Chapter 5

Matrices, Transforms, and Rotations

Most beginners believe that matrices and 3D math are the most difficult part of learning graphics programming. This might have been true a couple of years ago, but it's not true anymore. Direct3D has advanced during that time and taken away a lot of the complexity, leaving programmers to focus more on how they want their games to work.

This chapter introduces you to matrices and shows you how easily they can be made to work for you instead of against you.

Here's what you'll learn in this chapter:

- What a 3D model is and how it's created
- How to optimize rendering by using index buffers
- What the geometry pipeline is and what its stages are
- What matrices are and how they affect your 3D world
- How D3DX can make your job easier
- What it takes to manipulate your objects within a scene
- How to create a virtual camera

Creating a 3D Model

Now that you've been introduced to drawing triangles, it's time to expand on that knowledge and create full 3D models. Almost everything in games is represented with 3D objects, from the character you play to the environment the character interacts with. A 3D

05 DX9_GP CH05 3/12/04 4:15 PM Page 88

object's complexity can range from a single polygon to thousands of polygons, depending on what the model represents. Full cities complete with cars, buildings, and people can be represented this way.

Although 3D objects might seem intimidating, it helps to think of them as just a series of connected triangles. By breaking down a model into its most primitive type, it becomes a little easier to grasp.

I'm going to take you through the steps you need to follow to create and render a cube. A cube isn't the most complicated object, but it does give you the basics you need to handle any 3D model.

Defining the Vertex Buffer

In Chapter 4, "3D Primer," you were introduced to vertex buffers as a clean and handy place for storing vertex information. As the complexity of the objects you're using grows, the convenience of vertex buffers will become more apparent. The vertex buffer is a great place to store object vertices, allowing you easy access and simple methods for rendering those vertices.

Your previous use of a vertex buffer needed only three vertices to create a triangle. Now that you'll be creating a more complicated object, you'll need to store more vertices.

When you're defining the vertices for a static object, consider storing them in an array. The array has a type of `CUSTOMVERTEX`, which, as you'll recall from Chapter 4, allows you to define the layout of your vertex data. Each element in the array holds all the information that Direct3D needs to describe a single vertex. Following is the code you need to define the vertices for a cube.

```
// a structure for your custom vertex type
struct CUSTOMVERTEX
{
    FLOAT x, y, z;    // the untransformed, 3D position for the vertex
    DWORD color;    // the color of the vertex
};

CUSTOMVERTEX g_Vertices[] =
{
    // 1
    { -64.0f,  64.0f, -64.0f, D3DCOLOR_ARGB(0,0,0,255)},
    {  64.0f,  64.0f, -64.0f, D3DCOLOR_ARGB(0,0,0,255)},
    { -64.0f, -64.0f, -64.0f, D3DCOLOR_ARGB(0,0,0,255)},
    {  64.0f, -64.0f, -64.0f, D3DCOLOR_ARGB(0,0,0,255)},

    // 2
```

```
    { -64.0f,  64.0f, 64.0f, D3DCOLOR_ARGB(0,0,0,255)},
    { -64.0f, -64.0f, 64.0f, D3DCOLOR_ARGB(0,0,0,255)},
    {  64.0f,  64.0f, 64.0f, D3DCOLOR_ARGB(0,0,0,255)},
    {  64.0f, -64.0f, 64.0f, D3DCOLOR_ARGB(0,0,0,255)},

    // 3
    { -64.0f, 64.0f,  64.0f, D3DCOLOR_ARGB(0,0,0,255)},
    {  64.0f, 64.0f,  64.0f, D3DCOLOR_ARGB(0,0,0,255)},
    { -64.0f, 64.0f, -64.0f, D3DCOLOR_ARGB(0,0,0,255)},
    {  64.0f, 64.0f, -64.0f, D3DCOLOR_ARGB(0,0,0,255)},

    // 4
    { -64.0f, -64.0f,  64.0f, D3DCOLOR_ARGB(0,0,0,255)},
    { -64.0f, -64.0f, -64.0f, D3DCOLOR_ARGB(0,0,0,255)},
    {  64.0f, -64.0f,  64.0f, D3DCOLOR_ARGB(0,0,0,255)},
    {  64.0f, -64.0f, -64.0f, D3DCOLOR_ARGB(0,0,0,255)},

    // 5
    { 64.0f,  64.0f, -64.0f, D3DCOLOR_ARGB(0,0,0,255)},
    { 64.0f,  64.0f,  64.0f, D3DCOLOR_ARGB(0,0,0,255)},
    { 64.0f, -64.0f, -64.0f, D3DCOLOR_ARGB(0,0,0,255)},
    { 64.0f, -64.0f,  64.0f, D3DCOLOR_ARGB(0,0,0,255)},

    // 6
    {-64.0f,  64.0f, -64.0f, D3DCOLOR_ARGB(0,0,0,255)},
    {-64.0f, -64.0f, -64.0f, D3DCOLOR_ARGB(0,0,0,255)},
    {-64.0f,  64.0f,  64.0f, D3DCOLOR_ARGB(0,0,0,255)},
    {-64.0f, -64.0f,  64.0f, D3DCOLOR_ARGB(0,0,0,255)},
};
```

The first thing the code does is to declare the layout of the `CUSTOMVERTEX` structure. This structure includes two sections: the position using the `X`, `Y`, and `Z` variables, and the color. After the structure is defined, the `g_Vertices` array is created and filled with the vertex data necessary to create a cube. The vertex data is split up into six sections, each one representing a side of the cube. Previously, you always used a value of `1.0f` for the `Z` value in vertex declarations, which made your objects appear flat. Because the cube you are creating is a fully 3D model, the `Z` value is being used to define the distances of the vertices within the world.

The next step in the process is creating and filling the vertex buffer with the vertex data for the cube. The code that follows shows what is needed to do this.

```
// Create the vertex buffer
HRESULT hr;
```

05 DX9_GP CH05 3/12/04 4:15 PM Page 90


```
LPDIRECT3DVERTEXBUFFER9 vertexBuffer;

// Create the vertex buffer that will store the cube's vertices
hr = pd3dDevice->CreateVertexBuffer(sizeof(g_Vertices) * sizeof(CUSTOMVERTEX),
                                    0,
                                    D3DFVF_CUSTOMVERTEX,
                                    D3DPOOL_DEFAULT,
                                    &vertexBuffer,
                                    NULL );

// Check the return code of CreateVertexBuffer call to make sure it succeeded
if FAILED (hr)
    return false;

// Prepare to copy the vertices into the vertex buffer
VOID* pVertices;

// Lock the vertex buffer
hr = vertexBuffer->Lock(0,
                        sizeof(g_Vertices),
                        (void**) &pVertices,
                        0);

// Check to make sure the vertex buffer can be locked
if FAILED (hr)
    return false;

// Copy the vertices into the buffer
memcpy ( pVertices, g_Vertices, sizeof(g_Vertices) );

// Unlock the vertex buffer
vertexBuffer->Unlock();
```

Using the call to `CreateVertexBuffer` creates the vertex buffer; its size and type are defined as well. Instead of specifically stating the size of the vertex buffer to create, I've used the `sizeof` function to calculate this at compile time. Multiply the size of the `g_Vertices` array by the size of the `CUSTOMVERTEX` structure to get the exact size that the vertex buffer needs to be to hold all the vertices.

