Chapter 7

Viewing and Projection

"The eye is not satisfied with seeing."
Ecclesiastes I, 8, c. 200 B.C.

7.1 Overview

Viewing and projection map a portion of a three dimensional scene to a two dimensional portion of the render target. Viewing positions and orients a virtual camera within the scene. Projection maps a three dimensional portion of the scene to a two dimensional plane.

Direct3D applications use a **planar geometric projection**. A planar geometric projection of an object is formed by lines, called **projectors**, which pass through each vertex of an object and the **center of projection**. When the center of projection is finite, all projectors converge at the center of projection resulting in a perspective projection. When the center of projection is at infinity, all the projectors are parallel lines resulting in a parallel projection. The projected object is located on the **projection plane**. Each projected vertex is located at the intersection of its projector with the projection plane.

Planar geometric projections have a long history in art and engineering illustration. A perspective projection produces images that are similar to those created by the human eye and are considered more realistic in appearance. However, a perspective projection distorts the length and intersection angles of lines not parallel to the projection plane. The distortion makes objects closer to the virtual camera appear larger in the image than objects far away. This distortion is the result of nonuniform **perspective foreshortening**.

A parallel projection can preserve relative line lengths and angles of intersection. The length preserving property of parallel projections makes them useful for engineering drawings. Multiple parallel projections of an object illustrate its shape and manufacture as measurements can be taken directly from the projection.

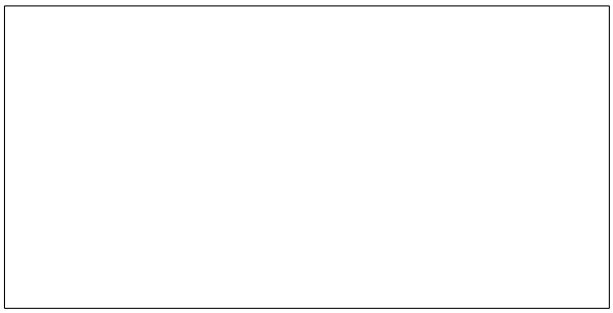

Figure 7.1: Viewing

For objects with three principal perpendicular axes, projections can be further classified. The relative orientation of the projectors, the projection plane and the principal axes of the object define the classification. Parallel projections are sudbivided into orthographic and oblique projections. Orthographic parallel projections are further classified into axonometric, isometric, dimetric, and trimetric projections. Cavalier and cabinet projections are common oblique parallel projections. Perspective projections can be subdivided into one point, two point and three point perspective projections.

Direct3D uses planar geometric projections that can be defined by a homogeneous transformation matrix. Other projections are possible, but not directly representable by Direct3D. Cartography uses a variety of projections for mapping between the surface of an ellipsoid and a plane. Alternative projections can be applied to vertex data before it is presented for rendering by the pipeline. In addition to the standard cartographic projections, points on the ellipsoid can be directly represented in world coordinates and Direct3D can perform a perspective or parallel projection of the resulting three dimensional shape.

Motion of the camera through a scene can be achieved by changing the camera parameters over time. Changes in camera position and orientation are most readily obtained by changing the view transformation matrix. The portion of the scene shown by the virtual camera is most readily obtained by changing the projection transformation matrix.

7.2 Viewing Through a Virtual Camera

Direct3D uses a model of a virtual camera to construct a rendering of a scene. The camera is conceptually positioned and oriented in world coordinates. The camera is pointed in a particular direction, the **direction of gaze**, as in figure 7.1. The position and orientation of the camera in the scene defines a new coordinate frame relative to world coordinates. This coordinate frame is called camera space, or eye space as if an eye were looking through the camera.

All the models in a scene are in world coordinates after their vertices have been mapped by the world transformation. The view transformation maps

vertices from world coordinates to **camera space**. Camera space orients the camera at the origin with the line of sight down the positive z axis, y increasing upward and x increasing to the right.

With a left-handed coordinate system, the z value will increase with distance from the camera. Increasing x values will move toward the right side of the rendered image and increasing y values will move towards the top of the rendered image. Note that y is increasing upwards in camera space, but increases downwards in screen space.

