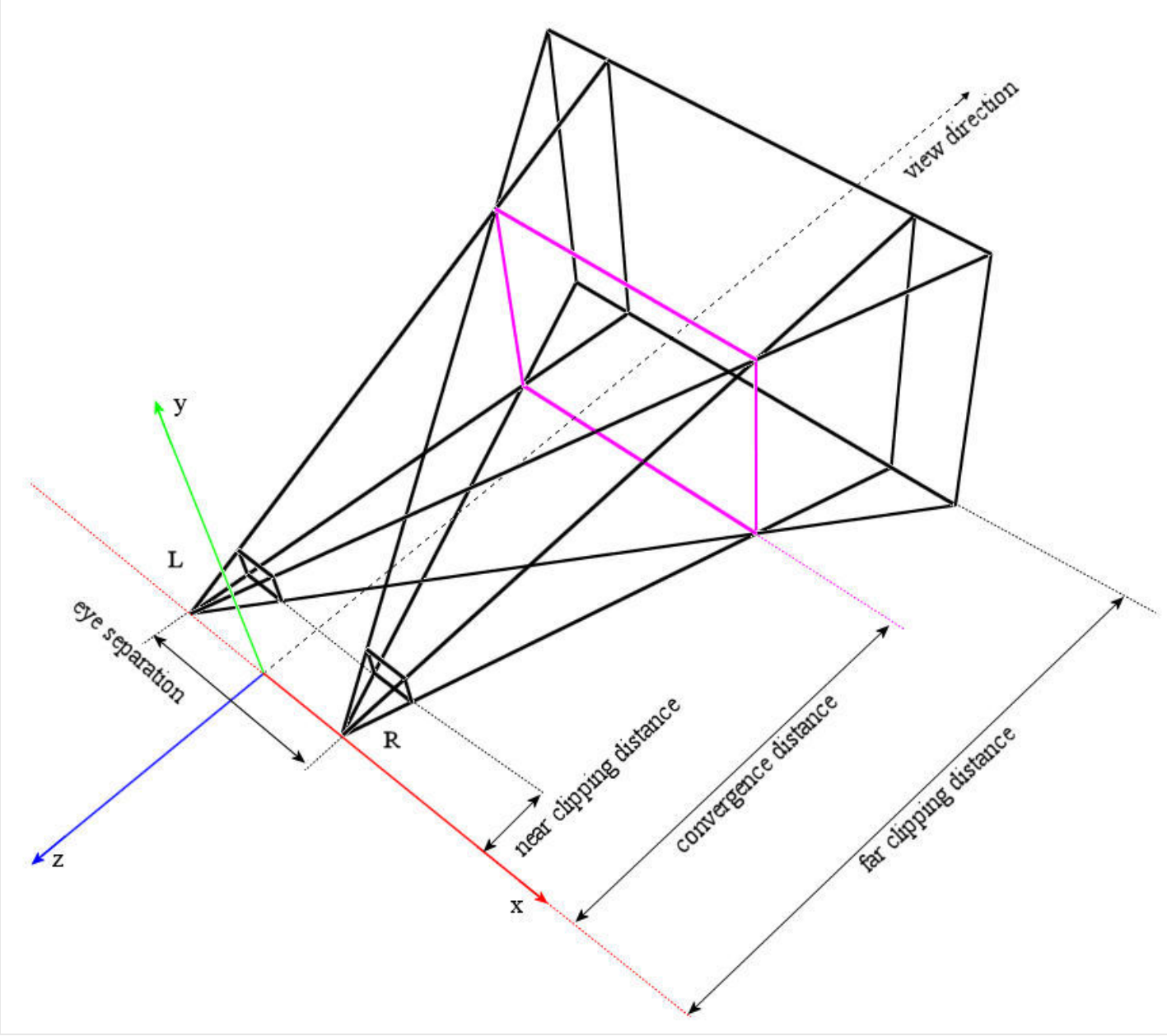


Fig. 8(a): Twin-camera system for stereoscopic rendering

In this setup, we have two frustums: One originating at point *L* (for the left eye) and the other originating at point *R* (for the right eye). The distance *LR* is the eye-separation, so that the points *L* and *R* are offset from the origin along negative and the positive X-axes respectively by an amount *LR*2 each. Observant readers might have already noticed that the the two frustums in the figure above are not the same as the mono-frustum that was shown before and that we did not offset a mono-frustum along the X-axis to obtain the twin-camera system. In fact, the two frustums shown above are asymmetric,