The resulting buffer is then locked, and the vertices from the `g_Vertices` array are copied into it using the `memcpy` function.

Now that you have a filled vertex buffer, you are ready to draw your 3D object.

05 DX9_GP CH05 3/12/04 4:15 PM Page 91

Rendering the Cube

Rendering the cube is just like drawing any other object from a vertex buffer, regardless of its complexity. The major difference separating a cube, a triangle, and a car is the number of vertices involved. After the object is stored in the vertex buffer, it's easy to render it.

The `Render` function shown here details the code needed to render the cube you defined in the `g_Vertices` array.

```
/******************************************************************************
* Render
******************************************************************************/
void Render(void)
{
    // Clear the back buffer to a white color
    pd3dDevice->Clear( 0,
                        NULL,
                        D3DCLEAR_TARGET,
                        D3DCOLOR_XRGB(255,255,255),
                        1.0f,
                        0 );

    pd3dDevice->BeginScene();

    // Set the vertex stream for the model
    pd3dDevice->SetStreamSource( 0, vertexBuffer, 0, sizeof(CUSTOMVERTEX) );

    // Set the vertex format
    pd3dDevice->SetFVF( D3DFVF_CUSTOMVERTEX );

    // Call DrawPrimitive to draw the cube
    pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP,  0, 2 );
    pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP,  4, 2 );
    pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP,  8, 2 );
    pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 12, 2 );
    pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 16, 2 );
    pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 20, 2 );

    pd3dDevice->EndScene();

    // Present the back buffer contents to the display
    pd3dDevice->Present( NULL, NULL, NULL, NULL );
}
```

The cube is rendered first by setting the vertex stream source and the vertex format. The biggest difference between drawing one triangle and drawing the multiple triangles required to render a 3D cube is the additional calls to DrawPrimitive. Each of the six DrawPrimitive calls renders a single side of the cube using the triangle strip primitive.

Figure 5.1 shows what the resulting cube looks like. The cube is rendered using wireframe so that you can see the triangles that go into its creation.

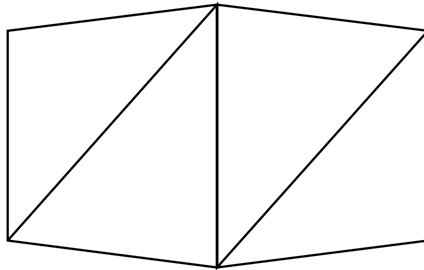

Figure 5.1 The full 3D cube.

Index Buffers

Index buffers are areas of memory that store index data. Each index in the buffer refers to a particular vertex with a vertex buffer. The indices reduce the amount of data that must be sent to the graphic card by sending only a single value for each vertex instead of the full X, Y, and Z data. The vertex data lives in the vertex buffer, and the values within the index buffer reference the vertex buffer.

You create index buffers, which are based on the IDirect3DIndexBuffer9 interface, by using the CreateIndexBuffer function, which is defined next.

```
HRESULT CreateIndexBuffer(
    UINT Length,
    DWORD Usage,
    D3DFORMAT Format,
    D3DPOOL Pool,
    IDirect3DIndexBuffer9** ppIndexBuffer,
    HANDLE* pHandle
);
```

The CreateIndexBuffer function requires six parameters:

- **Length**. The size of the index data in bytes.
- **Usage**. A value of type D3DUSAGE that details how the buffer is to be used.
- **Format**. The format of the buffer indices. You have two choices here: D3DFMT_INDEX16 or D3DFMT_INDEX32. D3DFMT_INDEX16 means the indices are 16 bits each, and D3DFMT_INDEX32 means the indices are 32 bits each.
- **Pool**. The type of memory to be used in the index buffer creation.
- **ppIndexBuffer**. An address to a pointer where the newly created index buffer will reside.
- **pHandle**. A reserved value that should be NULL.

05 DX9_GP CH05 3/12/04 4:15 PM Page 93

A sample call to `CreateIndexBuffer` is shown here.

```
// Create the index buffer
hr = pd3dDevice->CreateIndexBuffer(sizeof(IndexData)*sizeof(WORD),
                                    D3DUSAGE_WRITEONLY,
                                    D3DFMT_INDEX16,
                                    D3DPOOL_DEFAULT,
                                    &iBuffer,
                                    NULL);
```

The `CreateIndexBuffer` call is similar to the `CreateVertexBuffer` function you've used before. The major difference between the two functions is the third parameter, which specifies the format of the indices that will be in the buffer. You have the option of 16- or 32-bit indices, which allows you to define your indices as either `WORD` or `DWORD` types.

Previously, I showed you how to create a cube using vertex buffers. The cube required 24 overlapping vertices to create 12 triangles. Using index buffers, you can create the same cube using only eight vertices. The next section shows you how to do this.

Generating a Cube by Using Index Buffers

The first step to creating a cube using index buffers is to define the vertices and the indices that will go into making up the model you're trying to create. As before when you were creating an object using vertex buffers, it's easiest to define the values in an array.

You define the vertices first, again using the `CUSTOMVERTEX` type you created previously. Each vertex has an `X`, `Y`, and `Z` value as well as a color component.

```
/ vertices for the vertex buffer
CUSTOMVERTEX g_Vertices[ ] = {
    // X      Y      Z     U     V
    {-1.0f,-1.0f,-1.0f, D3DCOLOR_ARGB(0,0,0,255)},      // 0
    {-1.0f, 1.0f,-1.0f, D3DCOLOR_ARGB(0,0,0,255)},      // 1
    {1.0f, 1.0f,-1.0f,  D3DCOLOR_ARGB(0,0,0,255)},      // 2
    { 1.0f,-1.0f,-1.0f, D3DCOLOR_ARGB(0,0,0,255)},      // 3
    {-1.0f,-1.0f, 1.0f, D3DCOLOR_ARGB(0,0,0,255)},      // 4
    {1.0f,-1.0f, 1.0f,  D3DCOLOR_ARGB(0,0,0,255)},      // 5
    { 1.0f, 1.0f, 1.0f, D3DCOLOR_ARGB(0,0,0,255)},      // 6
    {-1.0f, 1.0f, 1.0f, D3DCOLOR_ARGB(0,0,0,255)}       // 7
};
```

After you have the vertices defined, the next step is to generate the indices. The indices, like the vertices, are defined and stored in an array. Remember when I mentioned that the format of the indices could be 16 or 32 bits? This is where that choice comes into play.