To define a view transformation, start with a camera model. As far as Direct3D is concerned the camera model is only a transformation matrix. For an application, its easier to define a camera model with some parameters and generate the matrix from the parameters.

A simple camera model is to keep the camera located at $C(x_c, y_c, z_c)$. The camera is pointed at some point $A(x_a, y_a, z_a)$ in the scene. The direction of gaze of the camera is then $\vec{g} = \langle x_a - x_c,\ y_a - y_c,\ z_a - z_c \rangle$. While keeping direction of gaze fixed, we can rotate the camera by θ around $\vec{g}$ in the plane perpendicular to $\vec{g}$. The camera is oriented with an "up" vector $\vec{u}$ in the plane perpendicular to $\vec{g}$.

With this model, the view transformation maps world coordinates so that the origin is at C, the positive z axis is aligned with the direction $\vec{g}$ and the positive y axis of the image plane is aligned with $\vec{u}$. See figure 7.1 for an illustration of the parameters. The model can be represented as a transformation matrix by a composite transformation $\mathbf{V}$.

$$\mathbf{V} = \mathbf{T}(-C)\mathbf{R}_{g\to z}\mathbf{R}_z(\theta)$$

The transformation matrix $\mathbf{R}_{g\to z}$ is a pure rotation that rotates $\vec{g}$ to the positive z axis.

The relationship between the parameters in this camera model is redundant. An application can store C and A and derive $\vec{g}$. Similarly, an application can store θ and derive $\vec{u}$.

The view transform property of the device is manipulated through the `D3DTS_VIEW` argument to `GetTransform` and `SetTransform`. The view transform defaults to an identity matrix, which makes camera space coordinates identical to the world space coordinates. Some applications prefer to include the view transformation in the world matrix, using an identity view matrix. D3DX includes the functions for generating view transformation matrices, see chapter 16.

The camera can be dynamically moved through the scene by changing the transformation. Repeatedly accumulating matrices through post-multiplication for incremental camera movement suffers from numerical roundoff error. By keeping a camera model from which we generate the matrix, we avoid this problem with matrices, but it can still crop up with the floating-point variables containing our camera parameters. If the matrix is known not to contain any skew transformations, it can be renormalized. The rotation portion of a homogeneous transformation matrix is modified so that the axes of rotation are all mutually perpendicular to correct any introduced skew.

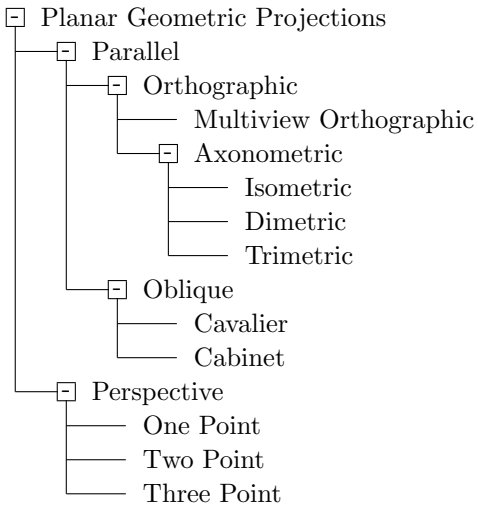

Figure 7.2: Taxonomy of planar geometric projections. A multiview orthographic projection is not a single projection, but a group of orthographic views of the same scene. Typically each view is chosen parallel to one of the principal axes with names such as top, bottom, front, back, left side, and right side.

Quaternions are a convenient representation for orienting transformations. Direct3D core interfaces don't use quaternions directly, but D3DX provides a quaternion data type that can be used to manipulate and define orienting transformations, see chapter 16.

7.3 Planar Geometric Projections

A planar geometric projection is characterized by its **projection plane** and its **center of projection**. A line through a point in the scene and the center of projection is a **projector**. Planar geometric projections for artistic and functional uses have a long history in the visual arts. The taxonomy of planar geometric projections is shown in figure 7.2.

