Chapter 7
Head-Mounted System Software Development

In designing a graphical eye tracking application, the most important requirement is mapping of eye tracker coordinates to the appropriate application program's reference frame. The eye tracker calculates the viewer's point of regard (POR) relative to the eye tracker's screen reference frame, e.g., a 512 × 512 pixel plane, perpendicular to the optical axis. That is, for each eye, the eye tracker returns a sample coordinate pair of x- and y-coordinates of the POR at each sampling cycle (e.g., once every ∼16 ms for a 60 Hz device). This coordinate pair must be mapped to the extents of the application program's viewing window.

If a binocular eye tracker is used, two coordinate sample pairs are returned during each sampling cycle, x_l, y_l for the left POR and x_r, y_r for the right eye. A virtual reality (VR) application must, in the same update cycle, also map the coordinates of the head's position and orientation tracking device (e.g., typically a 6 DOF tracker is used).

The following sections discuss the mapping of eye tracker screen coordinates to application program coordinates for both monocular and binocular applications. In the monocular case, a typical 2D image-viewing application is expected, and the coordinates are mapped to the 2D (orthogonal) viewport coordinates accordingly (the viewport coordinates are expected to match the dimensions of the image being displayed). In the binocular (VR) case, the eye tracker coordinates are mapped to the dimensions of the near viewing plane of the viewing frustum. For both calculations, a method of measuring the application's viewport dimensions in the eye tracker's reference frame is described, where the eye tracker's own (fine-resolution) cursor control is used to obtain the measurements. This information should be sufficient for most 2D image viewing applications.

For VR applications, subsequent sections describe the required mapping of the head position/orientation tracking device. Although this section should generalize to most kinds of 6 DOF tracking devices, the discussion in some cases is specific to the Ascension Flock Of Birds (FOB) d.c. electromagnetic 6 DOF tracker. This section discusses how to obtain the head-centric view and vectors from the matrix returned

A.T. Duchowski, *Eye Tracking Methodology*, DOI 10.1007/978-3-319-57883-5_7

by the tracker, and also explains the transformation of an arbitrary vector using the obtained transformation matrix.[1] This latter derivation is used to transform the gaze vector to head-centric coordinates, which is initially obtained from, and relative to, the binocular eye tracker's left and right POR measurement.

Finally, a derivation of the calculation of the gaze vector in 3D is given, and a method is suggested for calculating the three-dimensional gaze point in VR.

7.1 Mapping Eye Tracker Screen Coordinates

When working with the eye tracker, the data obtained from the tracker must be mapped to a range appropriate to the given application. If working in VR, the 2D eye tracker data, expressed in eye tracker screen coordinates, must be mapped to the 2D dimensions of the near viewing frustum. If working with images, the 2D eye tracker data must be mapped to the 2D display image coordinates.

In general, if $x' \in [a, b]$ needs to be mapped to the range $[c, d]$, we have:

$$x = c + \frac{(x' - a)(d - c)}{(b - a)}. \tag{7.1}$$

This is a linear mapping between two (one-dimensional) coordinate systems (or lines in this case). Equation (7.1) has a straightforward interpretation:

1. The value x' is translated (shifted) to its origin, by subtracting a.
2. The value $(x' - a)$ is then normalized by dividing through by the range $(b - a)$.
3. The normalized value $(x' - a)/(b - a)$ is then scaled to the new range $(d - c)$.
4. Finally, the new value is translated (shifted) to its proper relative location in the new range by adding c.

7.1.1 Mapping Screen Coordinates to the 3D Viewing Frustum

The 3D viewing frustum employed in the perspective viewing transformations is defined by the parameters `left, right, bottom, top, near, far`, e.g., as used in the `OpenGL` function call `glFrustum()`. Figure 7.1 shows the dimensions of the eye tracker screen (left) and the dimensions of the viewing frustum (right). Note that the eye tracker origin is the top-left of the screen and the viewing frustum's origin is bottom-left (this is a common discrepancy between imaging and graphics). To convert the eye tracker coordinates (x', y') to graphics coordinates (x, y), using Eq. (7.1), we have:

[1] Equivalently, the rotation quaternion may be used, if these data are available from the head tracker.

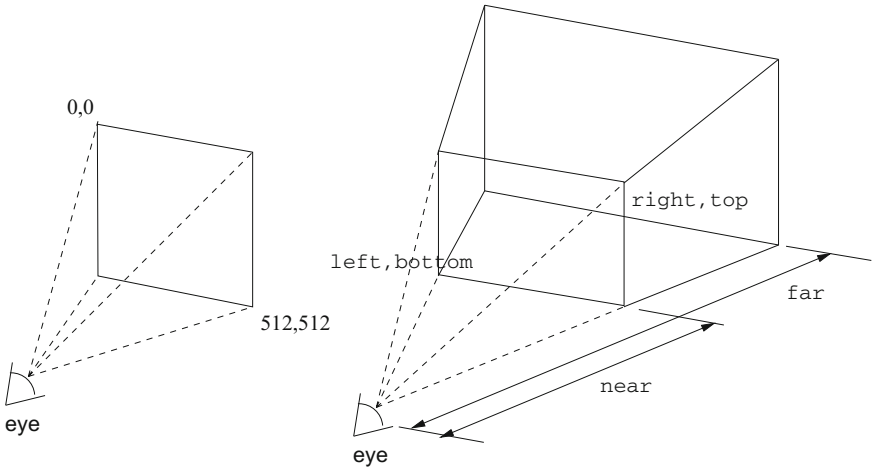

Fig. 7.1 Eye tracker to VR mapping

$$x = \texttt{left} + \frac{x'(\texttt{right} - \texttt{left})}{512} \tag{7.2}$$

$$y = \texttt{bottom} + \frac{(512 - y')(\texttt{top} - \texttt{bottom})}{512}. \tag{7.3}$$

Note that the term $(512 - y')$ in Eq. (7.3) handles the y-coordinate mirror transformation so that the top-left origin of the eye tracker screen is converted to the bottom-left of the viewing frustum.