whereas the mono-frustum was symmetric (see the next two figures for a better idea). If the two frustums were symmetric and displaced laterally, they wouldn't converge at all. Asymmetry causes the two frustums to converge at the *convergence distance*. The magenta colored rectangle at the convergence distance represents the *virtual screen* for the stereoscopic rendering. Any vertex at on the virtual screen will be appear with zero-parallax. Vertices closer or further than this distance will cause appropriate amounts of negative or positive parallax. Following figure shows the same system with an orthographic view from top:

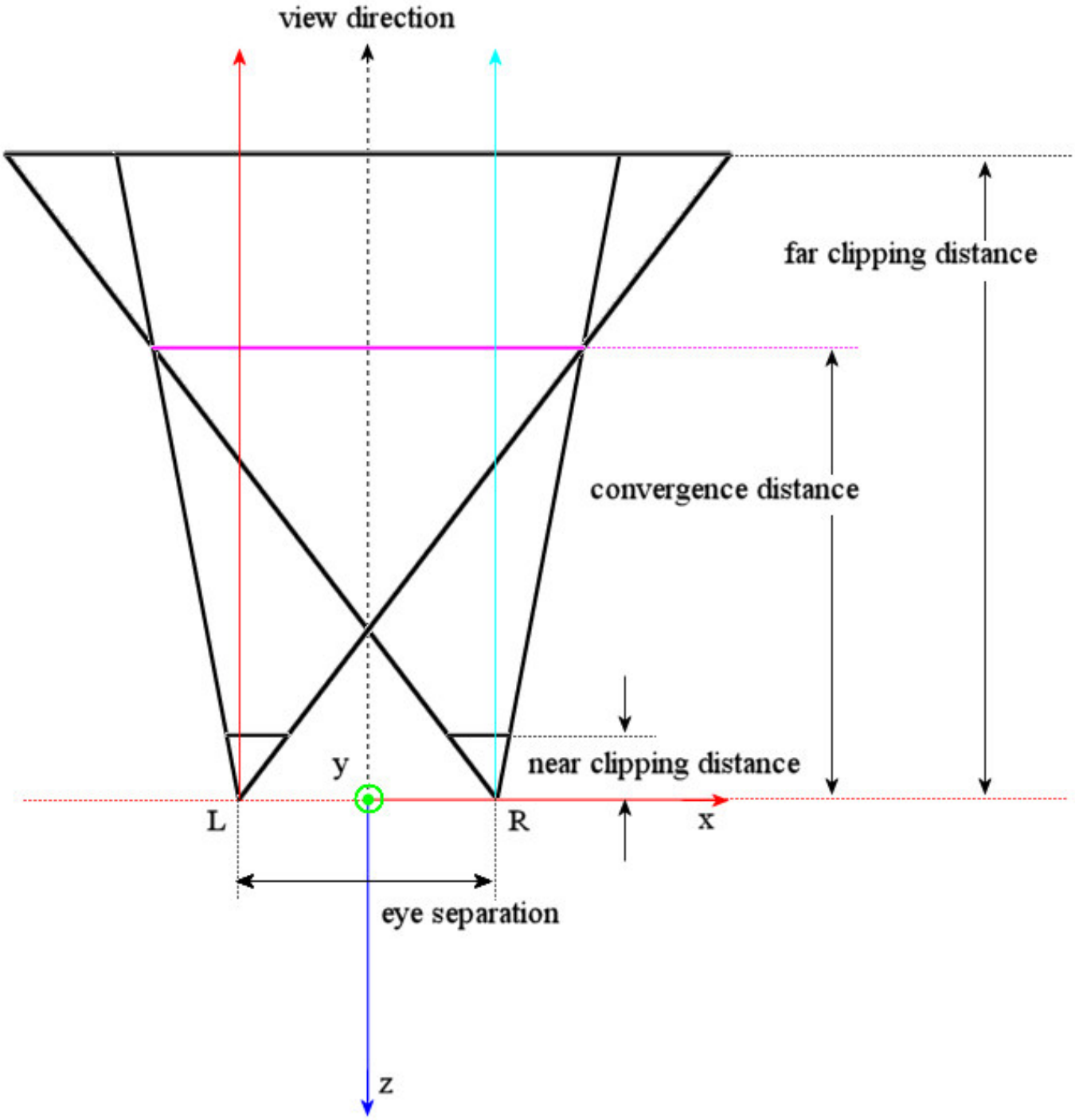


