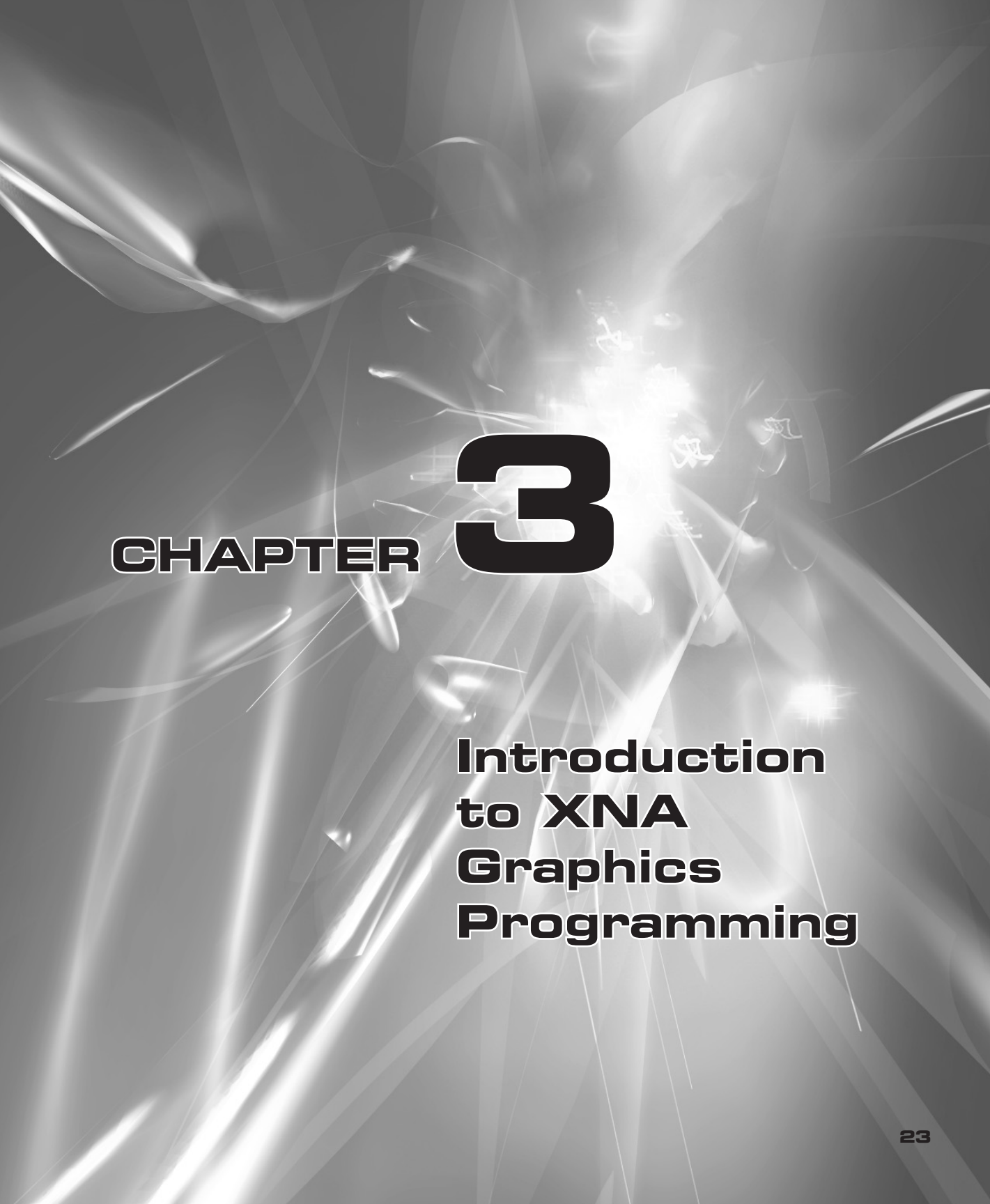
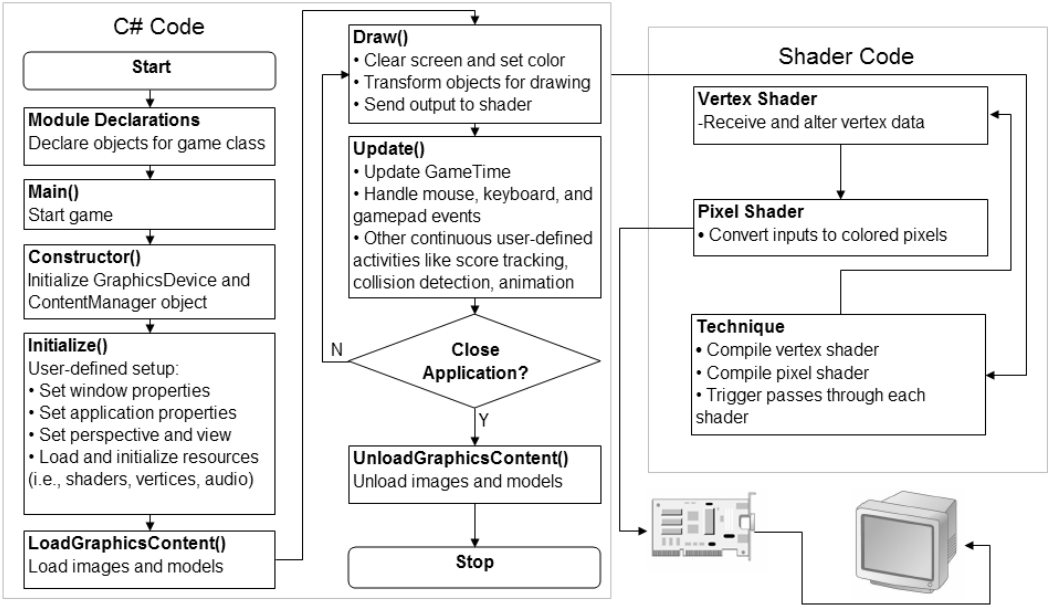