If typical dimensions of the near plane of the graphics viewing frustum are 640 $\times$ 480, with the origin at (0, 0), then Eqs. (7.2) and (7.3) reduce to:

$$x = \frac{x'(640)}{512} = x'(1.3) \tag{7.4}$$

$$y = \frac{(512 - y')(480)}{512} = (512 - y')(0.9375). \tag{7.5}$$

7.1.2 Mapping Screen Coordinates to the 2D Image

The linear mapping between eye tracker screen coordinates and 2D image coordinates is handled similarly. If the image dimensions are 640 $\times$ 480, then Eqs. (7.4) and (7.5) are used without change. Note that an image with dimensions 600 $\times$ 450 appears to fit the display of the TV in the Clemson VRET lab better.[2] In this case, Eqs. (7.4) and (7.5) become:

[2] A few of the pixels of the image do not fit on the TV display, possibly due to the NTSC flicker-free filter used to encode the SGI console video signal.

$$x = \frac{x'(600)}{512} = x'(1.171875) \tag{7.6}$$

$$y = \frac{(512 - y')(450)}{512} = (512 - y')(0.87890625). \tag{7.7}$$

7.1.3 Measuring Eye Tracker Screen Coordinate Extents

The above coordinate mapping procedures assume that the eye tracker coordinates are in the range [0, 511]. In reality, the *usable*, or effective coordinates will be dependent on: (a) the size of the application window, and (b) the position of the application window. For example, if an image display program runs wherein an image is displayed in a 600 × 450 window, and this window is positioned arbitrarily on the graphics console, then the eye tracking coordinates of interest are restricted only to the area covered by the application window.

The following example illustrates the eye tracker/application program coordinate mapping problem using a schematic depicting the scene imaging display produced by an eye tracker manufactured by ISCAN. The ISCAN eye tracker is a fairly popular video-based corneal-reflection eye tracker which provides an image of the scene viewed by the subject and is seen by the operator on small ISCAN black and white scene monitors. Most eye trackers provide this kind of functionality. Because the application program window is not likely to cover the entire scene display (as seen in the scene monitors), only a restricted range of eye tracker coordinates will be of relevance to the eye tracking experiment. To use the ISCAN as an example, consider a 600 × 450 image display application. Once the window is positioned so that it fully covers the viewing display, it may appear slightly off-center on the ISCAN black/white monitor, as sketched in Fig. 7.2.

The trick to the calculation of the proper mapping between eye tracker and application coordinates is the measurement of the application window's extents, in the eye tracker's reference frame. This is accomplished by using the eye tracker itself. One of the eye tracker's most useful features, if available for this purpose, is its ability to move its crosshair cursor. The software may allow cursor fine (pixel by pixel) or coarse (in jumps of ten pixels) movement. Furthermore, a data screen at the bottom of the scene monitor may indicate the coordinates of the cursor, relative to the eye tracker's reference frame.

To obtain the extents of the application window in the eye tracker's reference frame, simply move the cursor to the corners of the application window. Use these coordinates in the above mapping formulas. The following measurement "recipe" should provide an almost exact mapping, and only needs to be performed once. Assuming the application window's dimensions are fixed, the mapping obtained from this procedure can be hardcoded into the application. Here are the steps:

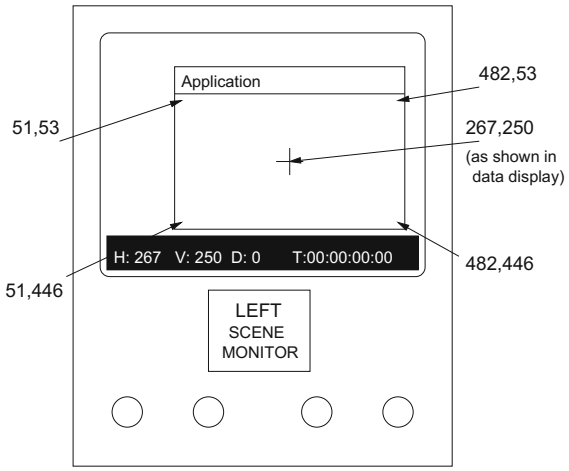

Fig. 7.2 Example mapping measurement

1. Position your display window so that it covers the display fully, e.g., in the case of the image stimulus, the window should just cover the entire viewing (e.g., TV or virtual reality head-mounted display (HMD)) display.
2. Use the eye tracker's cursor positioning utility to measure the viewing area's extents (toggle between course and fine cursor movement).
3. Calculate the mapping.
4. In Reset mode, position the eye tracker cursor in the middle of the display.