Fig. 8(b): Stereo-frustum (orthographic view from top)

The asymmetry of the frustums is clearly evident above. Also notice that the view direction for each frustum is parallel to the other and also to the the Z-axis, same as would be for a mono-frustum. This is the correct way to set-up the stereo pair. There is another twin-camera setup method called *toed-in* cameras that involves symmetric frustums but the left and right view directions are at an angle to each other. It is sufficient to say that toed-in

camera setup is incorrect. Here's another figure showing our stereo-camera system along the Z-axis:

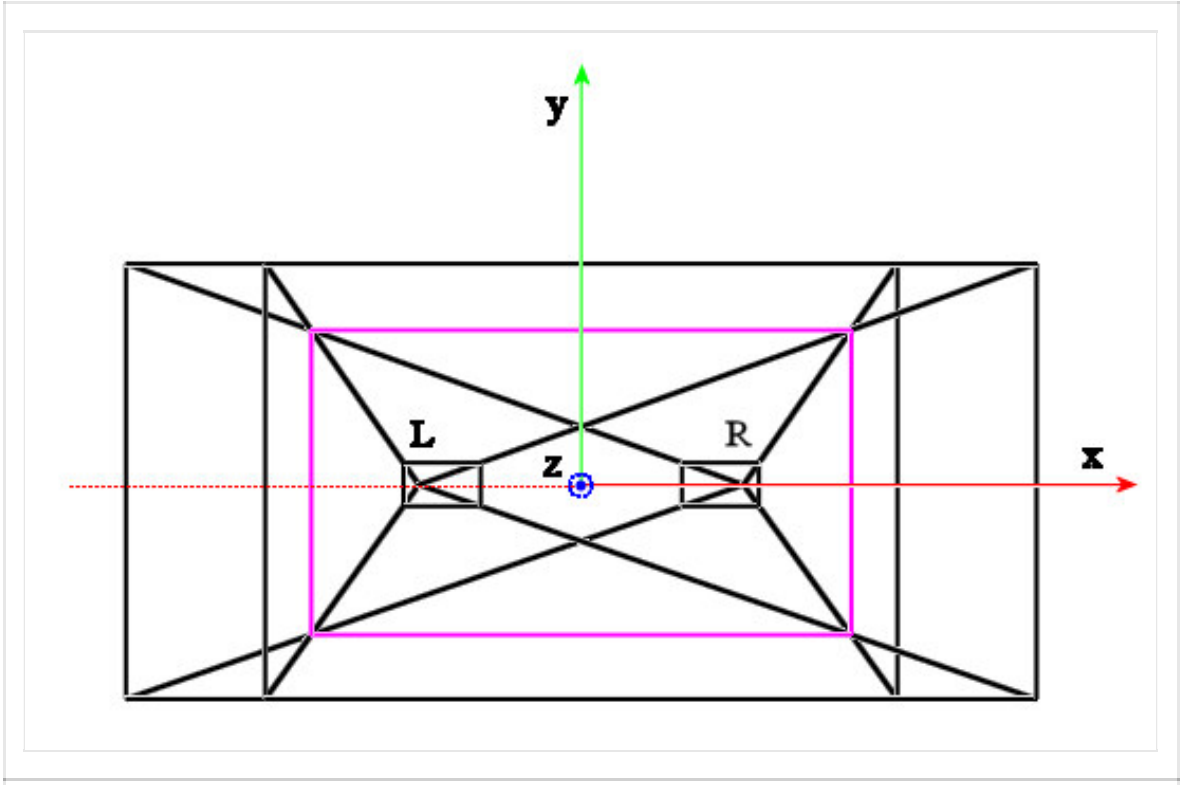


Fig. 8(c): Stereo-frustum (orthographic view from back)