CHAPTER 3

Introduction to XNA Graphics Programming

THIS chapter discusses the most basic elements for all game graphics by introducing the structures needed to begin graphics programming. Namely, it shows how to create a game window and explains the methods behind it. Then, the methods and objects for drawing points, lines, and triangles are introduced and applied. By the end of this chapter, you could use this logic to build a basic village in a 3D world. Learning how to draw basic shapes in a game window might not grab you at first, but all great graphics effects and even 3D models are rendered with the same logic presented in this chapter.

The XNA platform offers a simple process for creating, drawing, and updating a game window. The flowchart shown here summarizes the steps required to build, update, and draw graphics within a game window.

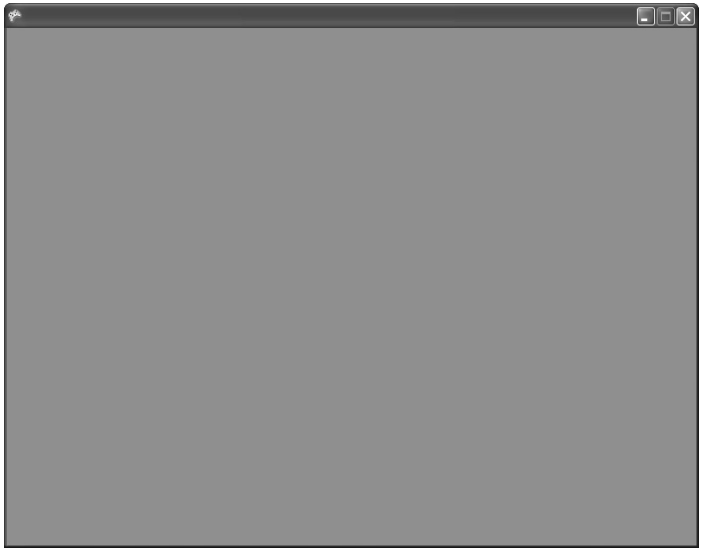


CREATING THE XNA GAME WINDOW

Hardly any C# code is needed to generate and display a basic XNA game window like the one shown in Figure 3-1.

Chapter 2, "Developer Basics," explained how to create a game studio project for either a Windows PC or the Xbox 360 platform. These projects can be created using the Xbox 360 and Windows Game project templates—they generate practically identical code. The only difference in code is the namespace for the game class. The Windows template assigns the name WindowsGame1 by default and the Xbox 360 assigns the namespace Xbox360Game1 by default—that's it. These templates

FIGURE 3-1



Basic XNA game window

provide the basic foundation you need to create an XNA game window. The XNA code in these templates is basically the same, but the XNA framework references for the Windows and Xbox 360 projects are different. You can write all of your code in one environment and then reference it in either an Xbox 360 or a Windows project to run it. Microsoft has intentionally made window creation and portability between projects simple so you can run with it. Obviously, Microsoft wants you to take the platform beyond the outer limits.