7.3.1 Parallel Projections

A **parallel projection** results when the center of projection is located at infinity. All the projectors will effectively be parallel to each other. Parallel projections preserve the length of lines within a model and provide uniform foreshortening. This can result in an image that doesn't look realistic, in contrast to the image taken by a camera or seen with the human eye. However, the resulting projection does allow for measurements to be made directly from the rendered image. For this reason, parallel projections are often used in engineering drawings.

Parallel projections can be subdivided into **orthographic** and **oblique** parallel projections. In an orthographic projection, the projectors are all perpendicular to the projection plane, while in an oblique parallel projection the projectors intersect the projection plane at an angle.

Orthographic Parallel Projections

Orthographic projections are probably the most common form of parallel projection. An orthographic parallel projection is most often used to illustrate an object in a static engineering drawing and the object usually has a natural coordinate system with three principal perpendicular axes. For example, most computer cases are shaped like a box, with three natural coordinate axes defining the width, height and depth of the case.

Because orthographic projections do not provide realistic views of an object, multiple projections of the same object are often used to provide a clear illustration of a complex object. "Front", "side" and "plan" orthographic views of an object are commonly used to illustrate an object from three sides in order to clearly show its three-dimensional shape.

An orthographic projection which projects an object into a single projection plane is called an **axonometric projection**. The choice of the projection plane is made to illustrate the general shape of the object without ambiguity.

Axonometric projections are further distinguished by the relationship of the projection plane to the three principal axes of the depicted object. When all the principal axes of the object intersect the projection plane with the same angle, an **isometric** projection results. If two of the principal axes intersect the projection plane at the same angle, while the remaining principal axis intersects at a different angle, then a **dimetric** projection results. When all three angles of intersection are different, a **trimetric** projection results.

Oblique Parallel Projections

While orthographic projections preserve an object's shape, they often make it difficult to visualize the three dimensional shape of an object. Multiview orthographic projections can make up for this by showing an object from two or more views, but it may still be hard to visualize the object's shape from the multiple views. An oblique parallel projection provides an illustration of the general three-dimensional shape of an object while providing an exact representation of one of the object's faces.

An oblique parallel projection is characterized by the orientation of the projection plane relative to the object, the angle of the projectors relative to the projection plane and the orientation of the projectors about the projection plane normal. Usually the projection plane is chosen to be parallel to the most important face of the object and the angle of the projectors is chosen to give a realistic appearance to the third dimension of the object.

A **cavalier** projection results when the angle of the projectors to the projection plane is 45 degrees. A **cabinet** projection results when the angle of the

projectors is $\cot^{-1}(\frac{1}{2})$, or approximately 64 degrees. A cavalier projection does not foreshorten lines perpendicular to the primary face of the object, while a cabinet projection provides a sense of three dimensional appearance of an object with uniform foreshortening.

7.3.2 Perspective Projections

A **perspective projection** results when the center of projection is finite. The projectors then pass through points in the object and converge on the center of projection. This results in several visual effects in the projected image:

1. Parallel lines not in a plane parallel to the projection plane will appear to converge at a **vanishing point**.

2. Objects appear to change their size depending on their distance from the center of projection.

3. Objects are foreshortened nonuniformly.

As with parallel projections, perspective projections can be further subdivided based on the relationship of the projection plane to the principal axes of a depicted object. A **one-point perspective** projection results when the projection plane intersects a single principal axis of the object. In this case, the projection plane is parallel to one of the principal faces of the depicted object. Similarly, a **two-point perspective** projection results when the projection plane intersects two of the principal axes of the object and a **three-point perspective** projection results when the projection plane intersects all three of the principal axes.

7.4 Projection Transformation

In computer graphics we are often interested in dynamic views where the objects within the scene and the camera may both move freely in response to user interaction. In these situations, the distinctions between these different kinds of parallel or perspective projections are not that important since objects within the scene are likely to be at arbitrary orientations relative to the projection plane. The most common choice made is one between a parallel or perspective projection.

For instance, in a first-person-shooter style game, the player's view is most often shown with a perspective projection. A parallel projection may still be used for map displays where it is important to preserve the exact shape of the rendered primitives.

The projection transform property of the device is manipulated through the `D3DTS_PROJECTION` argument to `GetTransform` and `SetTransform`. The projection transform defaults to an identity matrix, resulting in a parallel projection.