At this point we know sufficiently to calculate the stereoscopic frustum parameters which we can use in an OpenGL program. Observe the following figure:

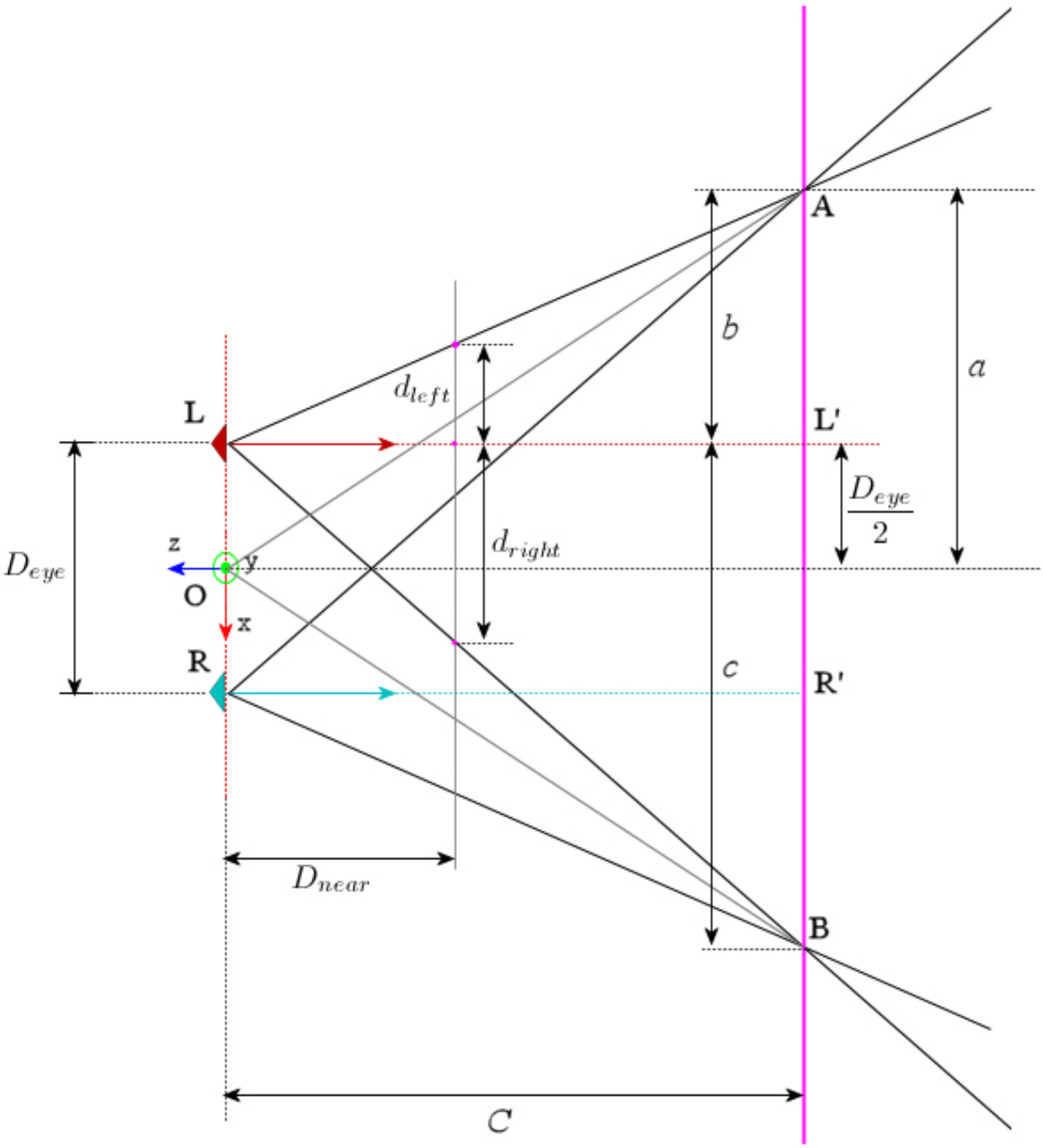


Fig.9: Calculation of frustum parameters

While it appears formidable, no new information has been added to the figure above. If you have understood everything so far, you will breeze through the simple calculations that follow. We have, as before, two cameras located at points *L* and *R* on the X-axis. The separation *LR* between them is *Deye* and their offsets are symmetric about the origin. The camera directions are parallel, both looking down $-Z$ axis. The near clipping distance of the

frustums is *Dnear* and the convergence distance is *C*. The extremities of the virtual screen are at points *A* and *B* as seen in the top-view. Point *A* is where the left side of both frustums meet and point *B* is where the right side of both frustums meet.

In OpenGL, the only way to create an asymmetric frustum is through the *glFrustum()*. The function *gluPerspective()* creates only symmetric frustums and hence cannot be used in this case. *gluPerspective()* takes natural looking parameters such as the field of view angle along Y-direction $\theta FOVY$ (see fig. 7(a)), the aspect ratio and the distance of the near and far clipping planes. For *glFrustum()*, you need to provide near clipping plane's top, bottom, left and right coordinates, as well as the distance of the near and far clipping planes. We will compute these parameters from the geometry of the dual frustum shown above.

In the figure, the equivalent of a mono-frustum corresponding to the virtual screen would be *AOB*. Let its field of view along *Y* direction be $\theta FOVY$ and the aspect ratio be *raspect* (same way these are in *gluPerspective()*). Then the *top* and *bottom* parameters for the *glFrustum()* will evaluate as:

$$top = Dnear \tan\frac{\theta FOVY}{2}$$

$$bottom = -top$$

These values apply to both left and right frustums. The half-width *a* of the virtual screen is

$$a = raspect C \tan\frac{\theta FOVY}{2}$$

Now look at the left frustum *ALB*. The near clipping plane intersects it at *dleft* distance left of *LL'* and *dright* distance right of *LL'*. In $\Delta ALL'$ and $\Delta BLL'$,

$dleft b = dright c = Dnear C$

Also, we have

$b = a - \frac{Deye}{2}$

$c = a + \frac{Deye}{2}$