The following code shows the array of indices that will go into creating the cube.

```
// index buffer data
WORD IndexData[ ] = {
        0,1,2,                          // triangle 1
        2,3,0,                          // triangle 2
        4,5,6,                          // triangle 3
        6,7,4,                          // triangle 4
        0,3,5,                          // triangle 5
        5,4,0,                          // triangle 6
        3,2,6,                          // triangle 7
        6,5,3,                          // triangle 8
        2,1,7,                          // triangle 9
        7,6,2,                          // triangle 10
        1,0,4,                          // triangle 11
        4,7,1                           // triangle 12
};
```

The previous IndexData array has split 36 indices into 12 groups, each consisting of 3 values that make up a triangle. Twelve triangles are needed to make a cube, using two triangles per face.

note

Remember: If you are tight on memory and your model doesn't require a DWORD type for your indices, use a WORD type instead.

Creating and Filling the Index Buffer

Now that the values you need for the index buffer have been defined, you need to copy them into the index buffer. This process is similar to copying vertices into a vertex buffer.

First, you lock the index buffer by using the Lock function. From there, you copy the indices into the buffer by using the memcpy function and then unlock the buffer. The result is an index buffer that contains the indices you need to render the cube.

The following code shows the process of creating and filling an index buffer with data.

```
// the index buffer
LPDIRECT3DINDEXBUFFER9 iBuffer;
HRESULT hr;

// Create the index buffer
```

```
hr = pd3dDevice->CreateIndexBuffer(sizeof(IndexData)*sizeof(WORD),
                                    D3DUSAGE_WRITEONLY,
                                    D3DFMT_INDEX16,
                                    D3DPOOL_DEFAULT,
                                    &iBuffer,
                                    NULL);

// Check to make sure the index buffer was created successfully
if FAILED(hr)
    return false;

// Prepare to copy the indexes into the index buffer
VOID* IndexPtr;

// Lock the index buffer
hr = iBuffer ->Lock(0, 0, (void**)& IndexPtr, D3DLOCK_DISCARD);

// Check to make sure the index buffer can be locked
if FAILED (hr)
    return hr;

// Copy the indices into the buffer
memcpy( pVertices, IndexData, sizeof(IndexData) );

// Unlock the index buffer
iBuffer->Unlock();
```

After the index buffer is filled with data, you can use the vertex and index data together to render your object.

Rendering the Cube with Index Buffers

Before, when you were drawing using vertex buffers, you used the `DrawPrimitive` function. The `DrawPrimitive` function used the data in the vertex buffer to create primitives, such as triangle strips and triangle lists. You can draw in a similar way using index buffers and the `DrawIndexedPrimitive` function.

The `DrawIndexedPrimitive` function uses an index buffer as its data source and renders graphic primitives to draw your 3D objects. The `DrawIndexedPrimitive` function is defined here.

```
HRESULT DrawIndexedPrimitive(
    D3DPRIMITIVETYPE Type,
```

05 DX9_GP CH05 3/12/04 4:15 PM Page 96


```
    INT BaseVertexIndex,
    UINT MinIndex,
    UINT NumVertices,
    UINT StartIndex,
    UINT PrimitiveCount
);
```

The `DrawIndexedPrimitive` function takes six parameters:

- **`Type`**. The primitive type to use when rendering the index data
- **`BaseVertexIndex`**. The starting index within the vertex buffer
- **`MinIndex`**. The minimum vertex index for this call
- **`NumVertices`**. The number of vertices that are used in this call
- **`StartIndex`**. The location in the vertex array to start reading vertices
- **`PrimitiveCount`**. The number of primitives to draw

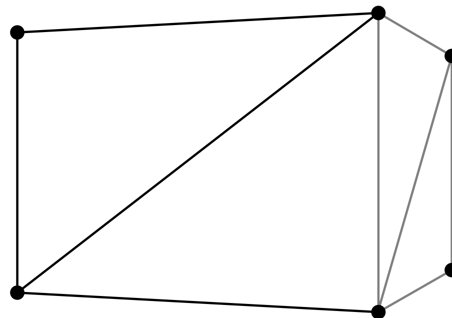

Figure 5.2 A cube with edge vertices highlighted.

Figure 5.2 shows a cube rendered in wireframe mode with the edge vertices highlighted. The edge vertices demonstrate the vertices referred to in the index buffers.

```
// Set the indices to use
m_pd3dDevice->SetIndices( m_pDolphinIB );
// Call DrawIndexedPrimitive to draw the object using the indices
m_pd3dDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST,
                                    0,                          // BaseVertexIndex
                                    0,                          // MinIndex
                                    m_dwNumDolphinVertices,     // NumVertices
                                    0,                          // StartIndex
                                    m_dwNumDolphinFaces );      // primitive count
```

Right before the call to `DrawIndexedPrimitive` is the `SetIndices` function. The `SetIndices` function, defined next, tells Direct3D which index buffer is going to be used as the data source when drawing. The `SetIndices` function works in much the same way as the `SetStreamSource` function does when you're drawing with vertex buffers.

```
HRESULT SetIndices(
    IDirect3DIndexBuffer9 *pIndexData
);
```

The `SetIndices` function requires only a single parameter: a pointer to an index buffer containing valid index data.

05 DX9_GP CH05 3/12/04 4:15 PM Page 97

The Geometry Pipeline

So far you've been using pretransformed coordinates to draw your objects to the screen. That means that the object's position is basically predefined in screen coordinates. This really restricts the size of the world and the movement of the objects within it.

3D models are for the most part created outside of your game code. For instance, if you're creating a racing game, you'll probably create the car models in a 3D art package. During the creation process, these models will be working off of the coordinate system provided to them in the modeler. This causes the objects to be created with a set of vertices that aren't necessarily going to place the car model exactly where and how you want it in your game environment. Because of this, you will need to move and rotate the model yourself. You can do this by using the geometry pipeline. The *geometry pipeline* is a process that allows you to transform an object from one coordinate system into another.

When a model first starts out, it is normally centered on the origin. This causes the model to be centered in the environment with a default orientation. Not every model you load needs to be at the origin, so how do you get models where they need to be? The answer to that is through transformations. Figure 5.3 shows a cube centered on the origin.

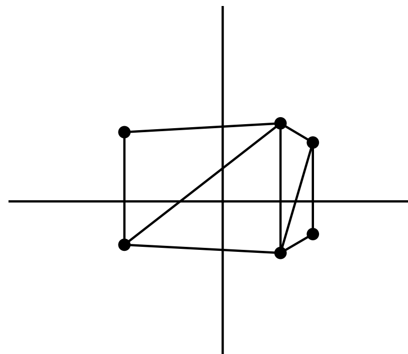

Figure 5.3 A cube centered on the origin.