7.4.1 Projection Matrices

So far, the matrices we have examined have not used the rightmost column of the matrix. These elements are used for applying a perspective projection. To see how this results, let's take a look at the equations for transforming a point $P = \langle x, y, z, w \rangle$ by a matrix $\mathbf{M}$ resulting in a transformed point $P' = \langle x', y', z', w' \rangle$, where $\mathbf{M}$ is an identity matrix with unknown elements in the rightmost column:

$$\begin{aligned}
P' &= P\mathbf{M} \\
&= \langle x, y, z, w \rangle \begin{bmatrix} 1 & 0 & 0 & m_{14} \\ 0 & 1 & 0 & m_{24} \\ 0 & 0 & 1 & m_{34} \\ 0 & 0 & 0 & m_{44} \end{bmatrix} \\
x' &= x \\
y' &= y \\
z' &= z \\
w' &= xm_{14} + ym_{24} + zm_{34} + wm_{44}
\end{aligned}$$

When P' is converted to a cartesian coordinate, its x, y and z coordinates will be divided by its w coordinate. Usually, w' would have the value of one and this division wouldn't change the cartesian position of the point, but when the rightmost column of the matrix contains non-zero elements, w will have a value other than one and the cartesian coordinate will change accordingly.

For instance, suppose we have a perspective transformation matrix:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This will result in a transformed w coordinate equal to the z coordinate of the original point. When the point is converted from a homogeneous coordinate system to a cartesian coordinate system, the x, y, and z values will be divided by $z + w$ from the original point. This is what causes the non-uniform perspective foreshortening in a perspective projection.

The projection transformation matrix should map the desired view frustum into the canonical view frustum:

$$\begin{aligned}
-w &\leq x \leq w \\
-w &\leq y \leq w \\
0 &\leq z \leq w
\end{aligned}$$

If your application uses a right-handed coordinate system for world and camera space, you should include the conversion to a left-handed coordinate system in the projection matrix.

D3DX provides functions for computing parallel and perspective projection matrices, see chapter 15. You can also compute your own projection matrices directly from the relationship of your desired view frustum to the canonical view frustum. Direct3D requires that the m_{34} element be normalized to 1 for perspective transformation matrices. If your computations result in a non-zero m_{34} element, you can multiply all the elements of the matrix the reciprocal of m_{34} to meet this requirement.

7.5 Further Reading

Projections have a long history in the visual arts and scientific or engineering illustration. In computer graphics, the most common choice is between a simple orthographic or perspective projection, but occasionally the need arises for something more elaborate.

Planar Geometric Projections and Viewing Transformations, I. Carlbom and J. Paciorek, *Computing Surveys*, 1(4), 1978, pp. 465-502.
A detailed description of the history of planar geometric projections and the derivation of their projection matrices.

Map Projections–A Working Manual, John P. Snyder, U.S. Geological Survey Professional Paper 1395, United States Government Printing Office, Washington, D.C., 1987.
An encyclopedic collection of projection formulas and ancilliary data for use in cartography.

7.6 `rt_Views` Sample Application

The following sample application shows an example of multiple views within a render target, with each view having a different viewport, view matrix and projection matrix. `rt_Views` was generated with the SDK framework using the AppWizard included with the SDK, see appendix A.

The entire source code is included in the samples. Listed here is `rt_Views.cpp`, containing the "interesting" code of the sample. The sample computes perspective and orthographic view matrices that enclose a portion of model space. The aspect ratio of the view frustum is chosen to enclose the mesh snugly in lines 30–40. The differing view and projection matrices for the four views is computed in lines 334–353 using some helper functions for the perspective and orthographic view matrices. The viewport sizes are initialized in lines 147–159 of the `RestoreDeviceObjects` method. Each viewport is set for the appropriate view in lines 322–328 of the `set_view` method.

The sample uses small helper classes that encapsulate reusable Direct3D coding idioms. Their meaning should be straightforward and all such helper classes are placed in the `rt` namespace to highlight their use. The `mat` classes construct transformation matrices so that matrix expressions can be more readily composed. The sample source code contains their definitions.

