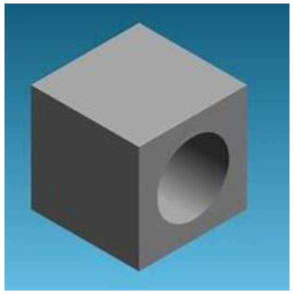
COSC 414/519I: Computer Graphics

2023W2

Shan Du

Positioning of the Camera

- Create an isometric view of a cube
  - Start with a cube centered at the origin and aligned with the axes.
  - Because the default camera is in the middle of the cube, we want to move the cube away from the camera by a translation.
  - We obtain an isometric view when the camera is located symmetrically with respect to the three adjacent faces of the cube; for example, anywhere along the line from the origin through the point (1,1,1).

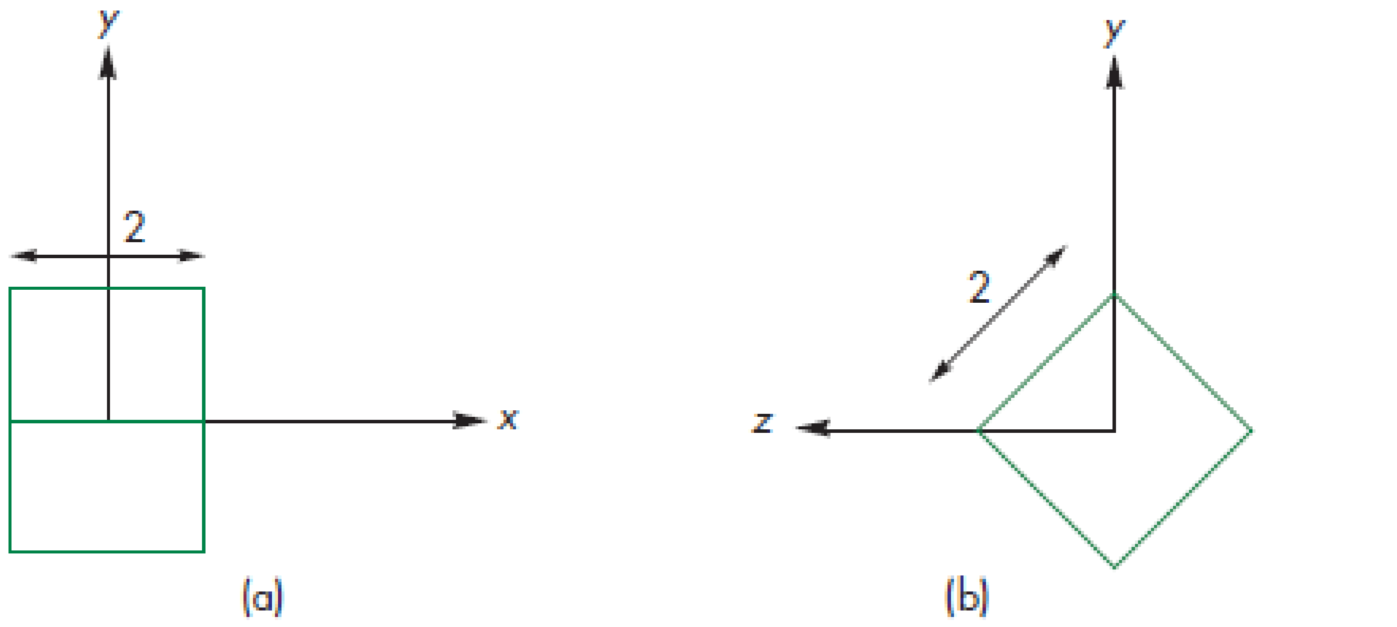


Positioning of the Camera

- Create an isometric view of a cube
  - We can rotate the cube and then move it away from the camera to achieve the desired view, or equivalently, move the camera away from the cube and then rotate it to point at the cube.
  - Starting with the default camera, suppose that we are now looking at the cube from somewhere on the positive *z*-axis.
  - We can obtain one of the eight isometric views-there is one for each vertex- by first rotating the cube about the *x*-axis until we see the two faces symmetrically.

Positioning of the Camera

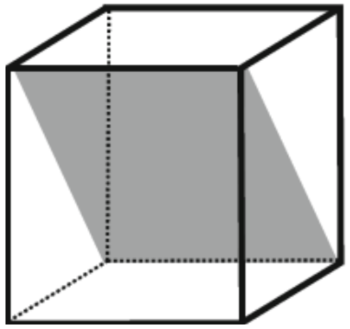
- Create an isometric view of a cube



Cube after rotation about x-axis. (a) View from positive z-axis. (b) View from positive x-axis.

Positioning of the Camera

- Create an isometric view of a cube
  - We obtain this view by rotating the cube about the *x*-axis by 45 degrees.
  - The second rotation is about the *y*-axis. We rotate the cube until we get the desired isometric.
  - The required angle of rotation is -35.26 degrees about the *y*-axis.
  - Finally, we move the camera away from the origin.



Positioning of the Camera

Model-view matrix is $M = TR_yR_x$

$$R = R_yR_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \sqrt{6}/3 & -\sqrt{3}/3 & 0 \\ 0 & \sqrt{3}/3 & \sqrt{6}/3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \sqrt{2}/2 & 0 & \sqrt{2}/2 & 0 \\ 0 & 1 & 0 & 0 \\ -\sqrt{2}/2 & 0 & \sqrt{2}/2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \sqrt{2}/2 & 0 & \sqrt{2}/2 & 0 \\ \sqrt{6}/6 & \sqrt{6}/3 & -\sqrt{6}/6 & 0 \\ -\sqrt{3}/3 & \sqrt{3}/3 & \sqrt{3}/3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

$$\mathrm{TR} = \begin{bmatrix} \sqrt{2}/2 & 0 & \sqrt{2}/2 & 0 \\ \sqrt{6}/6 & \sqrt{6}/3 & -\sqrt{6}/6 & 0 \\ -\sqrt{3}/3 & \sqrt{3}/3 & \sqrt{3}/3 & -d \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Positioning of the Camera

- Two Viewing APIs
  - The construction of the model-view matrix for an isometric view needs to compute the individual angles before specifying the transformations which is not convenient for an application program.
  - We can take a different approach to positioning the camera.
  - Our starting point is the object frame. We describe the camera’s position and orientation in this frame.

