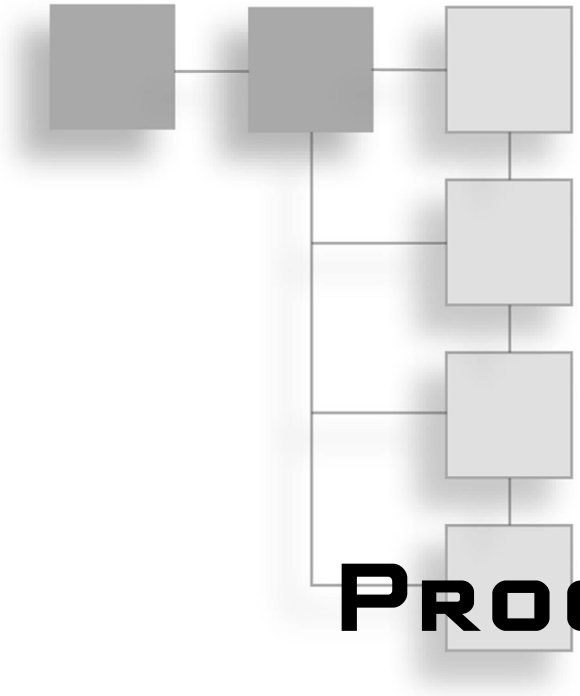

PART III

3D PROGRAMMING

Part III is dedicated to the subject of 3D programming.

CHAPTER 11

3D Graphics Fundamentals

This chapter covers the basics of 3D graphics. You will learn the basic concepts so that you are at least aware of the key points in 3D programming. However, this chapter will not go into great detail on 3D mathematics or graphics theory, which are far too advanced for this book. What you will learn instead is the practical implementation of 3D in order to write simple 3D games. You will get just exactly what you need to

write a simple 3D game without getting bogged down in theory. If you have questions about how matrix math works and about how 3D rendering is done, you might want to use this chapter as a starting point and then go on and read a book such as *Beginning Direct3D Game Programming*, by Wolfgang Engel (Course PTR). The goal of this chapter is to provide you with a set of reusable functions that can be used to develop 3D games.

Here is what you will learn in this chapter:

- How to create and use vertices.
- How to manipulate polygons.
- How to create a textured polygon.
- How to create a cube and rotate it.

Introduction to 3D Programming

It's a foregone conclusion today that everyone has a 3D accelerated video card. Even the low-end budget video cards are equipped with a 3D graphics processing unit (GPU) that would be impressive were it not for all the competition in this market pushing out more and more polygons and new features every year. Figure 11.1 shows a typical GeForce 4 card.

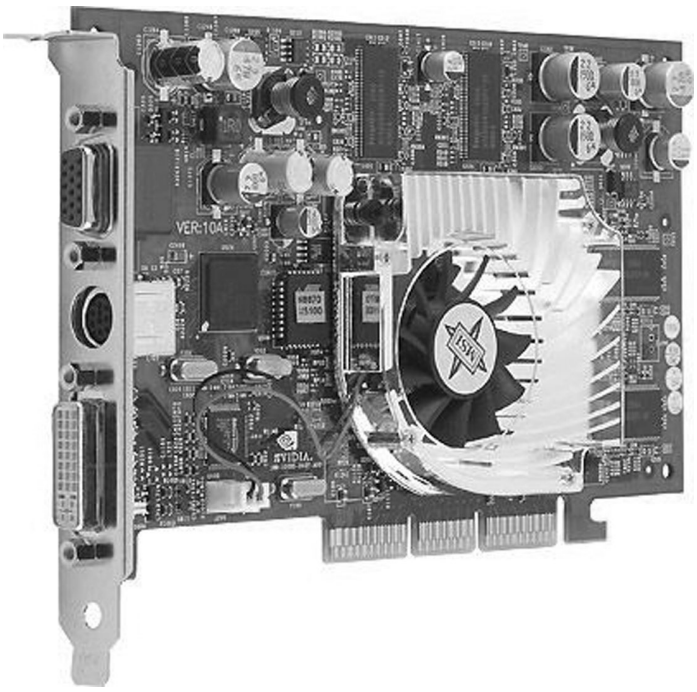

Figure 11.1 Modern 3D video cards are capable of producing real-time photorealistic graphics.

The Three Steps to 3D Programming

There are three steps involved in 3D graphics programming:

1. **World transformation.** This moves 3D objects around in the "world," which is a term that describes the entire scene. In other words, the world transformation causes things in the scene to move, rotate, scale, and so on.
2. **View transformation.** This is the camera, so to speak, that defines what you see on the screen. The camera can be positioned anywhere in the "world," so if you want to move the camera, you do so with the view transform.
3. **Projection transformation.** This is the final step, in which you take the view transform (what objects are visible to the camera) and draw them on the screen, resulting in a flat 2D image of pixels.

Direct3D provides all the functions and transformations that you need to create, render, and view a scene without using any 3D mathematics—which is good for you, the programmer, because 3D matrix math is not easy.

A transformation occurs when you add, subtract, multiply, or divide one matrix with another matrix, causing a change to occur within the resulting matrix; these changes cause 3D objects to move, rotate, and scale 3D objects. A matrix is a grid or two-dimensional array that is 4×4 (or 16 cells) in size. Direct3D defines all of the standard matrices that you need to do just about everything required for a 3D game.

The 3D Scene

Before you can do anything with the scene, you must first create the 3D objects that will make up the scene. In this chapter, I will show you how to create simple 3D objects from scratch, and will also go over some of the freebie models that Direct3D provides, mainly for testing. There are standard objects, such as a cylinder, pyramid, torus, and even a teapot, that you can use to create a scene.

Of course, you can't create an entire 3D game just with source code because there are too many objects in a typical game. Eventually, you'll need to create your 3D models in a modeling program like 3ds max or the free Anim8or program (included on the CD-ROM). The next two chapters will explain how to load 3D models from a file into a scene. But in this chapter, I'll stick with programmable 3D objects.

Introducing Vertices

The advanced 3D graphics chip that powers your video card sees only vertices. A *vertex* (singular) is a point in 3D space specified with the values of X, Y, and Z. The video card itself really only "sees" the vertices that make up the three angles of each triangle. It is the job of the video card to fill in the empty space that makes up the triangle between the three vertices. See Figure 11.2.

Creating and manipulating the 3D objects in a scene is a job for you, the programmer, so it helps to understand some of the basics of the 3D environment. The entire scene might be thought of as a mathematical grid with three axes. You might be familiar with the Cartesian coordinate system if you have ever studied geometry or trigonometry: The coordinate system is the basis for all geometric and trigonometric math, as there are formulas and functions for manipulating points on the Cartesian grid.

