that we used in Section 5.4.4. Hence, the transformation

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

must be inserted after the shear and before the final orthographic projection, so the final matrix is

$$\mathbf{N} = \mathbf{M}_{\text{orth}}\mathbf{ST}\ \mathbf{H}.$$

The values of *left*, *right*, *bottom*, and *top* are the vertices of the right parallelepiped view volume created by the shear. These values depend on how the sides of the original view volume are communicated through the application program; they may have to be determined from the results of the shear to the corners of the original view volume. One way to do this calculation is shown in Figure 5.29.

The specification for an oblique projection can be through the angles θ and ψ that projectors make with the projection plane. The parameters *near* and *far* are not changed by the shear. However, the x and y values where the sides of the view volume intersect the near plane are changed by the shear and become *left*, *right*, *top*, and *bottom*. If these points of intersection are $(x_{\min}, near)$, $(x_{\max}, near)$, $(y_{\min}, near)$, and $(y_{\max}, near)$, then our derivation of shear in Chapter 4 yields the relationships

$$left = x_{\min} - near * \cot\theta$$

$$right = x_{\max} - near * \cot\theta$$

$$top = y_{\max} - near * \cot\phi$$

$$bottom = y_{\min} - near * \cot\phi.$$

5.4.6 An Interactive Viewer

In this section, we extend the rotating cube program to include both the model-view matrix and an orthogonal projection matrix whose parameters can be set interactively. As in our previous examples with the cube, we have choices as to where to apply our transformations. In this example, we will send the model-view and projection matrices to the vertex shader. For an interface, we will use a set of slide bars to change parameters to alter both matrices.

The colored cube is centered at the origin in object coordinates, so wherever we place the camera, the `at` point is at the origin. Let's position the camera in polar coordinates so the `eye` point has coordinates

$$\mathbf{eye} = \begin{bmatrix} r\cos\theta \\ r\sin\theta\cos\phi \\ r\sin\theta\sin\phi \end{bmatrix},$$

where the radius r is the distance from the origin. We can let the up direction be the y direction in object coordinates. These values specify a model-view matrix through the `lookAt` function. In this example, we will send both a model-view and a projection matrix to the vertex shader with the render function

```
function render()
{
  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

  eye = vec3(radius * Math.sin(theta) * Math.cos(phi),
             radius * Math.sin(theta) * Math.sin(phi),
             radius * Math.cos(theta));
  modelViewMatrix = lookAt(eye, at, up);

  projectionMatrix = ortho(left, right, bottom, ytop, near, far);
  gl.uniformMatrix4fv(modelViewMatrixLoc, false,
                      flatten(modelViewMatrix));
  gl.uniformMatrix4fv(projectionMatrixLoc, false,
                      flatten(projectionMatrix));

  gl.drawArrays(gl.TRIANGLES, 0, numVertices);
  requestAnimFrame(render);
}
```

where `up` and `at` and other fixed values are set as part of the initialization

```
const at = vec3(0.0, 0.0, 0.0);
const up = vec3(0.0, 1.0, 0.0);
```

Note that we use the name `ytop` instead of `top` to avoid a naming conflict with the window object member names. The corresponding vertex shader is

```
attribute  vec4 vPosition;
attribute  vec4 vColor;
varying    vec4 fcolor;

uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;

void main()
{
  fcolor = vColor;
  gl_Position = projectionMatrix * modelViewMatrix * vPosition;
}
```

and the fragment shader is

```
varying vec4 color;

void main()
{
  gl_FragColor = fcolor;
}
```

The sliders are specified in the HTML file. For example, to control the near and far distances we can use

```
<div>
  depth .05
  <input id="depthSlider" type="range"
    min=".05" max="3" step="0.1" value ="2" />
  3
</div>
```

and the corresponding event handler in the JavaScript file:

```
document.getElementById("depthSlider").onchange = function() {
  far = event.srcElement.value/2;
  near = -event.srcElement.value/2;
};
```

Note that as we move the camera around, the size of the image of the cube does not change, which is a consequence of using an orthogonal projection. However, depending on the radius and the near and far distances, some or even all of the cube can be clipped out. This behavior is a consequence of the parameters in `ortho` being measured relative to the camera. Hence, if we move the camera back by increasing the radius, the back of the cube will be clipped out first. Eventually, as the radius becomes larger, the entire cube will be clipped out. In a similar manner, if we reduce the near and far distance, we can make the cube disappear from the display.

Now consider what happens as we change the parameters in `ortho`. As we increase `right` and `left`, the cube elongates in the x direction. A similar phenomenon occurs when we increase `bottom` and `ytop` in the y direction. Although this distortion of the cube's image may be annoying, it is a consequence of using an x–y rectangle in `ortho` that is not square. This rectangle is mapped to the full viewport, which has been unchanged. We can alter the program so that we increase or decrease all of `left`, `right`, `bottom`, and `top` simultaneously, or we can alter the viewport as part of any change to `ortho` (see Exercise 5.28). Two examples are on the website for interactive viewing of the colored cube; `ortho` uses buttons to alter the view, whereas `ortho2` uses slide bars.

5.5 PERSPECTIVE PROJECTIONS

We now turn to perspective projections, which are what we get with a camera whose lens has a finite focal length or, in terms of our synthetic-camera model, when the center of projection is finite.

As with parallel projections, we will separate perspective viewing into two parts: the positioning of the camera and the projection. Positioning will be done the same way, and we can use the `lookAt` function. The projection part is equivalent to selecting a lens for the camera. As we saw in Chapter 1, it is the combination of the lens and the size of the film (or the back of the camera) that determines how much of the world in front of a camera appears in the image. In computer graphics, we