Transformations refer to the actions of translating (moving), rotating, and scaling 3D objects. By applying these actions to a model, you can make the model appear to move around. These actions are handled through the geometry pipeline.

Figure 5.4 shows the different stages of the geometry pipeline.

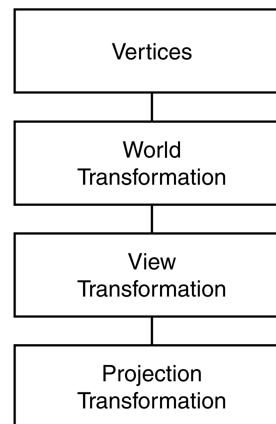

Figure 5.4 The stages of the geometry pipeline.

When you load a model, its vertices are in a local coordinate system called model space. *Model space* refers to the coordinate system on which the model is based that is independent of the rest of the world. For instance, upon creation, a model's vertices are in reference to the origin point around which they were created. A cube that is 2 units in size centered on the origin would have its vertices 1 unit on either side of the origin. If you then wanted to place this cube somewhere within your game, you would need to transform its vertices from the cube's local coordinate system into the system used by all the objects in your world. This world coordinate system is called *world space*, and the process of transforming vertices into this system is called *world transformation*.

05 DX9_GP CH05 3/12/04 4:15 PM Page 98

World Transformation

The world transformation stage of the geometry pipeline takes an existing object with its own local coordinate system and transforms that object into the world coordinate system. The world coordinate system is the system that places all objects within the 3D world in their proper locations. The world system has a single origin point that all models that are transformed into this system then become relative to. Figure 5.5 shows multiple objects within a 3D scene relative to the world origin point.

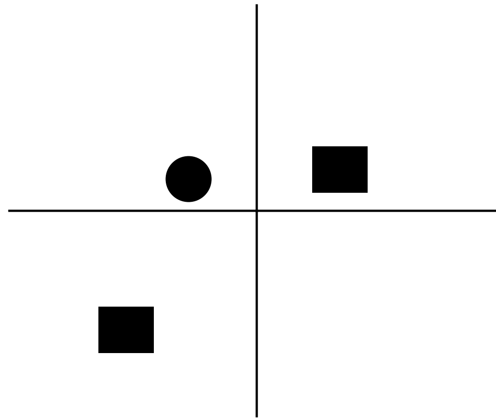

Figure 5.5 Multiple objects relative to a single world system origin point.

The next stage of the geometry pipeline is the view transformation. Because all objects at this point are relative to a single origin, you can only view them from this point. To allow you to view the scene from any arbitrary point, the objects must go through a view transformation.

View Transformation

The *view transformation* transforms the coordinates from world space into camera space. *Camera space* refers to the coordinate system that is relative to the position of a virtual camera. When you choose a point of view for your virtual camera, the coordinates in world space get reoriented in respect to the camera.

note

I've been saying "virtual camera" instead of "camera" because the concept of a camera in 3D doesn't really exist. By either moving the virtual camera up along the Y axis or by moving the entire world down along that same axis, you obtain the same visual results.

At this point, you have the camera angle and view for your scene, and you're ready to display it to the screen.

Projection Transformation

The next stage in the geometry pipeline is the projection transformation. The projection transformation is the stage of the pipeline where depth is applied. When you cause objects that are closer to the camera to appear larger than those farther away, you create an illusion of depth.

05 DX9_GP CH05 3/12/04 4:15 PM Page 99

Finally, the vertices are scaled to the viewport and projected into 2D space. The resulting 2D image appears on your monitor with the illusion of being a 3D scene. Table 5.1 shows the types of transformations within the geometry pipeline and the types of spaces each one affects.

Table 5.1 Coordinate System Transformations

Transformation Type	From Space	To Space
World transformation	Model space	World space
View transformation world space	View space	
Projection transformation	View space	Projection space

What Is a Matrix?

A matrix, in simplest terms, is an array of numbers that are arranged in columns and rows. Shown here is a simple 4 × 4 matrix containing the values 1 through 16.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Matrices are used within 3D to represent the transformations needed to move objects between coordinate spaces. The values that are contained in matrices are used to translate, rotate, and scale objects. Each row in the matrix represents the world coordinate of each axis. The first row contains the coordinate position of the X axis, the second row contains the Y axis position, and the third row contains the Z axis position.

Each position in the matrix represents a portion of a transformation.

For instance, positions 13, 14, and 15 hold the current X, Y, and Z position of a vertex. Positions 1, 6, and 11 contain the scaling values.

A matrix can be defined in code like this:

```
float matrix [4][4] = {
1.0f, 0.0f, 0.0f, 0.0f,
0.0f, 1.0f, 0.0f, 0.0f,
0.0f, 0.0f, 1.0f, 0.0f,
2.0f, 3.0f, 2.0f, 1.0f
};
```

05 DX9_GP CH05 3/12/04 4:15 PM Page 100

The final row of the previous matrix places the object with an X value of 2.0f, a Y value of 3.0f, and a Z value of 2.0f.

The Identity Matrix

The identity matrix is the default matrix that centers an object about the world origin and sets the object's scaling to 1. When you place a value of 1.0f in the 1, 6, and 11 positions, an object scaling of 1.0f is generated. Positions 13, 14, and 15 hold a value of 0.0f.

Following is an identity matrix.

```
float IdentityMatrix [4][4] = {
1.0f, 0.0f, 0.0f, 0.0f,
0.0f, 1.0f, 0.0f, 0.0f,
0.0f, 0.0f, 1.0f, 0.0f,
0.0f, 0.0f, 0.0f, 1.0f
};
```

If you ever need to get an object back to the world origin, you can translate the object's vertices by the identity matrix. The object is returned to the origin with no rotation or scaling applied. You are then free to move the object anywhere within your world.

Initializing a Matrix

Initializing or updating a matrix is as simple as changing individual values within the array. For instance, if you were given the identity matrix shown earlier and wanted to apply a translation to move an object 5 units along the X axis and 3 units along the Y axis, you would update the matrix like this:

```
Matrix[0][4] = 5.0f;
Matrix[1][4] = 3.0f;
Matrix[2][4] = 0.0f;
```

The resulting updated matrix would look like this.

```
float Matrix [4][4] = {
1.0f, 0.0f, 0.0f, 0.0f,
0.0f, 1.0f, 0.0f, 0.0f,
0.0f, 0.0f, 1.0f, 0.0f,
5.0f, 3.0f, 0.0f, 1.0f
};
```

The resulting matrix then contains the needed values to transform an object by the required units.

05 DX9_GP CH05 3/12/04 4:15 PM Page 101

Multiply Matrices

You're probably wondering how the matrices affect the vertices within an object. Well, each vertex in the object is multiplied individually by the matrix, resulting in a transformed vertex.