An important consequence of the above is that following the mapping calculation, it should be possible to always position the application window in the same place, provided that the program displays the calibration point obtained from the eye tracker mapped to local image coordinates. When the application starts up (and the eye tracker is on and in Reset mode), simply position the application so that the central calibration points (displayed when the tracker is in Reset mode) line up.

To conclude this example, assume that if the ISCAN cursor is moved to the corners of the drawable application area, the measurements would appear as shown in Fig. 7.2. Based on those measurements, the mapping is:

$$x = \frac{x' - 51}{(482 - 51 + 1)}(600) \tag{7.8}$$

$$y = 449 - \frac{y' - 53}{(446 - 53 + 1)}(450). \tag{7.9}$$

The central point on the ISCAN display is (267, 250). Note that y is subtracted from 449 to take care of the image/graphics vertical origin flip.

7.2 Mapping Flock of Birds Tracker Coordinates

In virtual reality, the position and orientation of the head is typically delivered by a real-time head tracker. In our case, we have a flock of birds d.c. electromagnetic tracker from Ascension. The tracker reports 6 degree of freedom information regarding sensor position and orientation. The latter is given in terms of Euler angles. Euler angles determine the orientation of a vector in three-space by specifying the required rotations of the origin's coordinate axes. These angles are known by several names, but in essence each describes a particular rotation angle about one of the principal axes. Common names describing Euler angles are given in Table 7.1 and the geometry is sketched in Fig. 7.3, where roll, pitch, and yaw angles are represented by R, E, and A, respectively. Each of these rotations is represented by the familiar homogeneous rotation matrices:

$$\text{Roll}\,(\texttt{rot z}) \;=\; \mathbf{R}_z = \begin{bmatrix} \cos R & \sin R & 0 & 0 \\ -\sin R & \cos R & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\text{Pitch}\,(\texttt{rot x}) \;=\; \mathbf{R}_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos E & \sin E & 0 \\ 0 & -\sin E & \cos E & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\text{Yaw}\,(\texttt{rot y}) \;=\; \mathbf{R}_y = \begin{bmatrix} \cos A & 0 & -\sin A & 0 \\ 0 & 1 & 0 & 0 \\ \sin A & 0 & \cos A & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

The composite 4×4 matrix, containing all of the above transformations rolled into one, is:

$$\mathbf{M} = \mathbf{R}_z\mathbf{R}_x\mathbf{R}_y = \begin{bmatrix} \cos R\cos A + \sin R\sin E\sin A & \sin R\cos E & -\cos R\sin A + \sin R\sin E\cos A & 0 \\ -\sin R\cos A + \cos R\sin E\sin A & \cos R\cos E & \sin R\sin A + \cos R\sin E\cos A & 0 \\ \cos E\sin A & -\sin E & \cos E\cos A & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

The FOB delivers a similar 4×4 matrix,

Table 7.1 Euler angles

`rot y`	`rot x`	`rot z`
Yaw	Pitch	Roll
Azimuth	Elevation	Roll
Longitude	Latitude	Roll

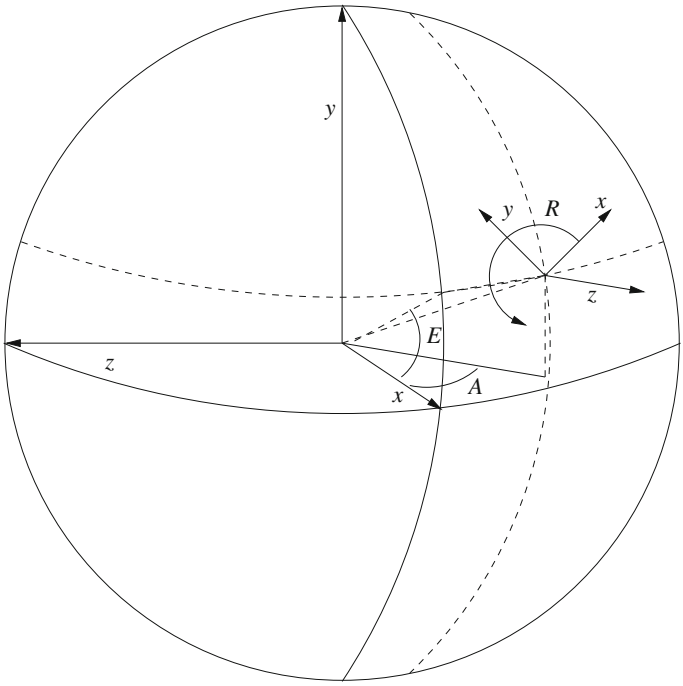

Fig. 7.3 Euler angles

$$\mathbf{F} = \begin{bmatrix} \cos E \cos A & \cos E \sin A & -\sin E & 0 \\ -\cos R \sin A + \sin R \sin E \cos A & \cos R \cos A + \sin R \sin E \sin A & \sin R \cos E & 0 \\ \sin R \sin A + \cos R \sin E \cos A & -\sin R \cos A + \cos R \sin E \sin A & \cos R \cos E & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

where the matrix elements are slightly rearranged, such that

$$\mathbf{M}[i, j] = \mathbf{F}[(i + 1)\%3][(j + 1)\%3],$$

e.g., row 1 of matrix **M** is now row 2 in matrix **F**, row 2 is now row 3, and row 3 is now row 1. Columns are interchanged similarly, where column 1 of matrix **M** is now column 2 in matrix **F**, column 2 is now column 3, and column 3 is now column 1. This "off-by-one" shift present in the Bird matrix may be due to the non-C style indexing which starts at 1 instead of 0.