So that we can readily calculate *dleft* and *dright*. Similarly for the right frustum *ARB*, we could obtain *dleft* and *dright* by interchanging *b* and *c*. Here is a code snippet showing how you could wrap the above equations in a small class called *StereoCamera*:

```
05          float Convergence,
06          float EyeSeparation,
07          float AspectRatio,
09          float NearClippingDistance,
10          float FarClippingDistance
13          mConvergence             = Convergence;
14          mEyeSeparation           = EyeSeparation;
15          mAspectRatio             = AspectRatio;
16          mFOV                     = FOV * PI / 180.0f;
17          mNearClippingDistance    = NearClippingDistance;
18          mFarClippingDistance     = FarClippingDistance;
21      void ApplyLeftFrustum()
23          float top, bottom, left, right;
25          top     = mNearClippingDistance * tan(mFOV/2);
26          bottom  = -top;
28          float a = mAspectRatio * tan(mFOV/2) *
mConvergence;
30          float b = a - mEyeSeparation/2;
31          float c = a + mEyeSeparation/2;
```

```
33          left    = -b * mNearClippingDistance/mConvergence;
34          right   =  c * mNearClippingDistance/mConvergence;
37          glMatrixMode(GL_PROJECTION);
38          glLoadIdentity();
39          glFrustum(left, right, bottom, top,
40                    mNearClippingDistance,
mFarClippingDistance);
43          glMatrixMode(GL_MODELVIEW);
44          glLoadIdentity();
45          glTranslatef(mEyeSeparation/2, 0.0f, 0.0f);
48      void ApplyRightFrustum()
50          float top, bottom, left, right;
52          top     = mNearClippingDistance * tan(mFOV/2);
53          bottom  = -top;
55          float a = mAspectRatio * tan(mFOV/2) *
mConvergence;
57          float b = a - mEyeSeparation/2;
58          float c = a + mEyeSeparation/2;
60          left    =  -c *
mNearClippingDistance/mConvergence;
61          right   =   b *
mNearClippingDistance/mConvergence;
64          glMatrixMode(GL_PROJECTION);
65          glLoadIdentity();
66          glFrustum(left, right, bottom, top,
67                    mNearClippingDistance,
mFarClippingDistance);
70          glMatrixMode(GL_MODELVIEW);
71          glLoadIdentity();
72          glTranslatef(-mEyeSeparation/2, 0.0f, 0.0f);
76      float mConvergence;
```

```
77      float mEyeSeparation;
78      float mAspectRatio;
80      float mNearClippingDistance;
81      float mFarClippingDistance;
```