The math involved in this is fairly simple. To get the transformed X vertex, each value in the first row of the matrix is multiplied by each portion of the original vertex. The results from each of these multiplications are totaled to get the final transformed vertex. The complete method for this process is shown here.

$$\begin{matrix} X \\ Y \\ Z \\ W \end{matrix} \times \begin{matrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ M & N & O & P \end{matrix} = \begin{matrix} AX + BY + CZ + DW \\ EX + FY + GZ + HW \\ IX + JY + KZ + LW \\ MX + NY + OZ + PW \end{matrix} = \begin{matrix} X \text{ (transformed)} \\ Y \text{ (transformed)} \\ Z \text{ (transformed)} \\ W \text{ (transformed)} \end{matrix}$$

The far-left column shows the matrix that the vertex will be transformed by. The next column represents the vertex being transformed. The third column demonstrates the matrix being multiplied by the vertex. The final column on the right shows the resulting transformed vertex.

As you can see, multiplying a matrix by a vector is pretty straightforward, although multiplying two matrices can get a little complicated. You can accomplish matrix multiplication by multiplying the values in each row of the first matrix by the values in each column in the second. The key to multiplying matrices is to do it one step at a time to simplify the process.

Here's a simple example to demonstrate matrix multiplication. First you define the two matrices side by side. Letters represent the values in the first matrix, whereas numbers represents the values in the second matrix. This makes it easier to describe the math involved.

$$\begin{matrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ M & N & O & P \end{matrix} \times \begin{matrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{matrix}$$

You start by multiplying each value from the rows in the first matrix by the values in the columns of the second matrix. Through the multiplication process, you are going to end up creating a third output matrix that will contain the results of the multiplication.

The first value in the output matrix is calculated like this:

$$A \times 1 + B \times 5 + C \times 9 + D \times 13$$

You need to perform four multiplies just to gain a single value for the output matrix. You calculate the successive values for the output matrix by continuing to follow this pattern.

At this point, you're probably thinking that the math and data involved will get daunting pretty quickly. Direct3D tries to help by defining its own matrix data type.

note

Matrix multiplication is not cumulative. Multiplying Matrix A by Matrix B does not result in the same output matrix as multiplying Matrix B by Matrix A. The order in which matrices are multiplied is important.

How Direct3D Defines a Matrix

Until now, you've been defining a matrix by using a 4×4 array of float values. Direct3D simplifies this for you by providing the D3DMATRIX data type, defined here.

```
typedef struct _D3DMATRIX {
    union {
        struct {
        float        _11, _12, _13, _14;
        float        _21, _22, _23, _24;
        float        _31, _32, _33, _34;
        float        _41, _42, _43, _44;
        };
        float m[4][4];
    };
} D3DMATRIX;
```

By using the D3DMATRIX data type that Direct3D provides, you are given a host of common functions for performing tasks such as initializing new matrices.

D3DX Makes Matrices Easier

Previously, you were introduced to the D3DMATRIX data type that Direct3D provides. It helps to simplify the definition and maintenance of matrices but still leaves you to perform all the calculations yourself; this is where the D3DX utility library can help.

The D3DX library declares the D3DXMATRIX data type. The values within the D3DXMATRIX structure are identical to those found in a D3DMATRIX structure, but the D3DXMATRIX type gives you an added bonus. It provides some built-in functions for handling matrix calculations and comparisons.

The D3DXMATRIX type is defined here.

```
typedef struct D3DXMATRIX : public D3DMATRIX {
public:
    D3DXMATRIX() {};
```

05 DX9_GP CH05 3/12/04 4:15 PM Page 103


```
    D3DXMATRIX( CONST FLOAT * );
    D3DXMATRIX( CONST D3DMATRIX& );
    D3DXMATRIX( FLOAT _11, FLOAT _12, FLOAT _13, FLOAT _14,
                FLOAT _21, FLOAT _22, FLOAT _23, FLOAT _24,
                FLOAT _31, FLOAT _32, FLOAT _33, FLOAT _34,
                FLOAT _41, FLOAT _42, FLOAT _43, FLOAT _44 );

    // access grants
    FLOAT& operator () ( UINT Row, UINT Col );
    FLOAT  operator () ( UINT Row, UINT Col ) const;

    // casting operators
    operator FLOAT* ();
    operator CONST FLOAT* () const;

    // assignment operators
    D3DXMATRIX& operator *= ( CONST D3DXMATRIX& );
    D3DXMATRIX& operator += ( CONST D3DXMATRIX& );
    D3DXMATRIX& operator -= ( CONST D3DXMATRIX& );
    D3DXMATRIX& operator *= ( FLOAT );
    D3DXMATRIX& operator /= ( FLOAT );

    // unary operators
    D3DXMATRIX operator + () const;
    D3DXMATRIX operator - () const;

    // binary operators
    D3DXMATRIX operator * ( CONST D3DXMATRIX& ) const;
    D3DXMATRIX operator + ( CONST D3DXMATRIX& ) const;
    D3DXMATRIX operator - ( CONST D3DXMATRIX& ) const;
    D3DXMATRIX operator * ( FLOAT ) const;
    D3DXMATRIX operator / ( FLOAT ) const;

    friend D3DXMATRIX operator * ( FLOAT, CONST D3DXMATRIX& );

    BOOL operator == ( CONST D3DXMATRIX& ) const;
    BOOL operator != ( CONST D3DXMATRIX& ) const;
} D3DXMATRIX, *LPD3DXMATRIX;
```

The first thing you'll probably notice about the `D3DXMATRIX` type is that it's a structure that inherits from `D3DMATRIX` and includes functions that make it appear like a C++ class.

05 DX9_GP CH05 3/12/04 4:15 PM Page 104

Because of the way this type is defined, you can only access it through C++, and it's treated as a full class with only public member functions.

If you look through the structure, you'll see functions that overload a lot of the assignment and comparison operators, as well as those used for calculating matrix operations. Because the `D3DXMATRIX` data structure is so much more useful, I'll continue to use it throughout the examples.

Manipulating 3D Objects by Using Matrices

Now that you know what matrices are, I'm going to tell you how they can be useful. You use matrices when you're manipulating objects in a scene. Whether you want to move an object around or just rotate it, you'll need matrices to do the job.

D3DX provides multiple functions that make manipulating objects easier by using matrices. I've listed a few of them here.

- **D3DXMatrixIdentity**. Resets a matrix to the origin
- **D3DXMatrixRotationX**. Rotates an object around the X axis
- **D3DXMatrixRotationY**. Rotates an object around the Y axis
- **D3DXMatrixScaling**. Scales an object by a specified amount
- **D3DXMatrixTranslation**. Moves an object along one or more axes

Moving an Object Around

To move an object around in your game world, you must translate it. *Translation* refers to the movement of an object along one or more of the coordinate system axes. If you wanted to move an object in your scene to the right, you would have to translate it along the X axis in a positive direction.