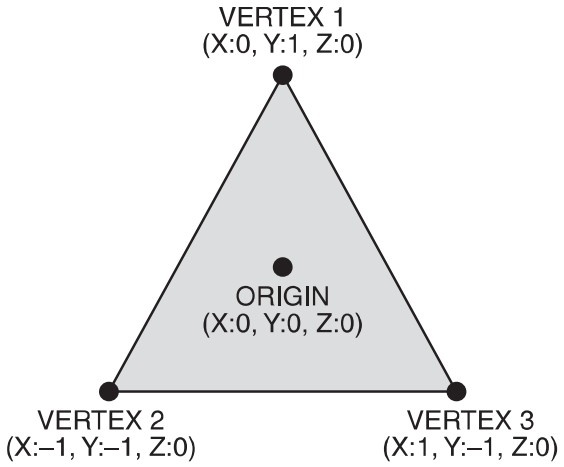

Figure 11.2 A 3D scene is made up entirely of triangles.

The Cartesian Coordinate System

The "grid" is really made up of two infinite lines that intersect at the origin. These lines are perpendicular. The horizontal line is called the *X axis* and the vertical line is called the *Y axis*. The origin is at position (0,0). The X axis goes up in value toward the right, and it goes down in value to the left. Likewise, the Y axis goes up in the up direction, and goes down in the down direction. See Figure 11.3.

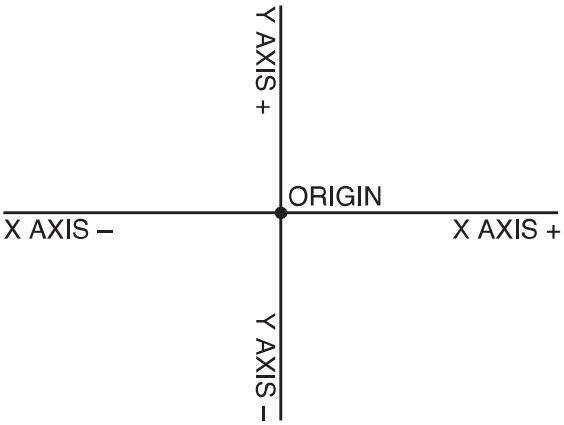

Figure 11.3 The Cartesian coordinate system

If you have a point at a specified position that is represented on a Cartesian coordinate system, such as at (100, –50), then you can manipulate that point using mathematical calculations. There are three primary things you can do with a point:

1. **Translation**. This is the process of moving a point to a new location. See Figure 11.4.
2. **Rotation**. This causes a point to move in a circle around the origin at a radius that is based on its current position. See Figure 11.5.
3. **Scaling**. You can adjust the point relative to the origin by modifying the entire range of the two axes. See Figure 11.6.

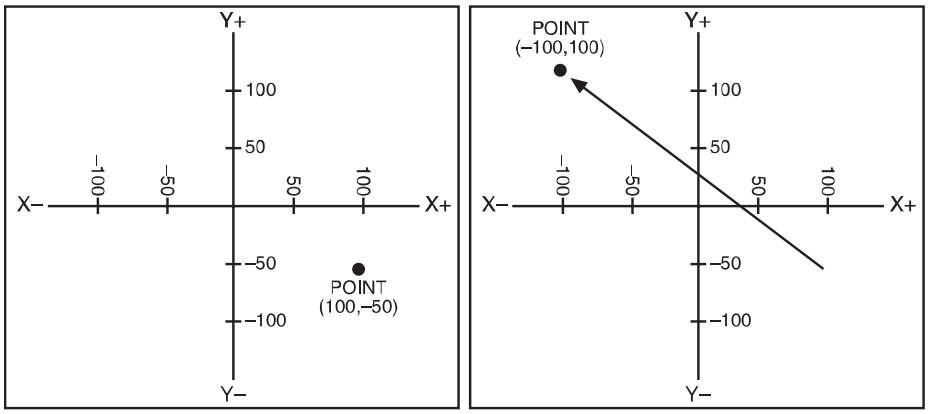

Figure 11.4 A point (100,–50) is translated by a value of (–200,150) resulting in a new position at (–100,100).

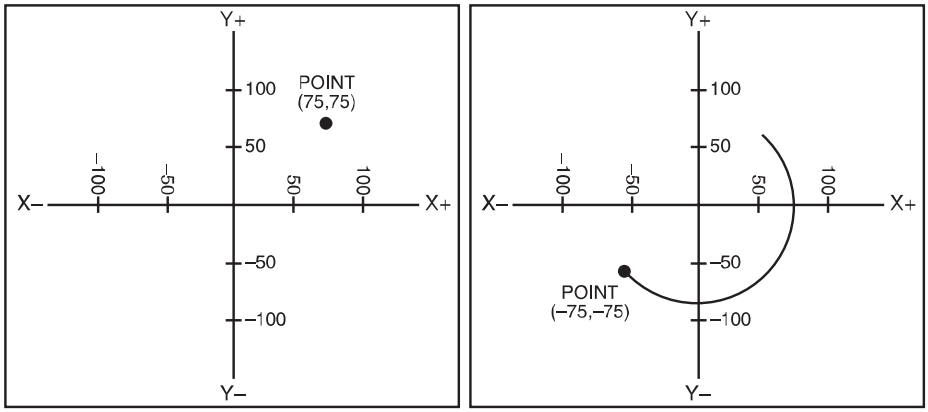

Figure 11.5 A point (75,75) is rotated by 180 degrees, resulting in a new position at (–75,–75).

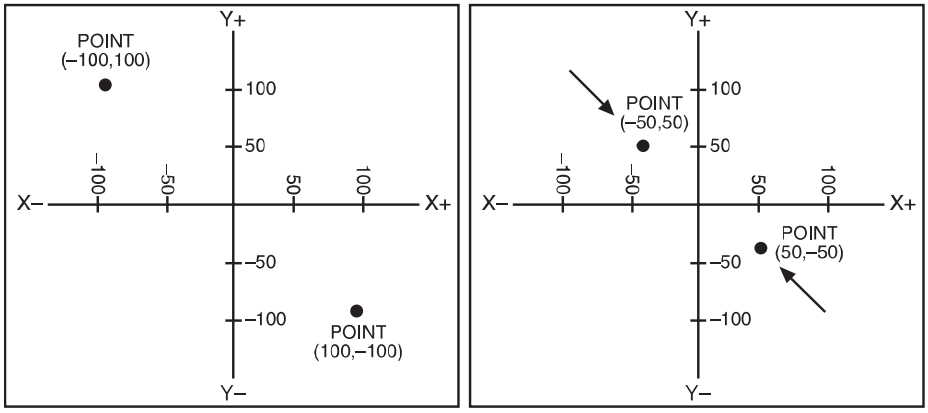

Figure 11.6 A point (–100,100) is scaled by –50 percent, resulting in a new position at (–50,50).

The Origin of Vertices

The one thing you want to remember when working with 3D graphics is that everything works around the origin. So, when you want to rotate a 3D object on the screen, you have to remember that all rotation is based on the origin point. If you translate the object to a new location that is no longer centered at the origin, then rotating the object will cause it to move around the origin in a circle!

So, what's the solution to this problem? This is the biggest sticking point most people run into with 3D programming because it's very hard to get a handle on it unless you have, say, a more senior programmer to explain it to you. In this case, you have an opportunity to learn an important lesson in 3D graphics programming that is all-too-often ignored: The trick is to not really move the 3D objects at all.