Positioning of the Camera

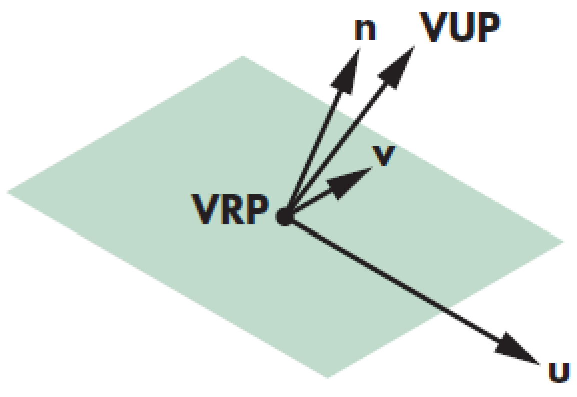
- Two Viewing APIs
  - The perspective or parallel projection is determined separately by the specification of the projection matrix.
  - The second part of the viewing process is often called the **normalization transformation**.

Positioning of the Camera

- We think of the camera as positioned initially at the origin, pointed in the negative *z* direction.
- Its desired location is centered at a point called the **view reference point** (**VRP**), whose position is given in the object frame.

```
var viewReferencePoint = vec4(x, y, z, 1);
```

- Next, we specify the orientation of the camera by dividing the specification into two parts: **view-plane normal** (**VPN**) and **view-up vector** (**VUP**).

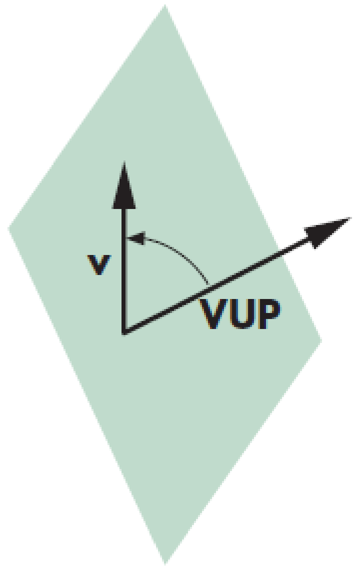


Camera frame.

Positioning of the Camera

```
var viewPlaneNormal = vec4(nx, ny, nz, 0);
var viewUp = vec4(vupX, vupY, vupZ, 0);
```

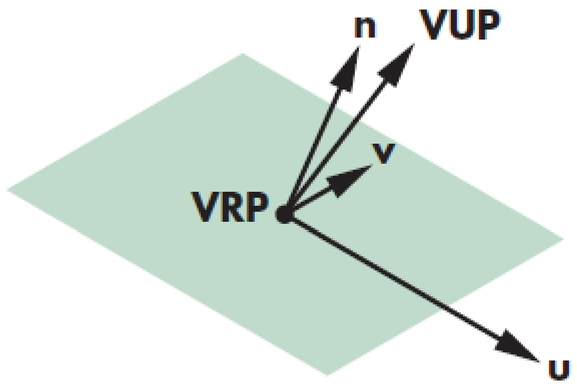
- We project the VUP vector on the view plane to obtain the up-direction vector ***v***.
- The vector ***v*** is orthogonal to ***n***
- We can use the cross product to obtain a third orthogonal direction ***u***.



Determination of the view-up vector.

Positioning of the Camera

- The new orthogonal coordinate system is usually referred to as either the **view-coordinate system** or the ***u-v-n* system**.
- With the addition of the VRP, we have the desired camera frame.



Camera frame.

Positioning of the Camera

- Assume the view reference point is $p = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$, the view-plane normal is $\mathrm{n} = \begin{bmatrix} n_x \\ n_y \\ n_z \\ 0 \end{bmatrix}$, and the view-up vector is $V_{up} = \begin{bmatrix} V_{up_x} \\ V_{up_y} \\ V_{up_z} \\ 0 \end{bmatrix}$.

Positioning of the Camera

- We construct a new frame with the view reference point as its origin, the view-plane normal as one coordinate direction, and two other orthogonal directions that we call ***u*** and ***v***.
- The original *x*, *y*, *z* axes become *u*, *v*, *n*, respectively.
- The view reference point can be handled through a simple translation $T(-x, -y, -z)$ from the viewing frame to the original origin.

Positioning of the Camera

- The rest of the model-view matrix is determined by a rotation so that the model-view matrix ***V*** is of the form $V = TR$.
- $n \cdot v = 0$
- v is the projection of V_{up} into the plane formed by n and V_{up} and thus $v = \alpha n + \beta V_{up}$.
- Ignore the length of vectors, we can set $\beta = 1$, then $\alpha = -\frac{V_{up} \cdot n}{n \cdot n}$ and $\mathrm{v} = V_{up} - \frac{V_{up} \cdot n}{n \cdot n} n$.

Positioning of the Camera

- $u = v \times n$
- Normalize *u*, *v*, *n* to obtain three unit-length vectors *u'*, *v'*, *n'*. The matrix

$$\mathbf{A} = \begin{bmatrix} u'_x & v'_x & n'_x & 0 \\ u'_y & v'_y & n'_y & 0 \\ u'_z & v'_z & n'_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

is the rotation matrix that orients a vector in the *u'*, *v'*, *n'* system with the original system.