7.2.1 Obtaining the Transformed View Vector

To calculate the transformed view vector $\mathbf{v}'$, assume the initial view vector is $\mathbf{v} = (0, 0, -1)$ (looking down the z-axis), and apply the composite transformation (in homogeneous coordinates)[3]:

$$\mathbf{vM} = \begin{bmatrix} 0 & 0 & -1 & 1 \end{bmatrix} * \begin{bmatrix} \cos R \cos A + \sin R \sin E \sin A & \sin R \cos E & -\cos R \sin A + \sin R \sin E \cos A & 0 \\ -\sin R \cos A + \cos R \sin E \sin A & \cos R \cos E & \sin R \sin A + \cos R \sin E \cos A & 0 \\ \cos E \sin A & -\sin E & \cos E \cos A & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -\cos E \sin A & \sin E & \cos E \cos A & 1 \end{bmatrix},$$

which is simply the third row of $\mathbf{M}$, negated. By inspection of the Bird matrix $\mathbf{F}$, the view vector is simply obtained by selecting appropriate entries in the matrix; i.e.,

$$\mathbf{v}' = \begin{bmatrix} -\mathbf{F}[0, 1] & -\mathbf{F}[0, 2] & \mathbf{F}[0, 0] & 1 \end{bmatrix}. \tag{7.10}$$

7.2.2 Obtaining the Transformed up Vector

The transformed up vector is obtained in a similar manner to the transformed view vector. To calculate the transformed up vector $\mathbf{u}'$, assume the initial up vector is $\mathbf{u} = (0, 1, 0)$ (looking up the y-axis), and apply the composite transformation (in homogeneous coordinates):

$$\mathbf{uM} = \begin{bmatrix} 0 & 1 & 0 & 1 \end{bmatrix} * \begin{bmatrix} \cos R \cos A + \sin R \sin E \sin A & \sin R \cos E & -\cos R \sin A + \sin R \sin E \cos A & 0 \\ -\sin R \cos A + \cos R \sin E \sin A & \cos R \cos E & \sin R \sin A + \cos R \sin E \cos A & 0 \\ \cos E \sin A & -\sin E & \cos E \cos A & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -\sin R \cos A + \cos R \sin E \sin A & \cos R \cos E & \sin R \sin A + \cos R \sin E \cos A & 1 \end{bmatrix}.$$

By inspection of the Bird matrix $\mathbf{F}$, the transformed up vector is simply obtained by selecting appropriate entries in the matrix; i.e.,

$$\mathbf{u}' = \begin{bmatrix} \mathbf{F}[2, 1] & \mathbf{F}[2, 2] & \mathbf{F}[2, 0] & 1 \end{bmatrix}. \tag{7.11}$$

Because of our setup in the lab, the z-axis is on the opposite side of the Bird transmitter (behind the Bird emblem on the transmitter). For this reason, the z-component of the up vector is negated, i.e.,

$$\mathbf{u}' = \begin{bmatrix} \mathbf{F}[2, 1] & \mathbf{F}[2, 2] & -\mathbf{F}[2, 0] & 1 \end{bmatrix}. \tag{7.12}$$

[3]Recall trigonometric identities: $\sin(-\theta) = -\sin(\theta)$ and $\cos(-\theta) = \cos(\theta)$.

Note that the negation of the z-component of the transformed view vector does not make a difference because the term is a product of cosines.

7.2.3 Transforming an Arbitrary Vector

To transform an arbitrary vector, an operation similar to the transformations of the up and view vectors is performed. To calculate the transformed arbitrary vector, $\mathbf{w} = \begin{bmatrix} x & y & z & 1 \end{bmatrix}$, apply the composite transformation by multiplying by the transformation matrix $\mathbf{M}$ (in homogeneous coordinates):

$$\mathbf{wM} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} * \begin{bmatrix} \cos R \cos A + \sin R \sin E \sin A & \sin R \cos E & -\cos R \sin A + \sin R \sin E \cos A & 0 \\ -\sin R \cos A + \cos R \sin E \sin A & \cos R \cos E & \sin R \sin A + \cos R \sin E \cos A & 0 \\ \cos E \sin A & -\sin E & \cos E \cos A & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

which gives transformed vector $\mathbf{w}'$,

$$\mathbf{w}' = \begin{bmatrix} x\mathbf{M}[1,1] + y\mathbf{M}[2,1] + z\mathbf{M}[3,1] \\ x\mathbf{M}[1,2] + y\mathbf{M}[2,2] + z\mathbf{M}[3,2] \\ x\mathbf{M}[1,3] + y\mathbf{M}[2,3] + z\mathbf{M}[3,3] \\ 1 \end{bmatrix}^T .$$

To use the Bird matrix, there is unfortunately no simple way to select the appropriate matrix elements to directly obtain $\mathbf{w}'$. Probably the best bet would be to undo the "off-by-one" shift present in the Bird matrix. On the other hand, hardcoding the solution may be the fastest method. This rather inelegant, but luckily localized, operation looks like this:

$$\mathbf{w}' = \begin{bmatrix} x\mathbf{F}[1,1] + y\mathbf{F}[2,1] + z\mathbf{F}[0,1] \\ x\mathbf{F}[1,2] + y\mathbf{F}[2,2] + z\mathbf{F}[0,2] \\ x\mathbf{F}[1,0] + y\mathbf{F}[2,0] + z\mathbf{F}[0,0] \\ 1 \end{bmatrix}^T . \tag{7.13}$$