What!? No, I'm not kidding. The trick is to leave all of the 3D objects at the origin and not move them at all. Does that mess with your head? Okay, I'll explain myself. You know that a 3D object is made up of vertices (three for every triangle, to be exact). The key is to draw the 3D objects at a specified position, with a specified rotation and scaling value, without moving the "original" object itself. I don't mean that you should make a copy of it; instead, just draw it at the last instant before refreshing the screen. Do you remember how you were able to draw many sprites on the screen with only a single sprite image? It's sort of like that, only you're just drawing a 3D object based on the original "image," so to speak, and the original does not change. By leaving the source objects at the origin, you can rotate them around what is called a *local origin* for each object, which preserves the objects.

So, how do you move a 3D object without moving it? The answer is by using matrices. A *matrix* is a 4×4 grid of numbers that represent a 3D object in "space." Each 3D object in your scene (or game) has its own matrix.

note

As you might have guessed, matrix mathematics is a subject way, way, way beyond the scope of this book, but I encourage you to look into it if you want to learn what *really* happens in the world of polygons.

The result of using matrices to give each 3D object its own origin is that your 3D world has its own coordinate system—as do all of the objects in the scene—so you can manipulate objects independently of one another. You can even manipulate the entire scene without affecting these independent objects. For example, suppose you are working on a racing game, and you have cars racing around an oval track. You want each car to be as realistic as possible so that each car can rotate and move on its own, regardless of what the other cars are doing. At some point, of course, you want to add the code that will cause the cars to crash if they collide. You also want the cars to stay "flat" on the pavement of the

track, which means calculating the angle of the track and positioning the four corners of the car appropriately.

Imagine taking it even further—think of the possibilities than arise when you can cause individual objects to contain sub-objects, each with their own local origins, that follow along with the "parent" object. You can then position the sub-objects with respect to the origin of the parent object and cause the sub-objects to rotate on their own. Does this help you to visualize how you might program the wheels of a car to roll on their own while the car remains stationary? The wheels "follow along" with the car, meaning they translate/rotate/scale with the parent object, but they also have the ability to roll and turn left or right.

caution

The most frustrating problem with 3D programming is not seeing anything come up on the screen after you have written what you believe to be clean code that "should work, dang it!" The number one most common mistake in 3D programming is forgetting about the camera and view transform. As you work through this chapter, keep the following points in mind.

The first thing you should set up in the scene is the perspective, camera, and view with a test poly or quad to make sure your scene is set up properly before proceeding. Once you know for sure that the view is good, you can move ahead with the rest of the code for your game. Another frequent problem involves the position of the camera, which might seem okay for your initial test but then may be too close to the object for it to show up, or the object may have moved off "the screen." One good test is to move the camera away from the origin (such as a Z of –100, for instance), and then make sure your target matrix points to the origin (0,0,0). That should clear up any viewing problems and allow you to get cracking on the game again

The second thing you should do to initially set up the scene is check the lighting conditions of your scene. Do you have lighting enabled without any lights? Direct3D is really literal and will not create ambient light for you unless you tell it there will be no light sources!

Moving to the Third Dimension

I hope you are now getting the hang of the Cartesian coordinate system. Although it is crucial to the study of 3D graphics, I will not go into any more detail because the subject requires more theory and explanation than I have room for here. Instead, I'm going to just cover enough material to teach you what you need to know to write a few simple 3D games, after which you can decide which aspect of 3D programming you'd like to study further. It's always more fun to do what works first and work on an actual game rather than try to learn every nook and cranny of a library like Direct3D all at once.

Figure 11.7 shows the addition of a third dimension to the Cartesian coordinate system. All of the current rules that you have learned about the 2D coordinate system apply, but each point is now referred to with three values (X,Y,Z) instead of just the two.

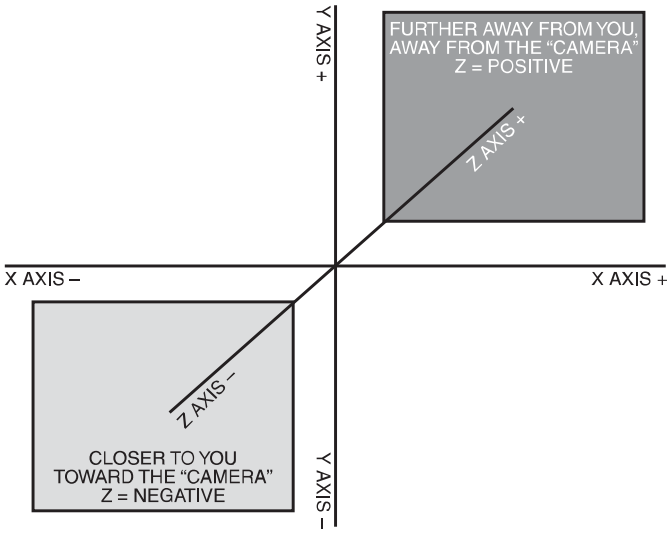

Figure 11.7 The Cartesian coordinate system with a third dimension

Grabbing Hold of the 3D Pipeline

The first thing you need to learn before you can draw a single polygon on the screen is that Direct3D uses a custom vertex format that *you* define. Here is the struct that you'll be using in this chapter:

```
struct VERTEX
{
    float x, y, z;
    float tu, tv;
};
```

The first three member variables are the position of the vertex, ande the `tu` and `tv` variables are used to describe how a texture is drawn. Now you have an incredible amount of control over how the rendering process takes place. These two variables instruct Direct3D how to draw a texture on a surface, and Direct3D supports wrapping of a texture around the curve of a 3D object. You specify the upper-left corner of the texture with `tu = 0.0` and `tv = 0.0`, and then you specify the bottom-right corner of the texture using `tu = 1.0` and `tv = 1.0`. All the polygons in between these two will usually have zeroes for the texture coordinates, which tells Direct3D to just keep on stretching the texture over them.

Texturing is an advanced subject and there are a thousand options that you will discover as you explore 3D programming in more depth. For now, let's stick to stretching a texture over two triangles in a quad.

Introducing Quads

Using the VERTEX struct as a basis, you can then create a struct that will help with creating and keeping track of quads:

```
struct QUAD
{
    VERTEX vertices;
    LPDIRECT3DVERTEXBUFFER9 buffer;
    LPDIRECT3DTEXTURE9 texture;
};
```

The QUAD struct is completely self-contained as far as the data goes. Here you have the four vertices for the four corners of the quad (made up of two triangles); you have the vertex buffer for this single quad (more on that in a minute) and you have the texture that is mapped onto the two triangles. Pretty cool, huh? The only thing missing is the code that actually creates a quad and fills the vertices with real 3D points. First, let's write a function to create a single vertex. That can then be used to create the four vertices of the quad:

```
VERTEX CreateVertex(float x, float y, float z, float tu, float tv)
{
    VERTEX vertex;
    vertex.x = x;
    vertex.y = y;
    vertex.z = z;
    vertex.tu = tu;
    vertex.tv = tv;
    return vertex;
}
```

This function just declares a temporary VERTEX variable, fills it in with the values passed to it via parameters, and then returns it. This is very convenient because there are five member variables in the VERTEX struct. I'll show you how to create and draw a quad in a bit. But first you need to learn about the vertex buffer.