Translation of objects is handled through the `D3DXMatrixTranslation` function, defined next.

```
D3DXMATRIX *D3DXMatrixTranslation(
    D3DXMATRIX *pOut,
    FLOAT x,
    FLOAT y,
    FLOAT z
);
```

The `D3DXMatrixTranslation` function requires just four parameters.

- **pOut**. The output matrix. This parameter is a pointer to a `D3DXMATRIX` object.
- **x**. The amount to translate the object along the X axis. This can be a positive or a negative value.

05 DX9_GP CH05 3/12/04 4:15 PM Page 105

- **y**. The amount to translate the object along the Y axis.
- **z**. The amount to translate the object along the Z axis.

The small sample of code that follows shows how to use the `D3DXMatrixTranslation` function.

```
D3DXMATRIX matTranslate;
D3DXMATRIX matFinal;

// Set the matFinal matrix to the identity
D3DXMatrixIdentity(&matFinal);

// Translate the object 64 units to the right of the origin along the X axis
// The resulting translated matrix is stored in the matTranslate variable
D3DXMatrixTranslation(&matTranslate, 64.0f, 0.0f, 0.0f);

// Multiply the translation and identity matrix together to get the final
// translated matrix stored in the finalMat variable
D3DXMatrixMultiply(&finalMat, &finalMat, & matTranslate);

// Transform the object in world space
pd3dDevice->SetTransform(D3DTS_WORLD, &finalMat);
```

The `D3DXMatrixTranslation` function is being used to translate an object 64 units to the right along the X axis. To apply the translation to the object, multiply the translation matrix by the identity matrix; then the object will be transformed into world space.

Rotating an Object

Being able to move an object along the axes is nice, but you're really limiting what your game can do. What fun would a racing game be if you couldn't drive the car around the track because the car was restricted to moving only in straight lines? That is why you need rotation. Being able to rotate the car enables it to make turns and follow the curves of the track.

Rotating 3D objects works alongside translation to give your characters freedom of movement within their environment. Rotation allows wheels on cars to spin, an arm to swing at the side of your character, or a baseball to curve right before it goes over the plate.

Rotating is the process of spinning an object around a coordinate system axis. Because rotation takes place using matrices, the D3DX library provides some helper functions to make rotating easier.

05 DX9_GP CH05 3/12/04 4:15 PM Page 106

Rotation occurs along a single axis at any one time and can take place on any of the three axes. D3DX provides a specific rotation function to handle rotating around each axis. For instance, if you wanted to rotate an object around the X axis, you would use the function `D3DXMatrixRotationX`, defined here.

```
D3DXMATRIX *D3DXMatrixRotationX(
    D3DXMATRIX *pOut,
    FLOAT Angle
);
```

The `D3DXMatrixRotationX` function takes just two parameters:

- **`pOut`**. A pointer to a `D3DXMATRIX` object. This holds the resulting rotation matrix.
- **`Angle`**. The angle, in radians, to rotate the object.

Using the `D3DXMatrixRotationX` function or any of its derivatives is simple. First, define a `D3DXMATRIX` structure to hold the rotation matrix, and then input the angle to rotate the object. The short code that follows shows how easy this function is to use.

```
D3DXMATRIX matRotate;                              // This is the output matrix
D3DXMatrixRotationX(&matRotate, D3DXToRadian(45.0f));
```

You define the output matrix and then call the `D3DXMatrixRotationX` function. You'll notice that the second parameter is using a helper macro called `D3DXToRadian`. This macro takes an angle from 0 to 360 and converts it to radians. In the previous example, the angle of rotation is 45 degrees.

The result of this rotation is the object rotating around the X axis by 45 degrees.

Figure 5.6 shows how a cube that is rotating around the Y axis behaves.

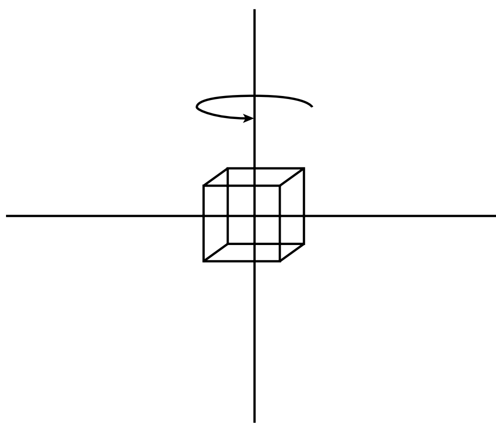

Figure 5.6 A cube rotating around the Y axis.

The following code shows what you need to rotate a cube around the Y axis. The rotation is based on a timer that allows the cube to rotate continuously.

```
/*************************************************************************
* render
*************************************************************************/
void render(void)
{
```

05 DX9_GP CH05 3/12/04 4:15 PM Page 107


```
    D3DXMATRIX objMat, matRotate, finalMat;

    // Clear the back buffer to a black color
    pd3dDevice->Clear( 0,
                      NULL,
                      D3DCLEAR_TARGET,
                      D3DCOLOR_XRGB(255,255,255),
                      1.0f,
                      0 );

    pd3dDevice->BeginScene();

    // Set the vertex stream
    pd3dDevice->SetStreamSource( 0, vertexBuffer, 0, sizeof(CUSTOMVERTEX) );
    // Set up the vertex format for the object
    pd3dDevice->SetFVF( D3DFVF_CUSTOMVERTEX );

    // Set meshMat to identity
    D3DXMatrixIdentity(&objMat);

    // Set the rotation
    D3DXMatrixRotationY(&matRotate, timeGetTime()/1000.0f);

    // Multiply the scaling and rotation matrices to create the objMat matrix
    D3DXMatrixMultiply(&finalMat, &objMat, &matRotate);

    // Transform the object in world space
    pd3dDevice->SetTransform(D3DTS_WORLD, &finalMat);

    // Render the cube using triangle strips
    pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP,  0, 2 );
    pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP,  4, 2 );
    pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP,  8, 2 );
    pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 12, 2 );
    pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 16, 2 );
    pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 20, 2 );

    pd3dDevice->EndScene();

    // Present the back buffer contents to the display
    pd3dDevice->Present( NULL, NULL, NULL, NULL );
}
```

Three variables are declared at the start of the render function: `objMat`, `matRotate`, and `finalMat`. These variables are the matrices that will store the cube's transformations. Earlier I showed you how to reset a matrix to represent the origin by setting it to the identity matrix; the `objMat` matrix will need to be reset each time the render function is called. This causes the rotations that you will apply to the cube to be centered on the origin. This is accomplished by using the `D3DXMatrixIdentity` function. The `objMat` matrix represents the actual position of the cube.

```
D3DXMatrixIdentity(&objMat);
```

The second matrix, `matRotate`, holds the rotation information for the cube. Because the cube is going to be in continuous motion, you must update the `matRotate` matrix each frame with the new position. The rotation takes place by using `D3DXMatrixRotationY`, which is one of the D3DX helper functions. The D3DX rotation functions overwrite the matrix each frame with the new rotation information, so you don't need to call the `D3DXMatrixIdentity` function to reset this matrix.

```
D3DXMatrixRotationY(&matRotate, timeGetTime()/1000.0f);
```

The `timeGetTime` function uses the current time divided by `1000.0f` to allow the cube to rotate in a smooth manner.