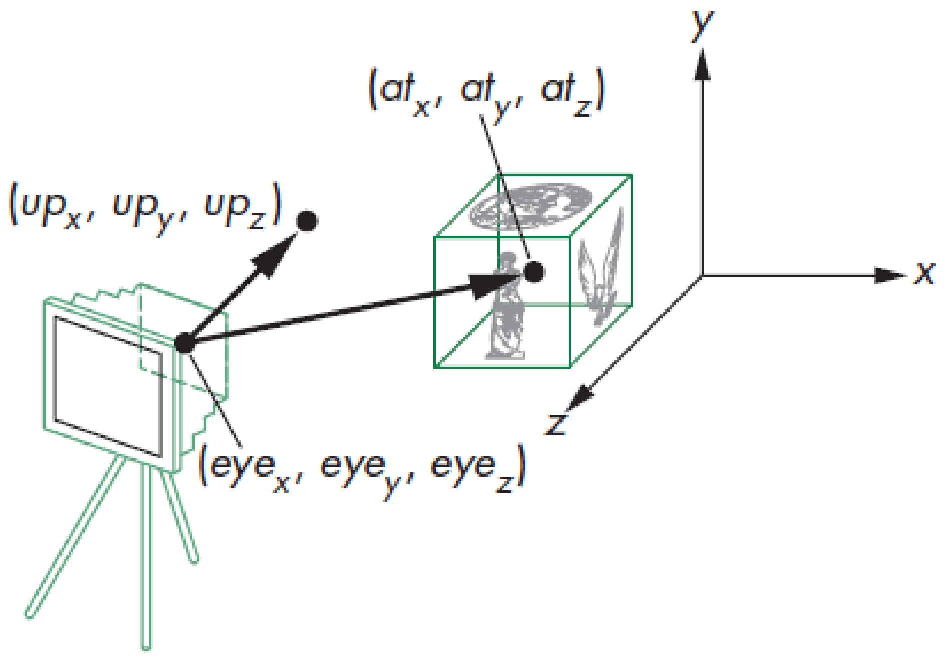
Positioning of the Camera

- To obtain the representation of vectors from the original system in the *u'*, *v'*, *n'* system, we need A^{-1}. The desired matrix is $\mathrm{R} = A^{-1} = A^{T}$.
- The final matrix is

$$\mathbf{V} = \mathbf{RT} = \begin{bmatrix} u'_x & u'_y & u'_z & -xu'_x - yu'_y - zu'_z \\ v'_x & v'_y & v'_z & -xv'_x - yv'_y - zv'_z \\ n'_x & n'_y & n'_z & -xn'_x - yn'_y - zn'_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Positioning of the Camera

- The Look-At Function
  - Eye point
  - At point

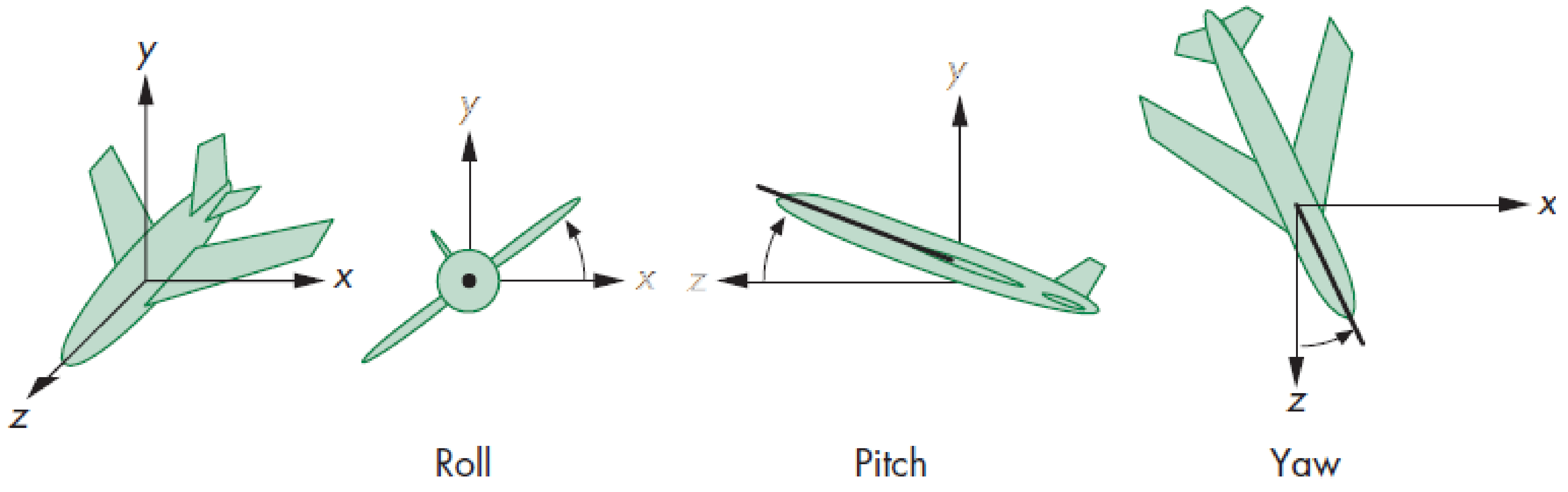


```
m = lookAt(eye, at, up)
```

Look-at positioning.

Positioning of the Camera

- Other Viewing APIs
  - In a flight simulator, the pilot usually use three angles – roll, pitch, and yaw – to specify the orientation.



Roll, pitch, and yaw.