Furthermore, as in the up vector transformation, it appears that the negation of the z component may also be necessary. If so, the above equation will need to be rewritten as

$$\mathbf{w}' = \begin{bmatrix} x\mathbf{F}[1,1] + y\mathbf{F}[2,1] + z\mathbf{F}[0,1] \\ x\mathbf{F}[1,2] + y\mathbf{F}[2,2] + z\mathbf{F}[0,2] \\ -(x\mathbf{F}[1,0] + y\mathbf{F}[2,0] + z\mathbf{F}[0,0]) \\ 1 \end{bmatrix}^T . \tag{7.14}$$

7.3 3D Gaze Point Calculation

The calculation of the gaze point in three-space depends only on the relative positions of the two eyes in the horizontal axis. The parameters of interest here are the three-dimensional virtual coordinates of the gaze point, (x_g, y_g, z_g), which can be determined from traditional stereo geometry calculations. Figure 7.4 illustrates the basic binocular geometry. Helmet tracking determines helmet position and orthogonal directional and up vectors, which determine viewer-local coordinates shown in the diagram. The helmet position is the origin (x_h, y_h, z_h), the helmet directional vector is the optical (viewer-local) z-axis, and the helmet up vector is the viewer-local y-axis.

Given instantaneous, eye tracked, viewer-local coordinates (mapped from eye tracker screen coordinates to the near view plane coordinates), (x_l, y_l) and (x_r, y_r) in the left and right view planes, at focal distance f along the viewer-local z-axis, we can determine viewer-local coordinates of the gaze point (x_g, y_g, z_g) by deriving the stereo equations parametrically. First, express both the left and right view lines in terms of the linear parameter s. These lines originate at the eye centers $(x_h - b/2, y_h, z_h)$ and $(x_h + b/2, y_h, z_h)$ and pass through (x_l, y_l, f) and (x_r, y_r, f), respectively. The left view line is (in vector form):

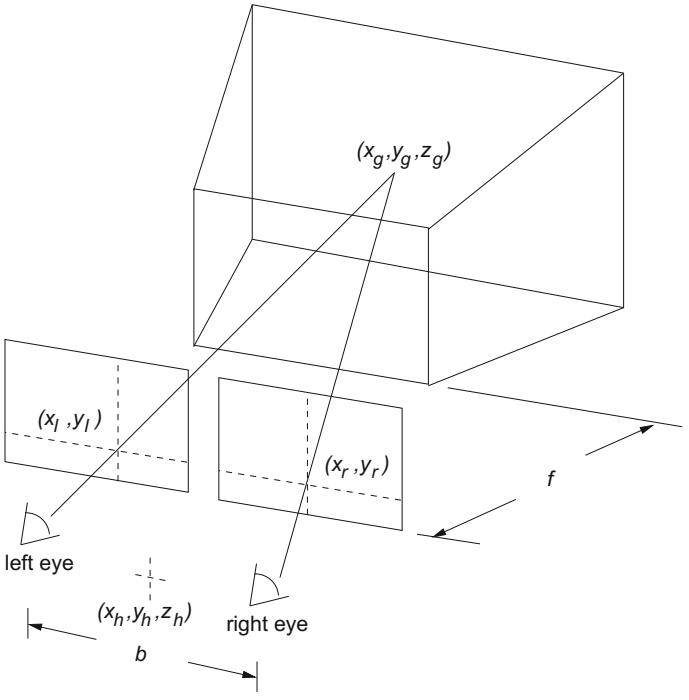

Fig. 7.4 Basic binocular geometry

$$\left[\,(1-s)(x_h - b/2) + sx_l \;\; (1-s)y_h + sy_l \;\; (1-s)z_h + sf\,\right] \tag{7.15}$$

and the right view line is (in vector form):

$$\left[\,(1-s)(x_h + b/2) + sx_r \;\; (1-s)y_h + sy_r \;\; (1-s)z_h + sf\,\right], \tag{7.16}$$

where b is the disparity distance between the left and right eye centers, and f is the distance to the near viewing plane. To find the central view line originating at the local center (x_h, y_h, z_h), calculate the intersection of the left and right view lines by solving for s using the x-coordinates of both lines, as given in Eqs. (7.15) and (7.16):

$$\begin{aligned}
(1-s)(x_h - b/2) + sx_l &= (1-s)(x_h + b/2) + sx_r \\
(x_h - b/2) - s(x_h - b/2) + sx_l &= (x_h + b/2) - s(x_h + b/2) + sx_r \\
s(x_l - x_r + b) &= b \\
s &= \frac{b}{x_l - x_r + b}.
\end{aligned} \tag{7.17}$$

The interpolant s is then used in the parametric equation of the central view line, to give the gaze point at the intersection of both view lines:

$$\begin{aligned}
x_g &= (1-s)x_h + s((x_l + x_r)/2) \\
y_g &= (1-s)y_h + s((y_l + y_r)/2) \\
z_g &= (1-s)z_h + sf,
\end{aligned}$$

giving:

$$x_g = \left(1 - \frac{b}{x_l - x_r + b}\right) x_h + \left(\frac{b}{x_l - x_r + b}\right)\left(\frac{x_l + x_r}{2}\right) \tag{7.18}$$

$$y_g = \left(1 - \frac{b}{x_l - x_r + b}\right) y_h + \left(\frac{b}{x_l - x_r + b}\right)\left(\frac{y_l + y_r}{2}\right) \tag{7.19}$$

$$z_g = \left(1 - \frac{b}{x_l - x_r + b}\right) z_h + \left(\frac{b}{x_l - x_r + b}\right) f. \tag{7.20}$$

Eye positions (at viewer local coordinates $(x_h - b/2, y_h, z_h)$ and $(x_h + b/2, y_h, z_h)$), the gaze point, and an up vector orthogonal to the plane of the three points then determine the view volume appropriate for display to each eye screen.