Listing 7.1: rt_Views.cpp: Drawing multiple views in a single render target.

```
1   //--------------------------------------------------------
2   // rt_Views.cpp
3   //
4   // Demonstrates multiple views of a scene using viewports.
5   //
6   #include <algorithm>
7
8   #define STRICT
9   #include <windows.h>
10
11  #include <atlbase.h>
12
13  #include <d3dx9.h>
14
15  #include "DXUtil.h"
16  #include "D3DEnumeration.h"
17  #include "D3DSettings.h"
18  #include "D3DApp.h"
19  #include "D3DFont.h"
20  #include "D3DUtil.h"
21
22  #include "resource.h"
23  #include "rt_Views.h"
24
25  #include "rt/hr.h"
26  #include "rt/mat.h"
27  #include "rt/misc.h"
28  #include "rt/states.h"
29
30  //--------------------------------------------------------
31  // compute_size
32  //
33  // Helper function to adjust the xy dimensions of the view
34  // volume for the back buffer aspect ratio.
35  //
36  inline void
37  compute_size(float model_width, float model_height,
38               float back_aspect,
39               float &proj_width, float &proj_height)
40  {
41      const float model_aspect = model_width/model_height;
42      if (back_aspect > model_aspect)
43      {
44          proj_height = model_height;
```

```
45          proj_width = proj_height*back_aspect;
46      }
47      else
48      {
49          proj_width = model_width;
50          proj_height = proj_width/back_aspect;
51      }
52  }
53
54  ////////////////////////////////////////////////////////
55  // mat_persp_ratios
56  //
57  // Compute a perspective projection matrix that encloses
58  // the model in a back buffer of the given aspect ratio.
59  //
60  class mat_persp_ratios : public D3DXMATRIX
61  {
62  public:
63      mat_persp_ratios(float model_x, float model_y,
64                       float back_aspect)
65      {
66          float w, h;
67          compute_size(model_x, model_y, back_aspect, w, h);
68          ::D3DXMatrixPerspectiveLH(this, w/4, h/4,
69              1.0f, 10.0f);
70      }
71  };
72
73  ////////////////////////////////////////////////////////
74  // mat_ortho_ratios
75  //
76  // Compute an orthographic projection matrix that encloses
77  // the model in a back buffer of the given aspect ratio.
78  //
79  class mat_ortho_ratios : public D3DXMATRIX
80  {
81  public:
82      mat_ortho_ratios(float model_x, float model_y,
83                       float back_aspect)
84      {
85          float w, h;
86          compute_size(model_x, model_y, back_aspect, w, h);
87          ::D3DXMatrixOrthoLH(this, w, h, 1.0f, 10.0f);
88      }
89  };
90
```

```
91  //////////////////////////////////////////////////////////
92  // CMyD3DApplication::RestoreDeviceObjects()
93  //
94  // Calculate the sizes of the four views based on the size
95  // of the back buffer.  Set a simple light and material
96  // properties for the scene.
97  //
98  HRESULT CMyD3DApplication::RestoreDeviceObjects()
99  {
100     const DWORD w = m_d3dsdBackBuffer.Width;
101     const DWORD h = m_d3dsdBackBuffer.Height;
102     const s_view views[NUM_VIEWS] =
103     {
104         { 0, 0, w, h, D3DCOLOR_XRGB(68, 68, 68) },
105         { 0, 0, w/2, h/2, D3DCOLOR_XRGB(0, 0, 0) },
106         { w/2, 0, w-(w/2), h/2, D3DCOLOR_XRGB(0, 68, 0) },
107         { 0, h/2, w/2, h-(h/2), D3DCOLOR_XRGB(0, 0, 68) },
108         { w/2, h/2, w-(w/2), h-(h/2),
109           D3DCOLOR_XRGB(68, 0, 0) }
110     };
111     ATLASSERT(sizeof(views) == sizeof(m_views));
112     std::copy(&views[0], &views[NUM_VIEWS], &m_views[0]);
113
114     // Setup a material
115     D3DMATERIAL9 mtrl;