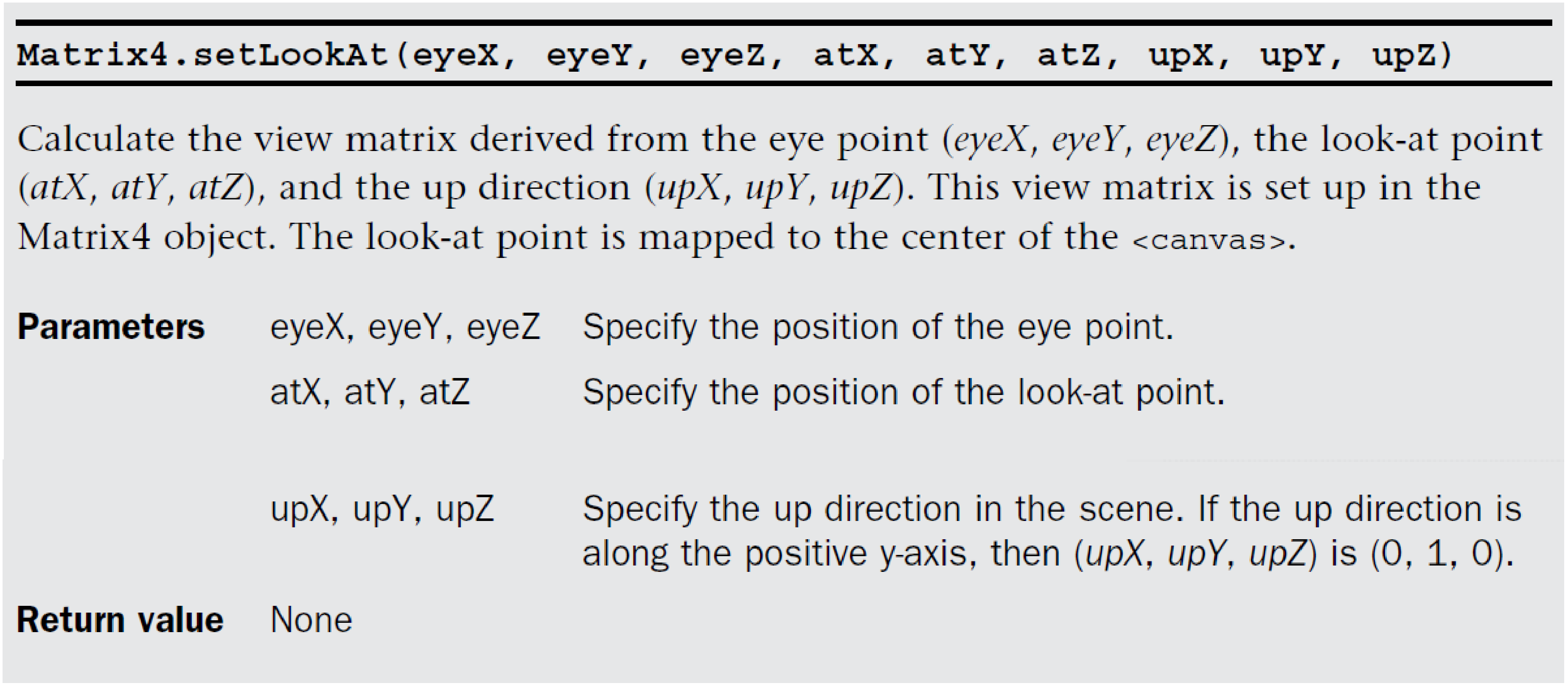
Eye Point, Look-At Point and Up Direction

- **Eye point**: This is the starting point from which the 3D space is viewed. The coordinates of this position are referred to as (*eyeX, eyeY, eyeZ*).
- **Look-at point**: This is the point at which you are looking and which determines the direction of the line of sight from the eye point. The coordinates of the look-at point are referred to as (*atX, atY, atZ*).
- **Up direction**: This determines the up direction in the scene that is being viewed from the eye point to the look-at point. The up direction is specified by three numbers representing the direction. The coordinates for this direction are referred to as (upX, upY, upZ).

Eye Point, Look-At Point and Up Direction

- In *cuon-matrix.js* , the method *Matrix4.setLookAt()* is defined to calculate the view matrix from the three items of information: eye point, look-at point and up direction.

Eye Point, Look-At Point and Up Direction



`Matrix4.setLookAt(eyeX, eyeY, eyeZ, atX, atY, atZ, upX, upY, upZ)`

Calculate the view matrix derived from the eye point (*eyeX*, *eyeY*, *eyeZ*), the look-at point (*atX*, *atY*, *atZ*), and the up direction (*upX*, *upY*, *upZ*). This view matrix is set up in the Matrix4 object. The look-at point is mapped to the center of the `<canvas>`.

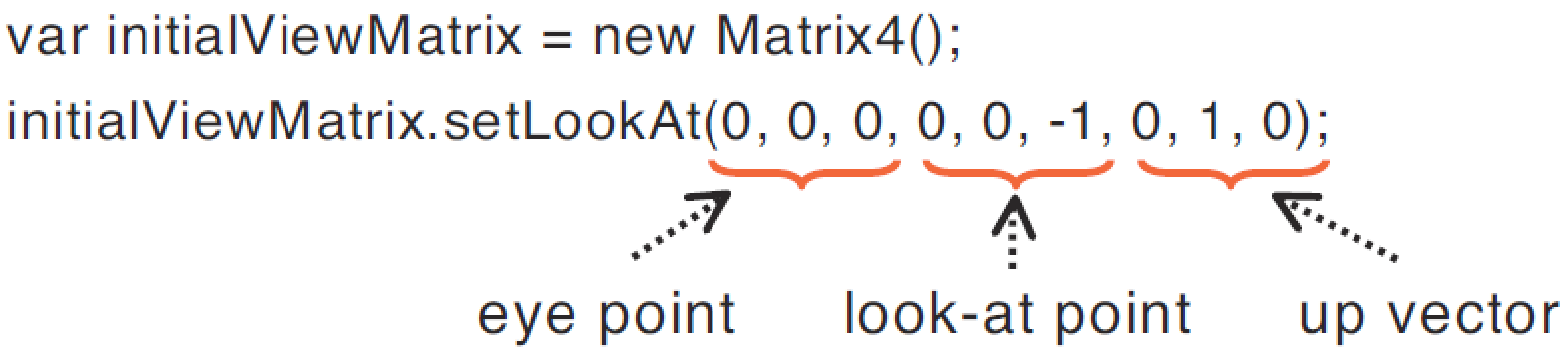
Parameters	eyeX, eyeY, eyeZ	Specify the position of the eye point.
	atX, atY, atZ	Specify the position of the look-at point.
	upX, upY, upZ	Specify the up direction in the scene. If the up direction is along the positive y-axis, then (*upX*, *upY*, *upZ*) is (0, 1, 0).
Return value	None	

Eye Point, Look-At Point and Up Direction

- In WebGL, the default settings when using *Matrix4.setLookAt()* are defined as follows:
  - The eye point is placed at (0, 0, 0) (that is, the origin of the coordinate system).
  - The look-at point is along the negative z-axis, so a good value is (0, 0, –1). The up direction is specified along the positive y-axis, so a good value is (0, 1, 0).