The gaze point, as defined above, is given by the addition of a scaled offset to the view vector originally defined by the helmet position and central view line in virtual world coordinates.[4] The gaze point can be expressed parametrically as a point on a

[4]Note that the vertical eye tracked coordinates y_l and y_r are expected to be equal (because gaze coordinates are assumed to be epipolar); the vertical coordinate of the central view vector defined by $(y_l + y_r)/2$ is somewhat extraneous; either y_l or y_r would do for the calculation of the gaze vector. However, because eye tracker data are also expected to be noisy, this averaging of the vertical coordinates enforces the epipolar assumption.

ray with origin (x_h, y_h, z_h) and the helmet position, with the ray emanating along a vector scaled by parameter s. That is, rewriting Eq. (7.18) through (7.20), we have:

$$\begin{aligned} x_g &= x_h + s\left(\frac{x_l + x_r}{2} - x_h\right) \\ y_g &= y_h + s\left(\frac{y_l + y_r}{2} - y_h\right) \\ z_g &= z_h + s\,(f - z_h) \end{aligned}$$

or, in vector notation,

$$\mathbf{g} = \mathbf{h} + s\mathbf{v}, \tag{7.21}$$

where $\mathbf{h}$ is the head position, $\mathbf{v}$ is the central view vector, and s is the scale parameter as defined in Eq. (7.17). Note that the view vector used here is not related to the view vector given by the head tracker. It should be noted that the view vector $\mathbf{v}$ is obtained by subtracting the helmet position from the midpoint of the eye tracked x-coordinate and focal distance to the near view plane; i.e.,

$$\begin{aligned} \mathbf{v} &= \begin{bmatrix} (x_l + x_r)/2 \\ (y_l + y_r)/2 \\ f \end{bmatrix} - \begin{bmatrix} x_h \\ y_h \\ z_h \end{bmatrix} \\ &= \mathbf{m} - \mathbf{h}, \end{aligned}$$

where $\mathbf{m}$ denotes the left and right eye coordinate midpoint. To transform the vector $\mathbf{v}$ to the proper (instantaneous) head orientation, this vector should be normalized, then multiplied by the orientation matrix returned by the head tracker (see Sect. 7.2 in general and Sect. 7.2.3 in particular). This new vector, call it $\mathbf{m}'$, should be substituted for $\mathbf{m}$ above to define $\mathbf{v}$ for use in Eq. (7.21); i.e.,

$$\mathbf{g} = \mathbf{h} + s(\mathbf{m}' - \mathbf{h}). \tag{7.22}$$

7.3.1 Parametric Ray Representation of Gaze Direction

Equation (7.22) gives the coordinates of the gaze point through a parametric representation (e.g., a point along a line) such that the depth of the three-dimensional point of regard in world coordinates is valid only if $s > 0$. Given the gaze point $\mathbf{g}$ and the location of the helmet $\mathbf{h}$, we can obtain just the three-dimensional gaze vector v that specifies the direction of gaze (but not the actual fixation point). This direction vector is given by:

$$\begin{aligned} v &= \mathbf{g} - \mathbf{h} && (7.23)\\ &= (\mathbf{h} + s\mathbf{v}) - \mathbf{h} && (7.24)\\ &= s\mathbf{v} && (7.25)\\ &= \left(\frac{b}{x_l - x_r + b}\right) \begin{bmatrix} (x_l + x_r)/2 - x_h \\ (y_l + y_r)/2 - y_h \\ (f - z_h) \end{bmatrix}, && (7.26) \end{aligned}$$

where $\mathbf{v}$ is defined as either $\mathbf{m} - \mathbf{h}$ as before, or as $\mathbf{m}' - \mathbf{h}$ as in Eq. (7.22). Given the helmet position $\mathbf{h}$ and the gaze direction v, we can express the gaze direction via a parametric representation of a ray using a linear interpolant t:

$$\text{gaze}(t) = \mathbf{h} + tv, \quad t > 0, \tag{7.27}$$

where h is the ray's origin (a point; the helmet position), and v is the ray direction (the gaze vector). (Note that adding h to tv results in the original expression of the gaze point $\mathbf{g}$ given by Eq. (7.21), provided $t = 1$.) The formulation of the gaze direction given by Eq. (7.27) can then be used for testing virtual fixation coordinates via traditional ray/polygon intersection calculations commonly used in ray-tracing.

7.4 Virtual Gaze Intersection Point Coordinates

In 3D eye tracking studies, we are often interested in knowing the location of one's gaze, or more importantly one's fixation, relative to some feature in the scene. In VR applications, we'd like to calculate the fixation location in the virtual world and thus identify the object of interest. The identification of the object of interest can be accomplished following traditional ray/polygon intersection calculations, as employed in ray-tracing (Glassner 1989).