Now that you have two matrices—one representing the position of the object and the other representing its movement—you need to multiply the two matrices to create the final matrix represented by the `finalMat` variable.

The resulting matrix transforms the cube into world space by using the `SetTransform` function shown here.

```
pd3dDevice->SetTransform(D3DTS_WORLD, &finalMat);
```

The `SetTransform` function results in the cube being placed in its new position and orientation in world space. The `render` function draws the cube by using multiple calls to the `DrawPrimitive` function.

You can find the full source code for rotating an object in the chapter5\example2 directory on the CD-ROM.

Center of Rotation

The center of an object's rotation is based on the axis it is rotating around. If an object, such as the cube in Figure 5.6, were rotated, its center of rotation would cause it to spin around the origin. If an object were translated away from the origin and along one of the axes, its center of rotation would remain along the same axes and the object would be translated to a new position during the rotation.

05 DX9_GP CH05 3/12/04 4:15 PM Page 109

Look at Figure 5.7, which shows a cube translated along the X and Y axis before being rotated. When the cube in this figure is rotated along the X axis, the object is translated while the rotation occurs.

To change an object's center of rotation, you must translate the object away from the origin before you apply rotation. The following code shows how to translate an object so that you can change the center of rotation.

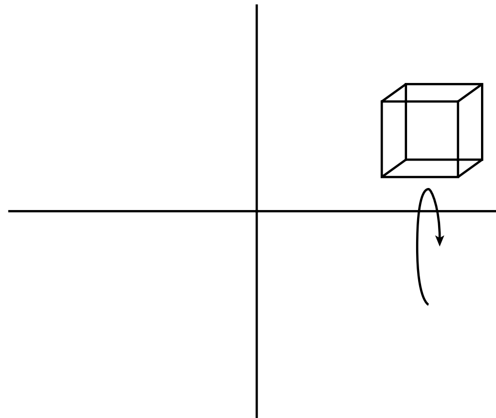

Figure 5.7 A cube being rotated around the X axis after being translated away from the origin.

```
/*************************************************************************
* render
*************************************************************************/
void render(void)
{
    // Clear the back buffer to a black color
    pd3dDevice->Clear( 0,
                        NULL,
                        D3DCLEAR_TARGET,
                        D3DCOLOR_XRGB(255,255,255),
                        1.0f,
                        0 );

    pd3dDevice->BeginScene();

    pd3dDevice->SetStreamSource( 0, vertexBuffer, 0, sizeof(CUSTOMVERTEX) );
    pd3dDevice->SetFVF( D3DFVF_CUSTOMVERTEX );

    // Translate the object away from the origin
    D3DXMatrixTranslation(&matTranslate, 64.0f, 0.0f, 0.0f);

    // Set the rotation
    D3DXMatrixRotationY(&matRotate, timeGetTime()/1000.0f);

    // Multiply the translation and rotation matrices to create the objMat matrix
    D3DXMatrixMultiply(&objMat, &matTranslate, &matRotate);
```

```
    // Transform the object in world space
    pd3dDevice->SetTransform(D3DTS_WORLD, &objMat);

    pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP,  0, 2 );
    pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP,  4, 2 );
    pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP,  8, 2 );
    pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 12, 2 );
    pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 16, 2 );
    pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 20, 2 );

    pd3dDevice->EndScene();

    // Present the back buffer contents to the display
    pd3dDevice->Present( NULL, NULL, NULL, NULL );
}
```

The biggest change to the `render` function is the addition of the `D3DXMatrixTranslation` function. The `D3DXMatrixTranslation` function moves the cube 64 units away from the origin during the rotation.

In this case, the cube is being translated away from the origin along the X axis and then rotated. Two matrices are being used to move the cube: `matTranslate` and `matRotate`. The two matrices are then multiplied together to create the `objMat` matrix, which holds the final position of the cube. The result is the cube rotating away from the origin.

Scaling

Scaling allows you to change the size of an object by multiplying each vertex within the object by a specified amount. To perform scaling on an object, you need to create a matrix that contains the values by which to scale the object. The scaled values detail just how much to scale each vertex. Remember the matrix layout from earlier? The positions 1, 6, and 11 hold the scaling amounts for the X, Y, and Z axes, respectively. By default, these values are `1.0f` and the object remains its original size. Changing any of these values affects the size of the object. If the values that are placed in these spots are greater than `1.0f`, the object will be enlarged; alternatively, if the values are less than `1.0f`, the object can be shrunk.

X	2	3	4
5	Y	7	8
9	10	Z	12
13	14	15	16

As I mentioned previously, scaling takes place by manipulating values within a matrix. To create a scaling matrix, simply define an identity matrix and change the values in the positions I detailed earlier. You can either change these values manually or use the `D3DXMatrixScaling` function, defined here.

```
D3DXMATRIX *D3DXMatrixScaling(
    D3DXMATRIX *pOut,
    FLOAT sx,
    FLOAT sy,
    FLOAT sz
);
```

The `D3DXMatrixScaling` function takes four parameters:

- **pOut.** A pointer to a `D3DXMATRIX` object that will hold the scaling matrix
- **sx.** The amount to scale the X vertices
- **sy.** The amount to scale the Y vertices
- **sz.** The amount to scale the Z vertices

The code sample that follows shows how to use the `D3DXMatrixScaling` function to double the size of an object.

```
D3DXMATRIX matScale;

// Set the scaling
D3DXMatrixScaling(&matScale, 2.0f, 2.0f, 2.0f);

// Multiply the object's matrix against the scaling matrix
D3DXMatrixMultiply(&objMat, & objMat, &matScaling);
```

The `objMat` variable in the previous code represents the object's original matrix. Multiplying the object's matrix by a scaling matrix enables you to scale the object when you draw it.

Order of Matrix Operations

The order in which matrix operations are applied is important. For instance, if you want to rotate an object around its center and then move the object somewhere in your world, you first must apply the rotation matrix operation followed by the translation matrix. If these two matrix operations were reversed, the object would first be translated into its new position in the world and then rotated around the world's origin point. This could cause your object to be in the wrong place and orientation within your world. The code that follows shows how an object should be rotated and translated correctly.

```
D3DXMATRIX objRotate;
D3DXMATRIX objTranslation;
```

05 DX9_GP CH05 3/12/04 4:15 PM Page 112


```
D3DXMATRIX objFinal;

// Set the rotation
D3DXMatrixRotationY(&objRotate, D3DXToRadian(45));

// Apply the translation matrix
D3DXMatrixTranslation(&objTranslation, 1.0f, 0.0f, 0.0f);

// Multiply the rotation and translation matrices to create the final matrix
D3DXMatrixMultiply(&objFinal, &objRotate, &objTranslation);

// Transform the object in world space
pd3dDevice->SetTransform(D3DTS_WORLD, &objFinal);
```

The first step is to create the object's rotation matrix, `objRatate`. Using the `D3DXMatrixRotationY` function, the object is made to rotate 45 degrees around the Y axis.