Eye Point, Look-At Point and Up Direction

- A view matrix representing the default settings in WebGL can be simply produced as follows:

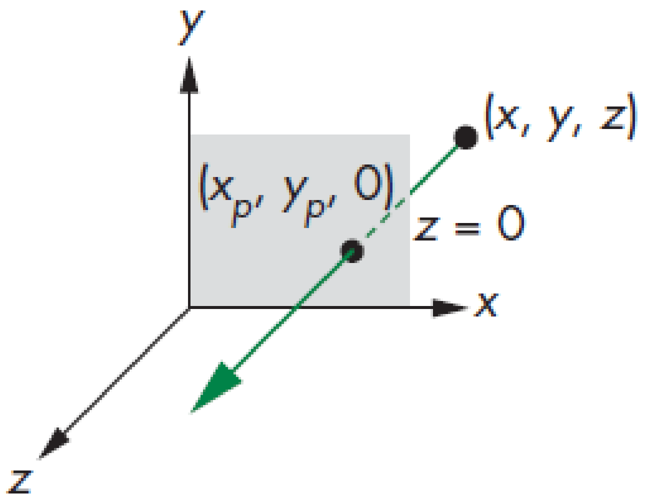


```
var initialViewMatrix = new Matrix4();
initialViewMatrix.setLookAt(0, 0, 0, 0, 0, -1, 0, 1, 0);
```

Figure 7.5 An example of setLookAt()

Parallel Projections

- Orthogonal Projections
  - The projectors are parallel.
  - The projectors are perpendicular to the view plane.
  - The equations of projection are $x_p = x, y_p = y, z_p = 0.$

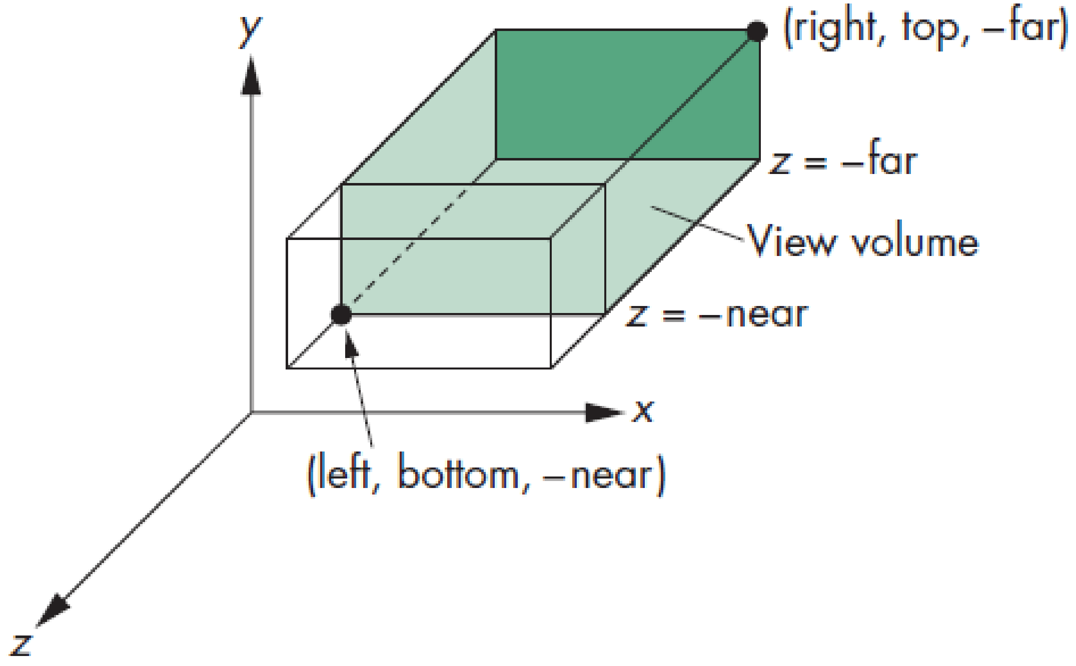


Orthogonal projection.

$$\begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Parallel Projections

- Parallel Viewing with WebGL
  - Assume the sides of the clipping volume are four planes
    - $x = right$
    - $x = left$
    - $y = top$
    - $y = bottom$



Orthographic viewing.

Parallel Projections

- Parallel Viewing with WebGL
  - The near (front) clipping plane is located at a distance *near* from the origin, and the far (back) clipping plane is at a distance *far* from the origin.
  - All these values are in camera coordinates.
  - We will derive a function

```
ortho = function(left, right, bottom, top, near, far) { ... }
```

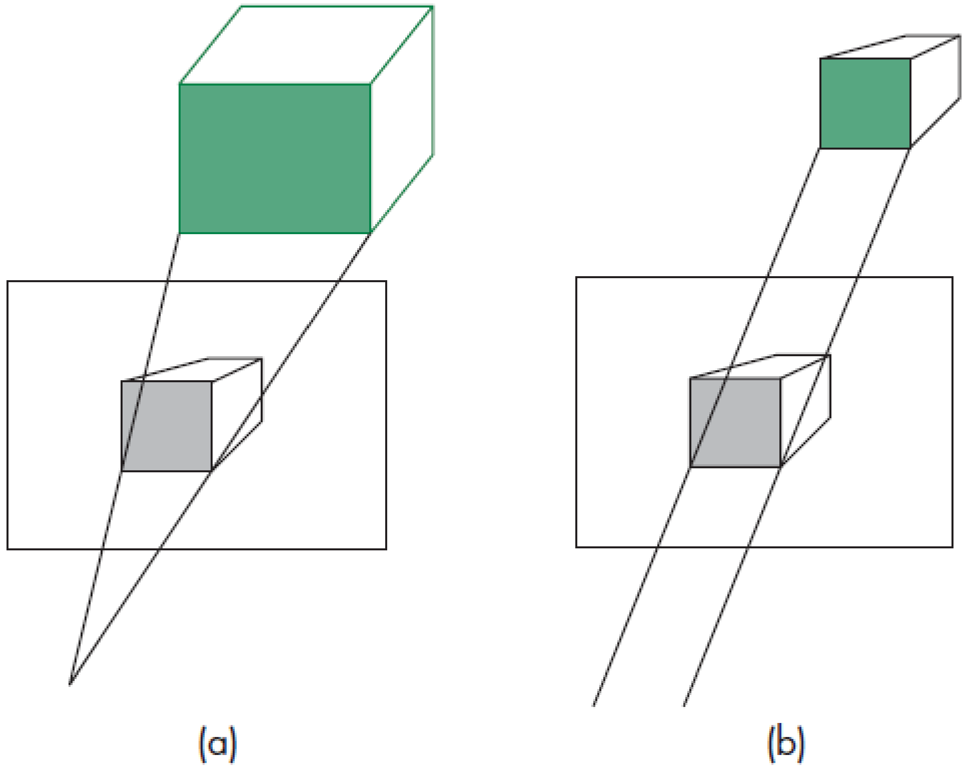
that will form the projection matrix *N*.