116     ::D3DUtil_InitMaterial(mtrl, 1.0f, 1.0f, 0.0f);
117     THR(m_pd3dDevice->SetMaterial(&mtrl));
118
119     // Set miscellaneous render states
120     const rt::s_rs states[] =
121     {
122         D3DRS_DITHERENABLE, false,
123         D3DRS_SPECULARENABLE, false,
124         D3DRS_ZENABLE, true,
125         D3DRS_AMBIENT, D3DCOLOR_XRGB(15, 15, 15),
126         D3DRS_LIGHTING, true
127     };
128     rt::set_states(m_pd3dDevice, states, NUM_OF(states));
129
130     // Set up two directional lights in opposite directions
131     D3DLIGHT9 light;
132     ::D3DUtil_InitLight(light, D3DLIGHT_DIRECTIONAL,
133         -1.0f, -1.0f, 2.0f);
134     m_pd3dDevice->SetLight(0, &light);
135     m_pd3dDevice->LightEnable(0, true);
136     ::D3DUtil_InitLight(light, D3DLIGHT_DIRECTIONAL,
```

```
137             1.0f, 1.0f, -2.0f);
138         THR(m_pd3dDevice->SetLight(1, &light));
139         THR(m_pd3dDevice->LightEnable(1, true));
140
141         // Restore the font
142         m_font.RestoreDeviceObjects();
143
144         return S_OK;
145     }
146
147     //////////////////////////////////////////////////////////
148     // CMyD3DApplication::set_view_identity
149     //
150     // set the view to cover the entire render target
151     //
152     void
153     CMyD3DApplication::set_view_identity()
154     {
155         D3DXMATRIX one;
156         ::D3DXMatrixIdentity(&one);
157         THR(m_pd3dDevice->SetTransform(D3DTS_WORLD, &one));
158         THR(m_pd3dDevice->SetTransform(D3DTS_VIEW, &one));
159         THR(m_pd3dDevice->SetTransform(D3DTS_PROJECTION, &one));
160     }
161
162     //////////////////////////////////////////////////////////
163     // CMyD3DApplication::set_view_persp
164     //
165     // Sets up a perspective view using the given camera
166     // parameters  and a view frustum sized to the model.
167     //
168     void
169     CMyD3DApplication::set_view_persp(const D3DXVECTOR3 &eye,
170                                       const D3DXVECTOR3 &at,
171                                       const D3DXVECTOR3 &view_up)
172     {
173         // world matrix
174         const D3DXVECTOR3 center =
175             (m_mesh_bounds.minima + m_mesh_bounds.maxima)*0.5f;
176         const D3DXMATRIX world =
177             rt::mat_trans(-center.x, -center.y, -center.z)*
178             rt::mat_rot_x(m_rot_x)*rt::mat_rot_y(m_rot_y);
179         THR(m_pd3dDevice->SetTransform(D3DTS_WORLD, &world));
180
181         // view matrix
182         const rt::mat_look_at view(eye, at, view_up);
```

```
183       THR(m_pd3dDevice->SetTransform(D3DTS_VIEW, &view));
184
185       // Set the projection matrix
186       const D3DXVECTOR3 size =
187           m_mesh_bounds.maxima - m_mesh_bounds.minima;
188       const float back_aspect = float(m_d3dsdBackBuffer.Width)
189           / m_d3dsdBackBuffer.Height;
190       const mat_persp_ratios proj(size.x, size.y, back_aspect);
191       THR(m_pd3dDevice->SetTransform(D3DTS_PROJECTION, &proj));
192   }
193
194   ////////////////////////////////////////////////////////
195   // CMyD3DApplication::set_view_ortho
196   //
197   // Sets up an orthographic view using the given camera
198   // parameters and a view volume sized to the model.
199   //
200   void
201   CMyD3DApplication::set_view_ortho(const D3DXVECTOR3 &eye,
202                                     const D3DXVECTOR3 &at,
203                                     const D3DXVECTOR3 &view_up)
204   {
205       // world matrix
206       const D3DXVECTOR3 center =
207           (m_mesh_bounds.minima + m_mesh_bounds.maxima)*0.5f;