The Vertex Buffer

The vertex buffer is not as scary as it might sound. My first impression of a vertex buffer was that it was some kind of surface onto which the 3D objects are drawn before being sent to the screen, sort of like a double buffer for 3D. I couldn't have been more wrong! A vertex buffer is just a place where you store the points that make up a polygon so that Direct3D can draw it. You can have many vertex buffers in your program—one for each triangle if you wish. It is common to use a vertex buffer for each 3D object in your game so that it is possible to draw each object with a simple reusable drawing function. As I'm basing this chapter on the concept of triangle-strip quads, it makes sense to create a ver-

tex buffer for each quad in the scene. I suppose that's not the fastest way in the world to render a 3D scene, but it really helps when you are just learning this material for the first time because having a vertex buffer for each quad makes it crystal-clear what's going on when a quad is rendered.

Creating a Vertex Buffer

To get started, you must define a variable for the vertex buffer:

```
LPDIRECT3DVERTEXBUFFER9 buffer;
```

Next, you can create the vertex buffer by using the `CreateVertexBuffer` function. It has this format:

```
HRESULT CreateVertexBuffer(
    UINT Length,
    DWORD Usage,
    DWORD FVF,
    D3DPOOL Pool,
    IDirect3DVertexBuffer9** ppVertexBuffer,
    HANDLE* pSharedHandle
);
```

The first parameter specifies the size of the vertex buffer, which should be big enough to hold all of the vertices for the polygons you want to render. The second parameter specifies the way in which you plan to access the vertex buffer, which is usually write-only. The third parameter specifies the vertex stream type that Direct3D expects to receive. You should pass the values corresponding to the type of vertex struct you have created. Here, we have just the position and texture coordinates in each vertex, so this value will be `D3DFVF_XYZ | D3DFVF_TEX1` (note that values are combined with *or*). Here is how I define the vertex format:

```
#define D3DFVF_MYVERTEX (D3DFVF_XYZ | D3DFVF_TEX1)
```

The fifth parameter specifies the vertex buffer pointer, and the last parameter is not needed. How about an example? Here y'go:

```
d3ddev->CreateVertexBuffer(
    4*sizeof(VERTEX),
    D3DUSAGE_WRITEONLY,
    D3DFVF_MYVERTEX,
    D3DPOOL_DEFAULT,
    &buffer,
    NULL);
```

As you can see, the first parameter receives an integer that is `sizeof(VERTEX)` times four (because there are four vertices in a quad). If you are drawing just a single triangle, you would specify `3 * sizeof(VERTEX)`, and so on for however many vertices are in your 3D object. The only really important parameters, then, are the vertex buffer length and pointer (first and fifth, respectively).

Filling the Vertex Buffer

The last step in creating a vertex buffer is to fill it with the actual vertices of your polygons. This step must follow any code that generates or loads the vertex array, as it will plug the data into the vertex buffer. For reference, here is the definition for the `QUAD` struct once more (pay particular attention to the `VERTEX` array):

```
struct QUAD
{
    VERTEX vertices;
    LPDIRECT3DVERTEXBUFFER9 buffer;
    LPDIRECT3DTEXTURE9 texture;
};
```

You can use the `CreateVertex` function, for instance, to set up the default values for a quad:

```
vertices[0] = CreateVertex(-1.0f, 1.0f, 0.0f, 0.0f, 0.0f);
vertices[1] = CreateVertex(1.0f, 1.0f, 0.0f,  1.0f, 0.0f);
vertices[2] = CreateVertex(-1.0f,-1.0f, 0.0f,  0.0f, 1.0f);
vertices[3] = CreateVertex(1.0f,-1.0f, 0.0f,  1.0f, 1.0f);
```

That is just one way to fill the vertices with data. You might define a different type of polygon somewhere in your program or load a 3D shape from a file (more on that in the next chapter!).

After you have your vertex data, you can plug it into the vertex buffer. To do so, you must `Lock` the vertex buffer, copy your vertices into the vertex buffer, and then `Unlock` the vertex buffer. Doing so requires a temporary pointer. Here is how you set up the vertex buffer with data that Direct3D can use:

```
void *temp = NULL;
buffer->Lock( 0, sizeof(vertices), (void**)&temp, 0 );
memcpy(temp,vertices, sizeof(vertices) );
buffer->Unlock();
```

For reference, here is the `Lock` definition. The second and third parameters are the important ones; they specify the length of the buffer and a pointer to it.

```
HRESULT Lock(

    UINT OffsetToLock,
```

```
    UINT SizeToLock,
    VOID **ppbData,
    DWORD Flags
);
```

Rendering the Vertex Buffer

After initializing the vertex buffer, it will be ready for the Direct3D graphics pipeline and your source vertices will no longer matter. This is called the "setup," and it is one of the features that have been moved out of the Direct3D drivers and into the GPU in recent years. Streaming the vertices and textures from the vertex buffer into the scene is handled much more quickly by a hard-coded chip than it is by software.

In the end, it's all about rendering what's inside the vertex buffer, so let's learn how to do just that. To send the vertex buffer that you're currently working on to the screen, set the stream source for the Direct3D device so that it points to your vertex buffer, and then call the `DrawPrimitive` function. Before doing this, you must first set the texture to be used. This is one of the most confusing aspects of 3D graphics, especially for a beginner. Direct3D deals with just one texture at a time, so you have to tell it which texture to use each time it changes or Direct3D will just use the last-defined texture for the entire scene! Kind of weird, huh? Well, it makes sense if you think about it. There is no pre-programmed way to tell Direct3D to use "this" texture for one polygon and "that" texture for the next polygon. You just have to write this code yourself each time the texture needs to be changed.

Well, in the case of a quad, we're just dealing with a single texture for each quad, so the concept is easier to grasp. You can create any size vertex buffer you want, but you will find it easier to understand how 3D rendering works by giving each quad its own vertex buffer. This is not the most efficient way to draw 3D objects on the screen, but it works great while you are learning the basics! Each quad can have a vertex buffer as well as a texture, and the source code to render a quad is therefore easy to grasp. As you can imagine, this makes things a lot easier to deal with because you can write a function to draw a quad, with its vertex buffer and texture easily accessible in the `QUAD` struct.

First, set the texture for this quad:

```
d3ddev->SetTexture(0, texture);
```

Next, set the stream source so that Direct3D knows where the vertices come from and how many need to be rendered:

```
d3ddev->SetStreamSource(0, q.buffer, 0, sizeof(VERTEX));
```

Finally, draw the primitive specified by the stream source, including the rendering method, starting vertex, and number of polys to draw:

```
d3ddev->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
```

Obviously, these three functions can be put into a reusable `Draw` function together (more on that shortly).