The fixated object of interest is the one closest to the viewer that intersects the gaze ray. This object is found by testing all polygons in the scene for intersection with the gaze ray. The polygon closest to the viewer is then assumed to be the one fixated by the viewer (assuming all polygons in the scene are opaque).

7.4.1 Ray/Plane Intersection

The calculation of an intersection between a ray and all polygons in the scene is usually obtained via a parametric representation of the ray; e.g.,

$$\text{ray}(t) = r_o + t\mathbf{r_d}, \tag{7.28}$$

where r_o defines the ray's origin (a point), and $\mathbf{r_d}$ defines the ray direction (a vector). Note the similarity between Eqs. (7.28) and (7.27); there, h is the head position and v is the gaze direction. To find the intersection of the ray with a polygon, calculate the interpolant value where the ray intersects each polygon, and examine all the intersections where $t > 0$. If $t < 0$, the object may intersect the ray, but behind the viewer.

Recall the plane equation $Ax + By + Cz + D = 0$, where $A^2 + B^2 + C^2 = 1$; i.e., A, B, and C define the plane normal. To obtain the ray/polygon intersection, substitute Eq. (7.28) into the plane equation:

$$A(x_o + tx_d) + B(x_o + ty_d) + C(z_o + tz_d) + D = 0 \tag{7.29}$$

and solve for t:

$$t = -\frac{Ax_o + Bx_o + Cz_o + D}{Ax_d + By_d + Cz_d} \tag{7.30}$$

or, in vector notation:

$$t = \frac{-(\mathbf{N} \cdot r_o + D)}{\mathbf{N} \cdot \mathbf{r_d}}. \tag{7.31}$$

A few observations of the above simplify the implementation.

1. Here $\mathbf{N}$, the face normal, is really $-\mathbf{N}$, because what we're doing is calculating the angle between the ray and face normal. To get the angle, we need both ray and normal to be pointing in the same relative direction. This situation is depicted in Fig. 7.5.
2. In Eq. (7.31), the denominator will cause problems in the implementation should it evaluate to 0. However, if the denominator is 0 (i.e., if $\mathbf{N} \cdot \mathbf{r_d} = 0$), then the cosine between the vectors is 0, which means that the angle between the two vectors is 90° which means the ray and plane are parallel and don't intersect. Thus, to avoid dividing by zero, and to speed up the computation, evaluate the denominator first. If it is sufficiently close to zero, don't evaluate the intersection further; we know the ray and polygon will not intersect.
3. Point 2 above can be further exploited by noting that if the dot product is greater than 0, then the surface is hidden to the viewer.

The first part of the intersection algorithm requires computation of the denominator, $v_d = \mathbf{N} \cdot \mathbf{r_d}$ followed by the numerator, $v_o = -(\mathbf{N} \cdot r_o + D)$ so that $t = v_o/v_d$ provided that $v_d < 0$.

In the algorithm, the intersection parameter t defines the point of intersection along the ray at the plane defined by the normal $\mathbf{N}$. That is, if $t > 0$, then the point of intersection p is given by:

$$p = r_o + t\mathbf{r_d}. \tag{7.32}$$

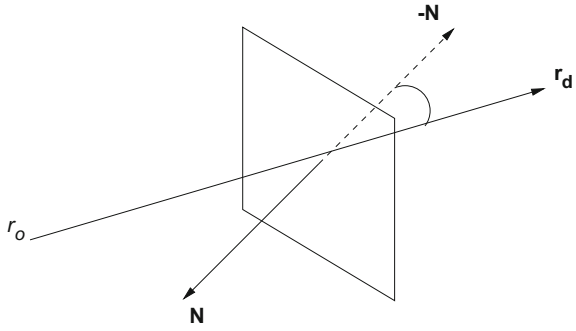

Fig. 7.5 Ray/plane geometry

Note that the above only gives the intersection point of the ray and the (infinite!) plane defined by the polygon's face normal. Because the normal defines a plane of infinite extent, we need to test the point p to see if it lies within the confines of the polygonal region, which is defined by the polygon's edges. This is essentially the "point-in-polygon" problem.

7.4.2 Point-in-Polygon Problem

To test whether a point p lies inside a polygon (defined by its plane equation which specifies a plane of finite extent), we need to test the point against all edges of the polygon. To do this, the following algorithm is used.

For each edge:

1. Get the plane perpendicular to the face normal $\mathbf{N}$, which passes through the edge's two vertices A and B. The perpendicular face normal $\mathbf{N}'$ is obtained by calculating the cross product of the original face normal with the edge; i.e.,

$$\mathbf{N}' = \mathbf{N} \times (B - A),$$

   where the face vertices A and B are specified in counterclockwise order. This is shown in Fig. 7.6.
2. Get the perpendicular plane's equation by calculating D using either of A or B as the point on the plane (e.g., plug in either A or B into the plane equation; then solve for D).
3. Test point p to see if it lies "above" or "below" the perpendicular plane. This is done by plugging p into the perpendicular plane's equation and testing the result. If the result is greater than 0, then p is "above" the plane.
4. If p is "above" all perpendicular planes, as defined by successive pairs of vertices, then the point is "boxed in" by all the polygon's edges, and so must lie inside the polygon as originally defined by its face normal $\mathbf{N}$.