208       const D3DXMATRIX world =
209           rt::mat_trans(-center.x, -center.y, -center.z)*
210           rt::mat_rot_x(m_rot_x)*rt::mat_rot_y(m_rot_y);
211       THR(m_pd3dDevice->SetTransform(D3DTS_WORLD, &world));
212
213       // view matrix
214       const rt::mat_look_at view(eye, at, view_up);
215       THR(m_pd3dDevice->SetTransform(D3DTS_VIEW, &view));
216
217       // Set the projection matrix
218       const D3DXVECTOR3 size =
219           m_mesh_bounds.maxima - m_mesh_bounds.minima;
220       const float back_aspect = float(m_d3dsdBackBuffer.Width)
221           / m_d3dsdBackBuffer.Height;
222       const mat_ortho_ratios proj(size.x, size.y, back_aspect);
223       THR(m_pd3dDevice->SetTransform(D3DTS_PROJECTION, &proj));
224   }
225
226   ////////////////////////////////////////////////////////
227   // CMyD3DApplication::set_view
228   //
```

```
229 // Set the appropriate view state for each of the 5
230 // views (4 quarters, plus a view that covers the
231 // entire render target).  For simplicity, the matrices
232 // are recomputed every time.
233 //
234 void
235 CMyD3DApplication::set_view(e_view view)
236 {
237     const D3DVIEWPORT9 vp =
238     {
239         m_views[view].x, m_views[view].y,
240         m_views[view].width, m_views[view].height,
241         0.0f, 1.0f
242     };
243     THR(m_pd3dDevice->SetViewport(&vp));
244     switch (view)
245     {
246     case VIEW_ALL:
247         set_view_identity();
248         break;
249     case VIEW_UPPER_LEFT:
250         set_view_persp(D3DXVECTOR3(0, 0, -5),
251                        D3DXVECTOR3(0, 0, 0),
252                        D3DXVECTOR3(0, 1, 0));
253         break;
254     case VIEW_UPPER_RIGHT:
255         set_view_ortho(D3DXVECTOR3(0, 5, 0),
256                        D3DXVECTOR3(0, 0, 0),
257                        D3DXVECTOR3(0, 0, 1));
258         break;
259     case VIEW_LOWER_LEFT:
260         set_view_ortho(D3DXVECTOR3(0, 0, -5),
261                        D3DXVECTOR3(0, 0, 0),
262                        D3DXVECTOR3(0, 1, 0));
263         break;
264     case VIEW_LOWER_RIGHT:
265         set_view_ortho(D3DXVECTOR3(-5, 0, 0),
266                        D3DXVECTOR3(0, 0, 0),
267                        D3DXVECTOR3(0, 1, 0));
268         break;
269
270     default:
271         ATLASSERT(false);
272     }
273 }
274
```

```
275 //////////////////////////////////////////////////////////
276 // CMyD3DApplication::draw_scene
277 //
278 // Set the appropriate view state, clear the viewport
279 // and draw the text mesh.
280 //
281 void
282 CMyD3DApplication::draw_scene(e_view view)
283 {
284     set_view(view);
285     THR(m_pd3dDevice->Clear(0L, NULL,
286         D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
287         m_views[view].bg, 1.0f, 0L));
288
289     THR(m_mesh->DrawSubset(0));
290 }
291
292 //////////////////////////////////////////////////////////
293 // Render()
294 //
295 // Render each of the views and possibly draw statistics
296 // on top of all four views.
297 //
298 HRESULT CMyD3DApplication::Render()
299 {
300     THR(m_pd3dDevice->BeginScene());
301
302     // draw four views of the model
303     draw_scene(VIEW_UPPER_LEFT);
304     draw_scene(VIEW_UPPER_RIGHT);
305     draw_scene(VIEW_LOWER_LEFT);
306     draw_scene(VIEW_LOWER_RIGHT);
307
308     // Render stats and help text
309     if (m_draw_stats)
310     {
311         set_view(VIEW_ALL);
312         RenderText();
313     }
314
315     THR(m_pd3dDevice->EndScene());
316
317     return S_OK;
318 }
```