The code does exactly what we described with the equations and diagrams earlier. Once you have created a *StereoCamera* object, you can call the methods *ApplyLeftFrustum()* and *ApplyRightFrustum()* to set up the respective asymmetric frustums. Note that in these methods, the projection transform is followed by a modelview transform in which we translate along the X-axis. This has the effect of moving the camera to a position offset from the origin. As such there is no camera transform in OpenGL. What we do is move the world in a direction opposite to the conceptual camera using the modelview transform. In order to use the above class, you could write your OpenGL rendering function as the following:

```
02  void
    DrawGLScene(GLvoid)
04      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
15      cam.ApplyLeftFrustum();
16      glColorMask(true, false, false, false);
18      PlaceSceneElements();
20      glClear(GL_DEPTH_BUFFER_BIT) ;
22      cam.ApplyRightFrustum();
23      glColorMask(false, true, true, false);
25      PlaceSceneElements();
27      glColorMask(true, true, true, true);
31  void PlaceSceneElements()
34      glTranslatef(0.0f, 0.0f, -1800.0f);
37      glRotatef(-60.0f, 1.0f, 0.0f, 0.0f);
```

```
38      glRotatef(-45.0f, 0.0f, 0.0f, 1.0f);
42          glTranslatef(0.0f, 0.0f, 240.0f);
43          glRotatef(90.0f, 1.0f, 0.0f, 0.0f);
44          glColor3f(0.2, 0.2, 0.6);
45          glutSolidTorus(40, 200, 20, 30);
46          glColor3f(0.7f, 0.7f, 0.7f);
47          glutWireTorus(40, 200, 20, 30);
51          glTranslatef(240.0f, 0.0f, 240.0f);
52          glColor3f(0.2, 0.2, 0.6);
53          glutSolidTorus(40, 200, 20, 30);
54          glColor3f(0.7f, 0.7f, 0.7f);
55          glutWireTorus(40, 200, 20, 30);
```

We begin the rendering function by clearing the color and depth buffers. Then we set up the stereo camera system. We apply the left frustum and instruct OpenGL to allow only red components in the color buffer. Then we call the routine to draw the scene. After this we clear the depth buffer, but retain the color buffer (which has only red-channel values). With depth buffer cleared we activate the right frustum and instruct OpenGL to allow only green and blue color components in the color buffer. We call our drawing routine one more time. Note that the colors scene for the left eye and the scene for the right eye have no overlapping color spaces, so no explicit blending/accumulation is required. Finally we enable all the color channels and the scene gets rendered as anaglyph. Note that we had to render geometry twice. That means that the frame-rate gets reduced to half of what we would obtain with a mono-frustum. This is typical of stereoscopic rendering. If you're wondering what output is generated by the above snippets, here it is:

Fig. 10: Output of the drawing routine in the listing above

I will leave you with a video that I made during writing of this (somewhat lengthy) tutorial. The video uses the same theory and code that I covered above. Try to watch this one at 720p full-screen for best effect:

Further Reading:

You could find a lot of material on stereographics compiled by Paul Bourke on this page. You can also watch a video presentation by NVIDIA from GTC 2010 here and download the slides for offline viewing.

Have fun!