Parallel Projections

- Projection Normalization
  - We can convert all projections into orthogonal projections by first distorting the objects such that the orthogonal projection of the distorted objects is the same as the desired projection of the original objects.
  - We retain depth information - distance along a projector - as long as possible so that we can do hidden-surface removal later.

Parallel Projections

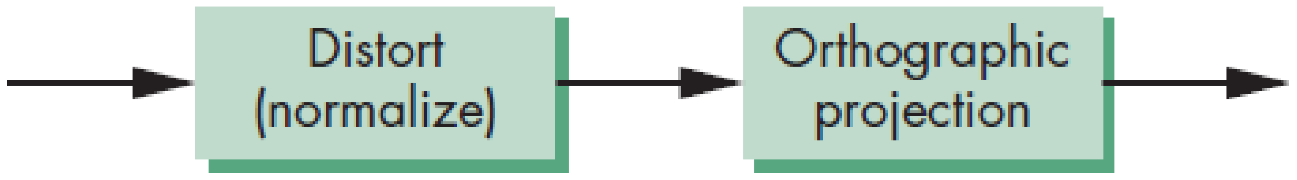
- Projection Normalization



Predistortion of objects. (a) Perspective view. (b) Orthographic projection of distorted object.

Parallel Projections

- The concatenation of the **normalization matrix**, which carries out the distortion, and the simple orthogonal projection matrix yields a homogeneous coordinate matrix that produces the desired projection.

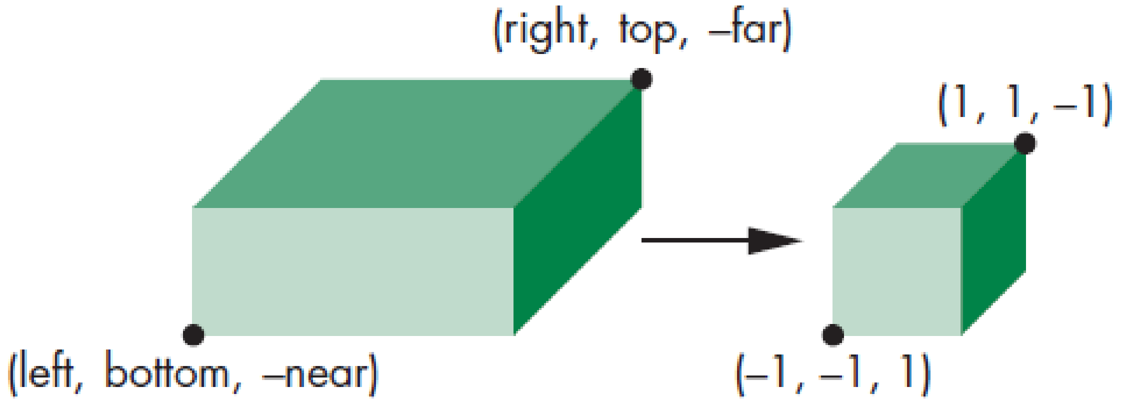


Normalization transformation.

Parallel Projections

- Orthogonal Projection Matrices
  - In WebGL, the default projection matrix is the identity matrix, or we could get using

```
var N = ortho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
```



Mapping a view volume to the canonical view volume.

Parallel Projections

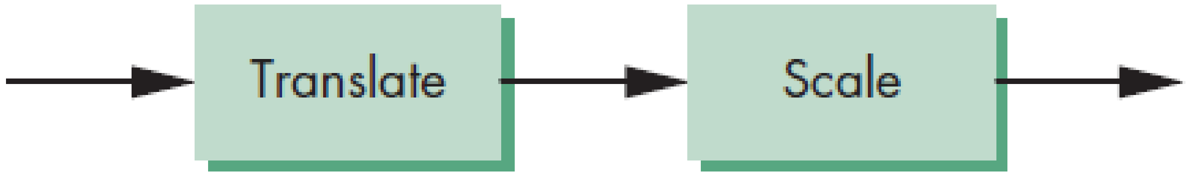
- Orthogonal Projection Matrices
  - We can get the desired projection matrix for the general orthogonal view by finding a matrix that maps the right parallelepiped specified by *ortho* to the canonical view cube.

```
var N = ortho(left, right, bottom, top, near, far);
```

  - This matrix converts the vertices that specify our objects to vertices within this canonical view volume, by scaling and translating them.

Parallel Projections

- Orthogonal Projection Matrices
  - We can use our knowledge of affine transformations to find the projection matrix.
  - There are two tasks that we need to perform.
  - First, we must move the center of the specified view volume to the center of the canonical view volume (the origin) by doing a translation.



Affine transformations for normalization.

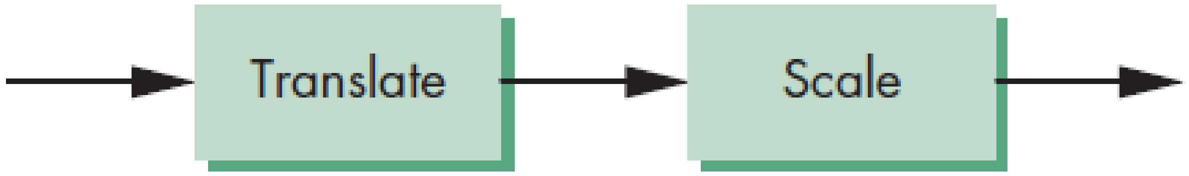
Parallel Projections

- Orthogonal Projection Matrices
  - Second, we must scale the sides of the specified view volume to each have a length of 2. Hence, the two transformations are

$$\mathbf{T} = \mathbf{T}(-(right + left)/2, -(top + bottom)/2, (far + near)/2)$$

and

$$\mathbf{S} = \mathbf{S}(2/(right - left), 2/(top - bottom), 2/(near - far))$$



Affine transformations for normalization.