Creating a Quad

I don't know if *quad* is an official term, but it doesn't matter, because the term describes what I want to do on two levels. The first aspect of the term *quad* is that it represents four corners of a rectangle, which is the building block of most 3D scenes. You can build almost anything with a bunch of cubes (each of which is made up of six quads). As you might have guessed, those corners are represented as vertices. The second aspect of a quad is that it represents the four vertices of a triangle strip.

Drawing Triangles

There are two ways you can draw objects (all of which are made up of triangles):

- A **Triangle List** draws every single polygon independently, each with a set of three vertices.
- A **Triangle Strip** draws many polygons that are connected with shared vertices.

Obviously, the second method is more efficient and, therefore, preferable, and it helps to speed up rendering because fewer vertices must be used. But you can't render the entire scene with triangle strips because most objects are not connected to each other. Now, triangle strips work great for things like ground terrain, buildings, and other large objects. It also works well for smaller objects like the characters in your game. But what helps here is an understanding that Direct3D will render the scene at the same speed regardless of whether all the triangles are in a single vertex buffer or in multiple vertex buffers. Think of it as a series of `for` loops. Tell me which one of these two sections of code is faster. Ignore the `num++` part and just assume that "something useful" is happening inside the loop.

```
for (int n=0; n<1000; n++) num++;
```

or

```
 (for int n=0; n<250; n++) num++;
(for int n=0; n<250; n++) num++;
(for int n=0; n<250; n++) num++;
(for int n=0; n<250; n++) num++;
```

What do you think? It might seem obvious that the first code is faster because there are fewer calls. Someone who is into optimization might think the second code listing is faster because perhaps it avoids a few `if` statements here and there (it's always faster to unroll a loop and put `if` statements outside of them).

Unrolling a Loop

What do I mean when I say *unrolling a loop*? (This is not directly related to 3D, but helpful nonetheless.) Take a look at the following two groups of code (from a fictional line-drawing function, assume `x` and `y` have already been defined):

```
for (x=0; x<639; x++)
    if (x % 2 == 0)
        DrawPixel(x, y, BLUE);
    else
        DrawPixel(x, y, RED);
```

and

```
for (x=0; x<639; x+=2)
    DrawPixel(x, y, BLUE);
for (x=1; x<639; x+=2)
    DrawPixel(x, y, RED);
```

The second snippet of code is probably twice as fast as the first one because the loops have been unrolled and the `if` statement has been removed. Try to think about optimization issues like this as you work on a game because loops should be coded carefully.

A quad is made up of two triangles. The quad requires only four vertices because the triangles will be drawn as a triangle strip. Check out Figure 11.8 to see the difference between the two types of triangle rendering methods.

Figure 11.9 shows some other possibilities for triangle strips. You can join any two vertices that share a side.

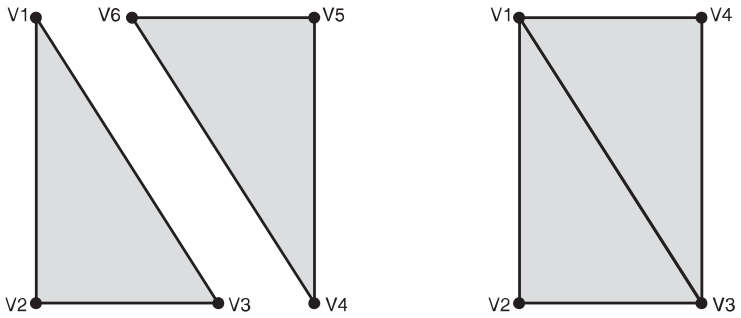

Figure 11.8 Triangle List and Triangle Strip rendering methods compared and contrasted

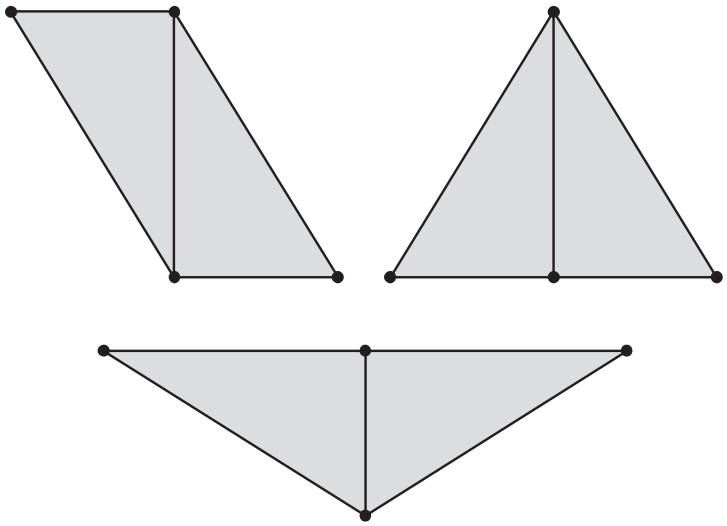

Figure 11.9 A triangle strip can take many forms. Note also that many more than two polygons can be used.

Creating the Quad

Creating a quad requires even less effort than creating two attached triangles, thanks to the triangle strip rendering process. To draw any polygon, whether it is a triangle, quad, or complete model, there are two basic steps involved.

First, you must copy the vertices into a Direct3D vertex stream. To do this, you first lock the vertex buffer, then copy the vertices to a temporary storage location with a pointer variable, then unlock the vertex buffer.

```
void *temp = NULL;
quad->buffer->Lock(0, sizeof(quad->vertices), (void**)&temp, 0);
memcpy(temp, quad->vertices, sizeof(quad->vertices));
quad->buffer->Unlock();
```

The next step is to set the texture, tell Direct3D where to find the stream source containing vertices, and then call on the `DrawPrimitive` function to draw the polygons specified in the vertex buffer stream. I like to think of this as a *Star Trek*-esque transporter. The polygons are transported from the vertex buffer into the stream and re-assembled on the screen!

```
d3ddev->SetTexture(0, quad->texture);
d3ddev->SetStreamSource(0, quad->buffer, 0, sizeof(VERTEX));
d3ddev->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
```

The Textured Cube Demo

Let's get realistic here. No one cares about drawing shaded and colored triangles, so I'm not going to waste time on the subject. Are you going to create a complete 3D game by programming triangles to assemble themselves into objects and then move them around and do collision checking and so on? Of course not, so why spend time learning about it? Triangles are critical to a 3D system, but not very useful in the singular sense. Only when you combine triangles do things get interesting.

The really interesting thing about modern 3D APIs is that it is easier to create a textured quad than one with shading. I will avoid the subject of dynamic lighting because it is beyond the scope of this book; ambient lighting will absolutely suffice for our purposes here. Did you know that *most* retail games use ambient lighting? Most of the dynamically lit games are first-person shooters.

Modifying the Framework

What comes next? Well, now that you have all this great code for doing stuff in 3D, let's just plug it into the Direct3D module in the game framework you've been building in the book. And it's about time, right? That "Direct3D" module has been stuck in 2D land for several chapters now!

There is a lot of information here, and I don't want to overwhelm you if this is your first experience with Direct3D or in 3D graphics programming in general. Anything that you do not fully grasp (or that I skim over) in this chapter will be covered again in a little more detail in the next chapter, in accordance with my "learn by repetition" concept.