Next, you translate the rotated object 1 unit to the right using the `D3DXMatrixTranslation` function.

Finally, you create the object's final transformed matrix by multiplying the rotation and translation matrices together using the `D3DXMatrixMultiply` function. If the rotation and translation matrices were to reverse positions in the `D3DXMatrixMultiply` call, the translation would take place before the rotation, dislocating the object.

Creating a Camera by Using Projections

You create a camera in Direct3D by defining a matrix for the projection transformation. This matrix defines the field of view (FOV) for the camera, the aspect ratio, and the near and far clipping planes.

After you've created the projection matrix, you apply it to your scene through the `SetTransform` function. You've probably noticed the `SetTransform` function used in the sample code earlier in this chapter. The `SetTransform` function, defined next, sets a matrix to a particular stage of the geometry pipeline. For instance, when you're setting the matrix for a camera, you are setting how the scene is going to be viewed during the projection stage. This stage, as the final part of the geometry pipeline, controls how the 3D scene is rendered to the 2D display.

```
HRESULT SetTransform(
    D3DTRANSFORMSTATETYPE State,
    CONST D3DMATRIX *pMatrix
);
```

The SetTransform function requires two parameters:

- **State**. The stage of the pipeline that is being modified
- **pMatrix**. A pointer to a D3DMATRIX structure that is to be applied to the pipeline

The code sample that follows shows how to create and define a matrix to be used for the projection stage.

```
D3DXMATRIX matProj;                              // the projection matrix
/*********************************************************************
* createCamera
* creates a virtual camera
*********************************************************************/
void createCamera(float nearClip, float farClip)
{
    // Here, you specify the field of view, aspect ratio,
    // and near and far clipping planes
    D3DXMatrixPerspectiveFovLH(&matProj, D3DX_PI/4, 640/480, nearClip, farClip);

    // Apply the matProj matrix to the projection stage of the pipeline
    pd3dDevice->SetTransform(D3DTS_PROJECTION, &matProj);
}
```

Instead of creating the projection matrix by hand, I used the D3DX helper function D3DXMatrixPerspectiveFovLH. This function creates an output matrix, held in the matProj variable earlier, by allowing you to specify the perspective, aspect ratio, and clipping planes in a single function call.

After you have generated the projection matrix, you apply it to the geometry pipeline by way of the SetTransform function. Because this matrix affects the projection piece of the pipeline, the value D3DTS_PROJECTION is used.

Positioning and Pointing the Camera

At this point, you can use the camera as is. The camera affects everything in the scene as the objects pass through the projection part of the geometry pipeline. There's just one problem; the camera is located at the origin, pointing off into the distance. Because a camera in the real world is a movable object, you want your virtual camera to behave the same way. The camera needs to be able to move around the scene and also be able to change the direction it's pointing. To satisfy these two criteria, you need to change the matrix that controls the view stage of the pipeline.

By default, the view matrix is set to the identity matrix, keeping your virtual camera steadfast at the origin. To change the camera's position and orientation, you need to create a

05 DX9_GP CH05 3/12/04 4:15 PM Page 114

new view matrix. The easiest way to create this matrix is through the D3DX helper function `D3DXMatrixLookAtLH`.

The `D3DXMatrixLookAtLH` function allows you to specify the position of the camera (defined as a `D3DXVECTOR3`), where the camera is looking (using a `D3DXVECTOR3`), and the direction that the camera should consider as up (also represented by a `D3DXVECTOR3`).

Following is a small code sample that will give you an idea of how to create the view matrix.

```
D3DXMATRIX matView;            // the view matrix
/*************************************************************************
* pointCamera
* points the camera at a location specified by the passed vector
*************************************************************************/
void pointCamera(D3DXVECTOR3 cameraPosition, D3DXVECTOR3 cameraLook)
{
    D3DXMatrixLookAtLH (&matView,
                        &cameraPosition,    //camera position
                        &cameraLook,        //look at position
                        &D3DXVECTOR3 (0.0f, 1.0f, 0.0f));      //up direction

    // Apply the matrix to the view stage of the pipeline
    pd3dDevice->SetTransform (D3DTS_VIEW, &matView);
}
```

The `pointCamera` function allows two parameters to be passed into it: the `cameraLook` variable and the `cameraPosition` variable.

The `cameraPosition` variable holds the camera's current position. For instance, if the camera were located 2 units away from the origin along the Z axis, the `cameraPosition` variable would contain the vector `(0.0f, -2.0f, 0.0f)`.

The `cameraLook` variable tells the camera where it needs to point and is relative to the location of the camera. For example, assume that the camera is located 10 units up along the Y axis and 10 units back along the Z axis. Now imagine that you want the camera to point at the origin. Because the camera is currently residing above the origin, it actually needs to look down to see it. By setting the `cameraLook` vector to `(0.0f, -10.0f; 0.0f)`, you are telling the camera to remain looking straight ahead but to look downward along the Y axis. The camera will then see the objects at the origin from a slightly overhead view.

The final view matrix that the `D3DXMatrixLookAtLH` creates is stored in the `matView` variable and then applied to the view stage of the pipeline. The `D3DTS_VIEW` value that is passed to the first parameter of `SetTransform` informs Direct3D that the view projection matrix will be updated.

05 DX9_GP CH05 3/12/04 4:15 PM Page 115

Chapter Summary

In this chapter, you were introduced to the concepts that you'll use every time you write a 3D-based application. Whether you're creating a simple model viewer or the next first-person shooter, matrices and transforms are the foundation your games are built on.

What You Have Learned

In this chapter, you learned the following:

- How 3D objects are taken through the geometry pipeline
- What matrices are, and when and how to apply them
- How to move and rotate objects in a scene
- Why the order of matrix multiplication is important
- How to create and use a camera in your scene to view 3D objects

Review Questions

You can find the answers to Review Questions and On Your Own exercises in Appendix A, "Answers to End-of-Chapter Exercises."

1. The indices of an object can be stored in what kind of buffer?
2. What is a matrix?
3. What are the steps in the geometry pipeline?
4. What does the identity matrix do?
5. Changing a camera's aspect ratio affects which part of the pipeline?

On Your Own

1. Using the `D3DXMatrixMultiply` function, show the code needed to first rotate and then translate an object 5 units along the X axis.
2. Write a `render` function that will constantly rotate an object around the Y axis.

05 DX9_GP CH05 3/12/04 4:15 PM Page 116