Parallel Projections

- Orthogonal Projection Matrices
  - They can be concatenated together to form the projection matrix:

$$\mathbf{N} = \mathbf{ST} = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{left+right}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & -\frac{2}{far-near} & -\frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Specify the Viewing Volume

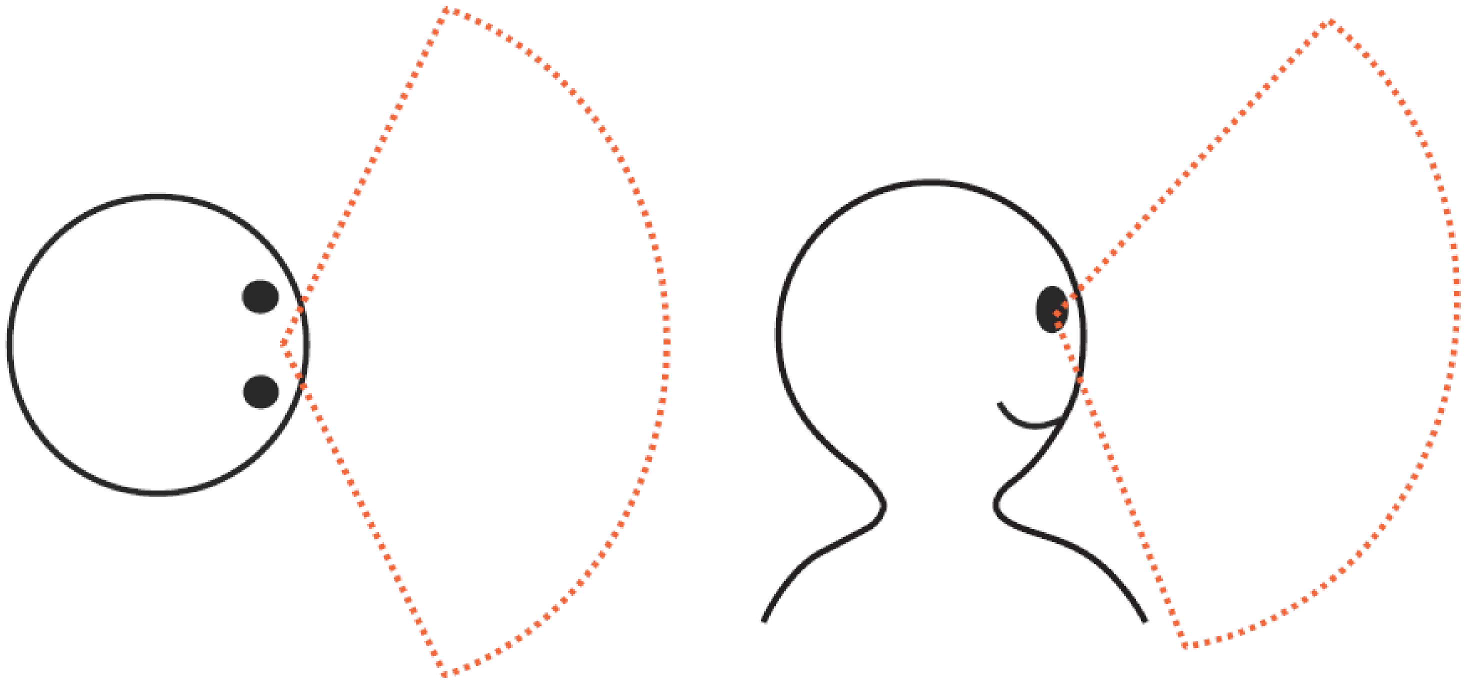


Figure 7.10 Human visual field

Specify the Viewing Volume

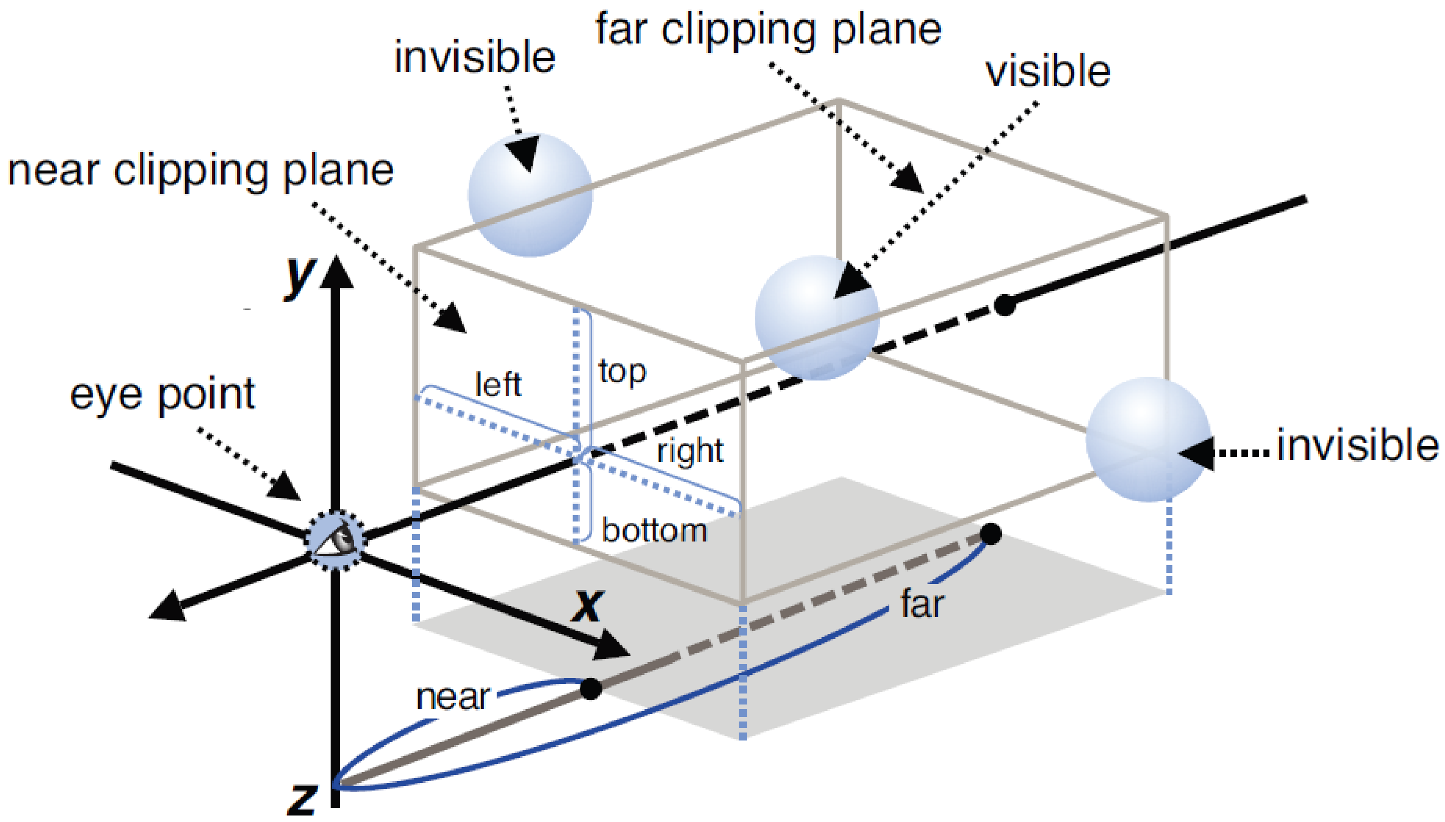
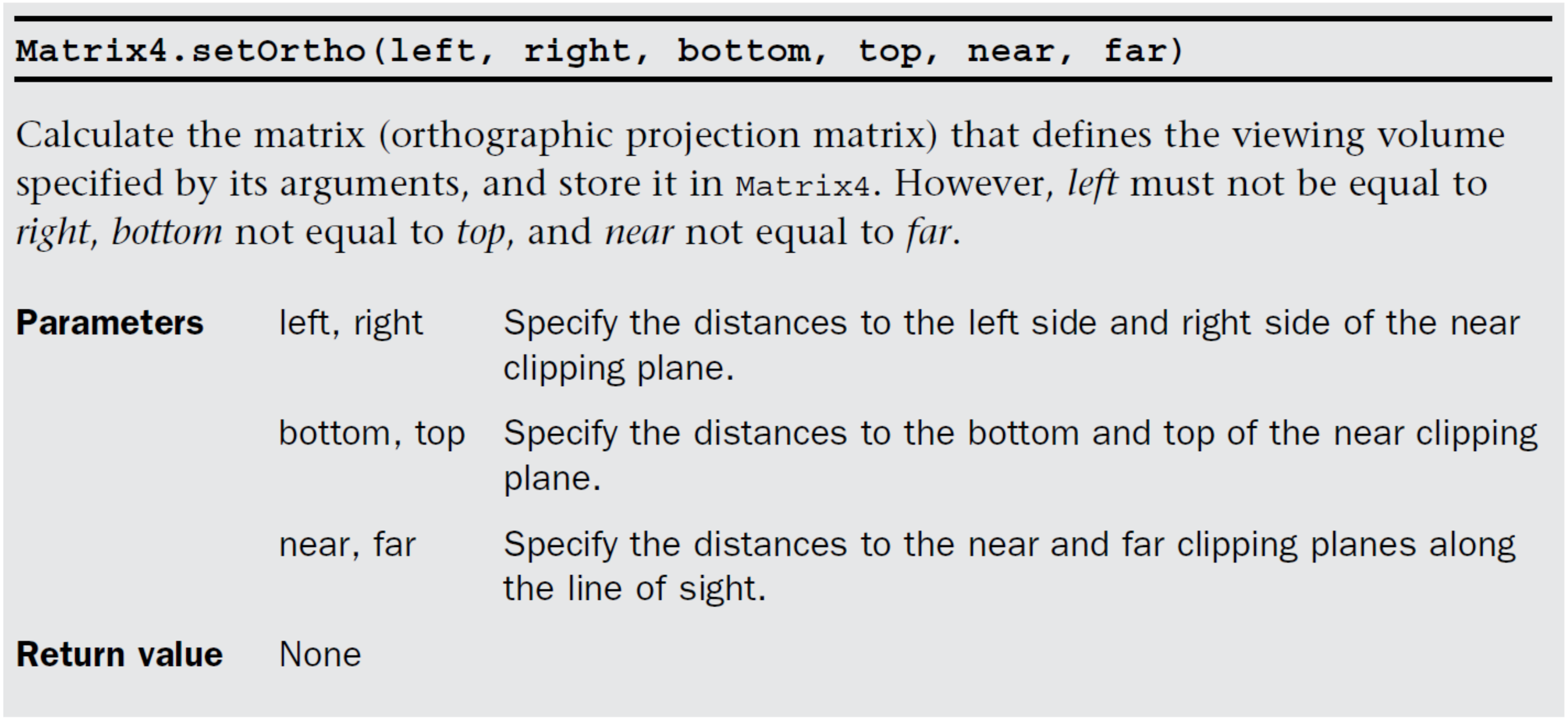


Figure 7.11 Box-shaped viewing volume

Specify the Viewing Volume

- To set the box-shaped viewing volume, you use the method *setOrtho()* supported by the *Matrix4* object defined in *cuon-matrix.js*.



```
Matrix4.setOrtho(left, right, bottom, top, near, far)
```

Calculate the matrix (orthographic projection matrix) that defines the viewing volume specified by its arguments, and store it in `Matrix4`. However, *left* must not be equal to *right*, *bottom* not equal to *top*, and *near* not equal to *far*.

Parameters	left, right	Specify the distances to the left side and right side of the near clipping plane.
	bottom, top	Specify the distances to the bottom and top of the near clipping plane.
	near, far	Specify the distances to the near and far clipping planes along the line of sight.
Return value	None	