The unfortunate fact of the situation at this point is that the framework is getting pretty big. There are now all of the following components in the framework that has been developed in this book:

- dxgraphics.h, dxgraphics.cpp
- dxaudio.h, dxaudio.cpp
- dxinput.h, dxinput.cpp
- winmain.cpp
- game.h, game.cpp

In addition, the DirectX components need the following support files, which are distributed with the DirectX SDK:

- dsutil.h, dsutil.cpp
- dxutil.h, dxutil.cpp

My goal is not to create some big game-engine type of library; it is just to group reusable code in a way that makes it more convenient to write DirectX programs. The problem is that many changes must be made to both the header and source file for each struct, function, and variable. So what I'm going to do at this point is just show you what code I'm adding to the framework, explain to you where it goes, and then just encourage you to open the project from the CD-ROM. The "open file and insert this code" method is just too confusing, don't you agree? Due to the way compilers work, it's just not a simple copy-and-paste operation because variables need to be defined in the header (using `extern`) before they are "declared" in the actual source file. It's an unwieldy process to say the least.

That said, I encourage you to open up the Cube_Demo project from \sources\chapter11 on the CD-ROM, which you should have copied to your hard drive already.

dxgraphics.h

I'm adding the following sections of code to dxgraphics.h. First, the function prototypes:

```
void SetPosition(QUAD*,int,float,float,float);
void SetVertex(QUAD*,int,float,float,float,float,float);
VERTEX CreateVertex(float,float,float,float,float);
QUAD* CreateQuad(char*);
void DeleteQuad(QUAD*);
void DrawQuad(QUAD*);
void SetIdentity();
void SetCamera(float,float,float,float,float,float);
void SetPerspective(float,float,float,float);
void ClearScene(D3DXCOLOR);
```

Next, the definitions for the `VERTEX` and `QUAD` structures and the camera:

```
#define D3DFVF_MYVERTEX (D3DFVF_XYZ | D3DFVF_TEX1)
struct VERTEX
{
    float x, y, z;
    float tu, tv;
};
struct QUAD
{
    VERTEX vertices[4];
    LPDIRECT3DVERTEXBUFFER9 buffer;
    LPDIRECT3DTEXTURE9 texture;
};

extern D3DXVECTOR3 cameraSource;
extern D3DXVECTOR3 cameraTarget;
```

I have not covered camera movement yet, but it is essential, and is not something I intend to just ignore. I will explain how the camera works below in the section on writing the actual Cube_Demo program.

dxgraphics.cpp

Okay, how about some really great reusable functions for 3D programming? I have gone over most of the basic code for these functions already. The rest are really just support functions that are self-explanatory. For instance, `SetPosition` just sets the position of a vertex inside a particular quad (without affecting the texture coordinates). The `SetVertex` function actually sets the position *and* the texture coordinates. These are very helpful support functions that will greatly simplify the 3D code in the main program (coming up!).

```
void SetPosition(QUAD *quad, int ivert, float x, float y, float z)
{
    quad->vertices[ivert].x = x;
    quad->vertices[ivert].y = y;
    quad->vertices[ivert].z = z;
}

void SetVertex(QUAD *quad, int ivert, float x, float y, float z, float tu, float tv)
{
    SetPosition(quad, ivert, x, y, z);
    quad->vertices[ivert].tu = tu;
    quad->vertices[ivert].tv = tv;
}

VERTEX CreateVertex(float x, float y, float z, float tu, float tv)
{
    VERTEX vertex;
    vertex.x = x;
    vertex.y = y;
    vertex.z = z;
    vertex.tu = tu;
    vertex.tv = tv;
    return vertex;
}

QUAD *CreateQuad(char *textureFilename)
{
    QUAD *quad = (QUAD*)malloc(sizeof(QUAD));

    //load the texture
```

```
    D3DXCreateTextureFromFile(d3ddev, textureFilename, &quad->texture);

    //create the vertex buffer for this quad
       d3ddev->CreateVertexBuffer(
        4*sizeof(VERTEX),
        0,
        D3DFVF_MYVERTEX, D3DPOOL_DEFAULT,
        &quad->buffer,
        NULL);

    //create the four corners of this dual triangle strip
    //each vertex is X,Y,Z and the texture coordinates U,V
    quad->vertices[0] = CreateVertex(-1.0f, 1.0f, 0.0f, 0.0f, 0.0f);
    quad->vertices[1] = CreateVertex( 1.0f, 1.0f, 0.0f, 1.0f, 0.0f);
    quad->vertices[2] = CreateVertex(-1.0f,-1.0f, 0.0f, 0.0f, 1.0f);
    quad->vertices[3] = CreateVertex( 1.0f,-1.0f, 0.0f, 1.0f, 1.0f);

    return quad;
}

void DeleteQuad(QUAD *quad)
{
    if (quad == NULL)
        return;

    //free the vertex buffer
    if (quad->buffer != NULL)
        quad->buffer->Release();

    //free the texture
    if (quad->texture != NULL)
        quad->texture->Release();

    //free the quad
    free(quad);
}

void DrawQuad(QUAD *quad)
{
    //fill vertex buffer with this quad's vertices
    void *temp = NULL;
    quad->buffer->Lock(0, sizeof(quad->vertices), (void**)&temp, 0);
```

```
    memcpy(temp, quad->vertices, sizeof(quad->vertices));
    quad->buffer->Unlock();

    //draw the textured dual triangle strip
    d3ddev->SetTexture(0, quad->texture);
    d3ddev->SetStreamSource(0, quad->buffer, 0, sizeof(VERTEX));
       d3ddev->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
}

void SetIdentity()
{
    //set default position, scale, and rotation
    D3DXMATRIX matWorld;
    D3DXMatrixTranslation(&matWorld, 0.0f, 0.0f, 0.0f);
    d3ddev->SetTransform(D3DTS_WORLD, &matWorld);
}

void ClearScene(D3DXCOLOR color)
{
    d3ddev->Clear(0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER, color, 1.0f, 0 );
}

void SetCamera(float x, float y, float z, float lookx, float looky, float lookz)
{
    D3DXMATRIX matView;
    D3DXVECTOR3 updir(0.0f,1.0f,0.0f);

    //move the camera
    cameraSource.x = x;
    cameraSource.y = y;
    cameraSource.z = z;

    //point the camera
    cameraTarget.x = lookx;
    cameraTarget.y = looky;
    cameraTarget.z = lookz;

    //set up the camera view matrix
    D3DXMatrixLookAtLH(&matView, &cameraSource, &cameraTarget, &updir);
    d3ddev->SetTransform(D3DTS_VIEW, &matView);
}
```

```
void SetPerspective(float fieldOfView, float aspectRatio, float nearRange, float
farRange)
{
    //set the perspective so things in the distance will look smaller
    D3DXMATRIX matProj;
    D3DXMatrixPerspectiveFovLH(&matProj, fieldOfView, aspectRatio, nearRange, farRange);
    d3ddev->SetTransform(D3DTS_PROJECTION, &matProj);
}
```

The Cube_Demo Program

The next step is the main code, which uses all of these reusable functions you just added to the dxgraphics module of the framework. The Cube_Demo program (shown in Figure 11.10) draws a textured cube on the screen and rotates it in the X and Z axes.