Note that this is just one solution to the point-in-polygon problem; other, possibly faster, algorithms may be available.

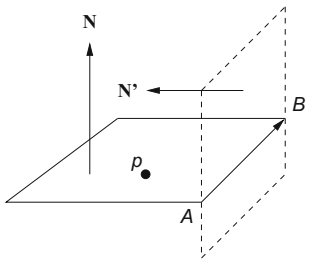

Fig. 7.6 Point-in-polygon geometry

The calculated intersection points p, termed here Gaze Intersection Points (GIPs) for brevity, are each found on the closest polygon to the viewer intersecting the gaze ray, assuming all polygons are opaque. The resulting ray-casting algorithm generates a scanpath constrained to lie on polygonal regions within the virtual environment. Eye movement analysis is required to identify fixation points in the environment (see Chap. 13).

7.5 Data Representation and Storage

Data collection is straightforward. For 2D imaging applications, the point of regard can be recorded, along with the timestamp of each sample. Listing 7.1 shows a pseudocode sample data structure suitable for recording the point of regard. Because the number of samples may be rather large, depending on sampling rate and duration of viewing trial, a linked list may be used to dynamically record the data. In Listing 7.1, the data type `queue` denotes a linked list pointer.

```
typedef struct _PORNode {
  _PORNode*      link; // linked list node
  int            x,y;  // POR data
  double         t;    // timestamp (usually in ms)
} PORnode;
```

Listing 7.1 2D imaging point of regard data structure

For 3D VR applications, the simplest data structure to record the calculated gaze point is similar to the 2D structure, with the addition of the third z component. Data captured (i.e., the three-dimensional gaze point) can then be displayed for review of the session statically in the VR environment in which it was recorded, as shown in Fig. 7.7. In Fig. 7.7 consecutive gaze points have been joined by straight line segments to display a three-dimensional scanpath. Note that in this visualization it may be easy to misinterpret the scanpath data with the position of the head. To disambiguate the two, head position (and tilt angle) may also be stored/displayed.

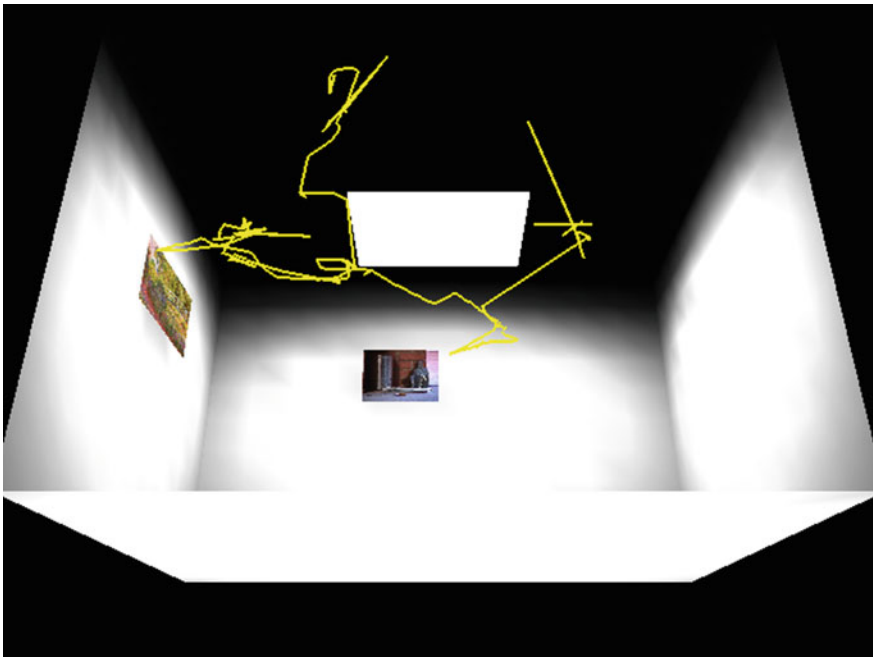

Fig. 7.7 Example of three-dimensional gaze point captured in VR. Courtesy of Tom Auchter and Jeremy Barron. Reproduced with permission, Clemson University

There is one fairly important aspect of data storage which is worth mentioning, namely, proper labeling of the data. Because eye movement experiments tend to generate voluminous amounts of data, it is imperative to properly label data files. Files should encode the number of the subject viewing the imagery, the imagery itself, the trial number, and the state of the data (raw or processed). One of the easiest methods for encoding this information is in the data filename itself (at least on file systems that allow long filenames).

7.6 Summary and Further Reading

This chapter focused on the main programming aspects of proper collection of raw eye movement data from the eye tracker, including appropriate mapping of raw eye movement coordinates. Coordinate mapping is crucial for both eye tracker calibration and subsequent eye movement coordination registration in the application reference frame.

Two important programming components have been omitted from this chapter, namely, a mechanism for user interaction and for stimulus display. Generally, stimulus display requires knowledge of some sort of graphical Application Program

Interface (API) that enables display of either digital images or graphics primitives (as required for VR). A particularly suitable API for both purposes is `OpenGL` (see http://www.opengl.org). For user interface development, particularly if developing on a UNIX or Linux platform, either the GTK Graphical User Interface (GUI) or Qt API is recommended. GTK is the freely available GNU Toolkit (see http://www.gtk.org). Qt is available from http://trolltech.com.