While it might seem like there are only eight vertices in a cube (refer to Figure 11.11), there are actually many more, because each triangle must have its own set of three vertices. But as

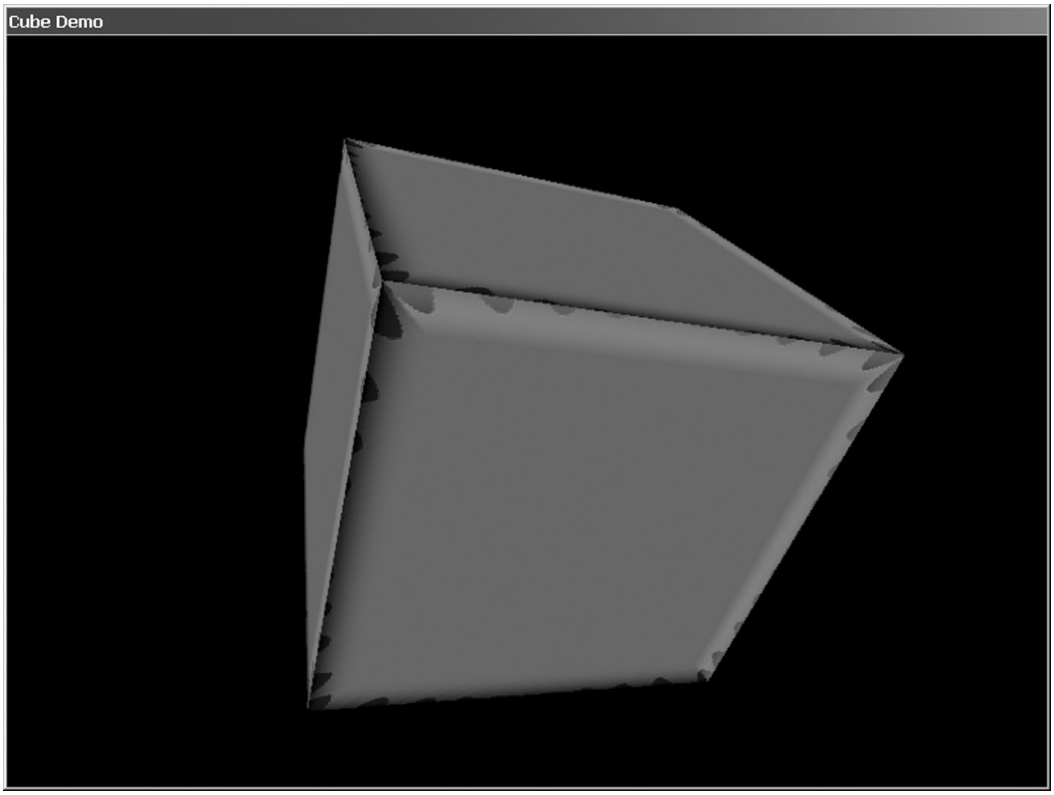

Figure 11.10 The Cube_Demo program demonstrates everything covered in this chapter about 3D programming.

you learned recently, a triangle strip works well to produce a quad with only four vertices.

As you have just worked with triangles and quads up to this point, a short introduction to cubes is in order. A cube is considered one of the simplest 3D objects you can create, and is a good shape to use as an example because it has six equal sides. As all objects in a 3D environment must be made up of triangles, it follows that a cube must also be made up of triangles. In fact, each side of a cube (which is a rectangle) is really two right triangles positioned side by side with the two right angles at opposing corners. See Figure 11.12.

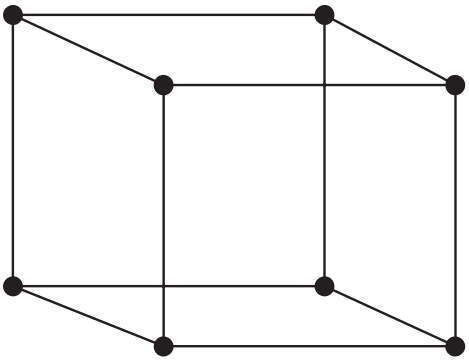

Figure 11.11 A cube might have only eight corners, but is comprised of many vertices.

note

A right triangle is a triangle that has one 90-degree angle.

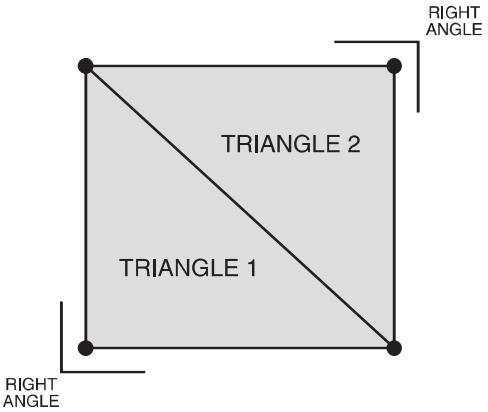

Figure 11.12 A rectangle is made up of two right triangles.

After you have put together a cube using triangles, you end up with something like Figure 11.13. This figure shows the cube sub-divided into triangles.

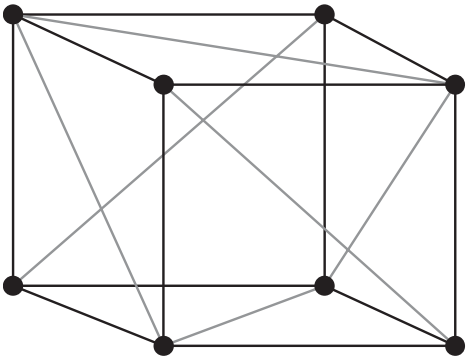

Figure 11.13 A cube is made up of six sides, with twelve triangles in all.

game.cpp

Well, now it's time to go over the main source code for the Cube_Demo program. I encourage you to load the project off the CD-ROM (which should be copied to your hard drive for convenience—and don't forget to turn off the read-only attribute so you can make changes to the files).

Nothing has changed in game.h since the last project, so you can just use one of your recent copies of game.h for this project or follow along with the Cube_Demo project itself. So much information has been covered that I elected to skip over setting up the project and so on.

```
#include "game.h"

#define BLACK D3DCOLOR_ARGB(0,0,0,0)

VERTEX cube[] = {
   {-1.0f, 1.0f,-1.0f, 0.0f,0.0f},     //side 1
   { 1.0f, 1.0f,-1.0f, 1.0f,0.0f },
   {-1.0f,-1.0f,-1.0f, 0.0f,1.0f },
   { 1.0f,-1.0f,-1.0f, 1.0f,1.0f },

   {-1.0f, 1.0f, 1.0f, 1.0f,0.0f },    //side 2
   {-1.0f,-1.0f, 1.0f, 1.0f,1.0f },
   { 1.0f, 1.0f, 1.0f, 0.0f,0.0f },
   { 1.0f,-1.0f, 1.0f, 0.0f,1.0f },

   {-1.0f, 1.0f, 1.0f, 0.0f,0.0f },    //side 3
   { 1.0f, 1.0f, 1.0f, 1.0f,0.0f },
   {-1.0f, 1.0f,-1.0f, 0.0f,1.0f },
   { 1.0f, 1.0f,-1.0f, 1.0f,1.0f },

   {-1.0f,-1.0f, 1.0f, 0.0f,0.0f },    //side 4
   {-1.0f,-1.0f,-1.0f, 1.0f,0.0f },
   { 1.0f,-1.0f, 1.0f, 0.0f,1.0f },
   { 1.0f,-1.0f,-1.0f, 1.0f,1.0f },

   { 1.0f, 1.0f,-1.0f, 0.0f,0.0f },    //side 5
   { 1.0f, 1.0f, 1.0f, 1.0f,0.0f },
   { 1.0f,-1.0f,-1.0f, 0.0f,1.0f },
   { 1.0f,-1.0f, 1.0f, 1.0f,1.0f },

   {-1.0f, 1.0f,-1.0f, 1.0f,0.0f },    //side 6
   {-1.0f,-1.0f,-1.0f, 1.0f,1.0f },
   {-1.0f, 1.0f, 1.0f, 0.0f,0.0f },
   {-1.0f,-1.0f, 1.0f, 0.0f,1.0f }
};

QUAD *quads[6];


void init_cube()
{
    for (int q=0; q<6; q++)
```

```
    {
        int i = q*4;    //little shortcut into cube array
        quads[q] = CreateQuad("cube.bmp");
        for (int v=0; v<4; v++)
        {
            quads[q]->vertices[v] = CreateVertex(
                cube[i].x, cube[i].y, cube[i].z,    //position
                cube[i].tu, cube[i].tv);            //texture coords
            i++;    //next vertex
        }
    }
}

//initializes the game
int Game_Init(HWND hwnd)
{

    //initialize keyboard
    if (!Init_Keyboard(hwnd))
    {
        MessageBox(hwnd, "Error initializing the keyboard", "Error", MB_OK);
        return 0;
    }

    //position the camera
    SetCamera(0.0f, 2.0f, -3.0f, 0, 0, 0);

    float ratio = (float)SCREEN_WIDTH / (float)SCREEN_HEIGHT;
    SetPerspective(45.0f, ratio, 0.1f, 10000.0f);

    //turn dynamic lighting off, z-buffering on
    d3ddev->SetRenderState(D3DRS_LIGHTING, FALSE);
    d3ddev->SetRenderState(D3DRS_ZENABLE, TRUE);

    //set the Direct3D stream to use the custom vertex
    d3ddev->SetFVF(D3DFVF_MYVERTEX);

    //convert the cube values into quads
    init_cube();

    //return okay
    return 1;
```

```
}

void rotate_cube()
{
    static float xrot = 0.0f;
    static float yrot = 0.0f;
    static float zrot = 0.0f;

    //rotate the x and z axes
    xrot += 0.05f;
    yrot += 0.05f;

    //create the matrices
    D3DXMATRIX matWorld;
    D3DXMATRIX matTrans;
    D3DXMATRIX matRot;

    //get an identity matrix
    D3DXMatrixTranslation(&matTrans, 0.0f, 0.0f, 0.0f);

    //rotate the cube
    D3DXMatrixRotationYawPitchRoll(&matRot,
                                   D3DXToRadian(xrot),
                                   D3DXToRadian(yrot),
                                   D3DXToRadian(zrot));
    matWorld = matRot * matTrans;

    //complete the operation
    d3ddev->SetTransform(D3DTS_WORLD, &matWorld);
}

//the main game loop
void Game_Run(HWND hwnd)
{
    ClearScene(BLACK);

    rotate_cube();

    if (d3ddev->BeginScene())
    {
        for (int n=0; n<6; n++)
            DrawQuad(quads[n]);
```

```
        d3ddev->EndScene();
    }

    d3ddev->Present(NULL, NULL, NULL, NULL);

    Poll_Keyboard();
    if (Key_Down(DIK_ESCAPE))
        PostMessage(hwnd, WM_DESTROY, 0, 0);
}

void Game_End(HWND hwnd)
{
    for (int q=0; q<6; q++)
        DeleteQuad(quads[q]);
}
```

What's Next?

That's a lot of information to digest in a single chapter, and I wouldn't be surprised if you needed to go over the information here again to really get a grasp of it. 3D programming is no easy chore, and mastery of the subject can take *years*. Don't be discouraged, though, because there is a *lot* of great things you can do *before* you have mastered it! For instance, this chapter has just scratched the surface of what you can do with even a novice understanding of Direct3D.

Figure 11.14 shows an image of a car that I created with Anim8or, a free 3D modeling program that is included on this book's CD-ROM. This powerful 3D modeling tool supports

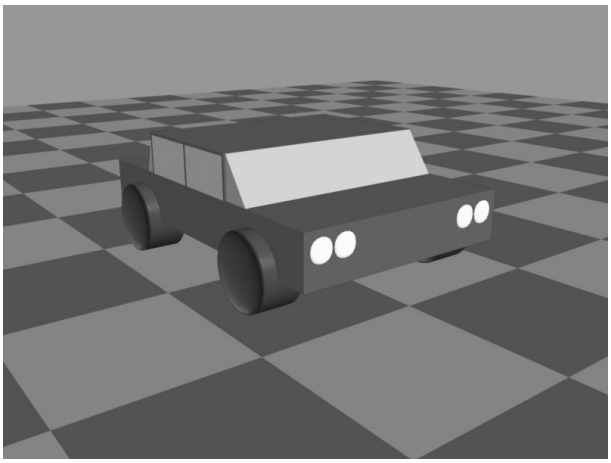

Figure 11.14 You will learn how to create and use your own 3D models in the next two chapters!

the 3D Studio Max file format (.3DS). The next two chapters will teach you how to create your own 3D models and how to load those models from a file into your Direct3D program.

What You Have Learned

This chapter has given you an overview of 3D graphics programming. You have learned a lot about Direct3D, and have seen a textured cube demo. Here are the key points:

- You learned what vertices are and how they make up a triangle.
- You learned how to create a vertex structure.
- You learned about triangle strips and triangle lists.
- You learned how to create a vertex buffer and fill it with vertices.
- You learned about quads and how to create them.
- You learned about texture mapping.
- You learned how to create a spinning cube.

Review Questions

The following questions will help to reinforce the information you have learned in this chapter.

1. What is a vertex?
2. What is the vertex buffer used for?
3. How many vertices are there in a quad?
4. How many triangles make up a quad?
5. What is the name of the Direct3D function that draws polygons?

On Your Own

The following exercises will help to challenge your retention of the information in this chapter.

Exercise 1. The Cube_Demo program creates a rotating cube that is textured. Modify the program so that the cube spins faster or slower based on keyboard input.

Exercise 2. Modify the Cube_Demo program so that each of the six sides of the cube have a different texture. Hint: You may need to copy the code from `DrawQuad` into your main source code file in order to use different textures.