Initializing the Game Application

When you want to create a new XNA game project, the easiest method is to use the project templates that come with GSE. To begin a new project, follow these steps:

1. Open GSE by choosing Start | Programs | Microsoft XNA Game Studio Express | XNA Game Studio Express.
2. From the main GSE window, choose File | New Project.
3. Choose either the Windows Game or Xbox 360 Game template.

You may want to create your own XNA application from scratch, or you may be curious to know what's happening under the hood when you choose one of these

templates. Like any other C# application, an XNA application begins by referencing the assemblies and the namespaces required by the program. To plug into the XNA platform you will need references to the XNA framework along with namespaces for the XNA framework's Audio, Content, Graphics, Input, and Storage components. When you use an Xbox 360 or Windows Game project template, these namespaces are automatically added for you in the default Game1.cs file that is generated.

To avoid potential naming conflicts for this class (with any identically named classes), a namespace is needed for the game class. The Xbox 360 Game project template generates the namespace Xbox360Game1. The Windows Game project template generates the namespace WindowsGame1. The namespace is followed by a class declaration for the game application class, which both project templates declare as Game1. The templates also add the required assembly references for you.

GraphicsDeviceManager

Every XNA application requires a GraphicsDeviceManager object to handle the configuration and management of the graphics device. The GraphicsDevice class is used for drawing primitive-based objects. The GraphicsDeviceManager object is declared at the module level:

```
GraphicsDeviceManager   graphics;
```

The GraphicsDeviceManager object is initialized in the game class constructor, Game1():

```
graphics = new GraphicsDeviceManager(this);
```

ContentManager

The ContentManager is used to load, manage, and dispose of binary media content through the *content pipeline*. Graphics and media content can be loaded with this object when it is referenced in the game project. The ContentManager object is declared at the top of the game class:

```
ContentManager   content;
```

The ContentManager object is initialized in the constructor Game1():

```
content = new ContentManager(Services);
```

Initialize()

After the GraphicsDeviceManager and ContentManager objects have been created, you can use the Initialize() override method to trap the one-time game startup event. Initialize() is a natural place to trigger basic setup activities such as the following:

- Setting window properties such as the title or full screen options
- Setting the perspective and view to define how a user sees the 3D game
- Initializing image objects for displaying textures
- Initializing vertices for storing color, and image coordinates to be used throughout the program
- Initializing vertex shaders to convert your primitive objects to pixel output
- Initializing audio objects
- Setting up other game objects

LoadGraphicsContent()

The LoadGraphicsContent() override method is generated by the Xbox 360 and Windows Game project templates for loading binary image and model content through the graphics pipeline. You could actually load your binary graphics content from the Initialize() method or from your own methods. Chapter 7, "Texturing Your Game World," will explain how to do this. However, loading your binary graphics content from LoadGraphicsContent() ensures that your managed graphics content is always loaded in the same place.

Drawing and Updating the Game Application

Once an XNA application is initialized, it enters a continuous loop that alternates between drawing and updating the application. The sequence is generally consistent but sometimes the Draw() method will be called twice, or more, before the Update() method is called, and vice versa. Consequently, your routines for updating and drawing your objects must account for this variation in timing. All code for drawing graphics objects in the window is triggered from the Draw() method. The Update() method handles code for updating objects, handling events within the application, and your own defined events—such as checking for game object collisions, handling keyboard or game pad events, tracking the score, and tending to other game features that require maintenance every frame. Both of these functions are performed for every frame that is displayed to the player.

Draw()

The Draw() method is an override that handles the drawing (also known as *rendering*) for the game program. Throughout this book, the Draw() routine is basically the same in every example. Draw() starts by clearing the screen background, setting the screen color, and then drawing the graphics onto the screen.

Update()

The Update() method is where you check and handle game-time events. The Xbox 360 and Windows Game project templates automatically add this override method. Events typically handled here include mouse clicks, keyboard presses, game-pad control events, and timers. Update() is also a place for many other activities that require continuous checks or updates. Update() activities might include advancing animations, detecting collisions, and tracking and modifying game scores.

Closing the Game Application

The Xbox 360 and Windows Game project templates automatically add an override for the UnloadGraphicsContent() method. This method will dispose of your managed graphics media when the game program shuts down. The UnloadGraphicsContent() method also conveniently frees your memory resources even when the game application is closed unintentionally.

Basic XNA Game Window Example

This example shows all of the C# code that is generated by the Xbox 360 and Windows Game project templates. When the GSE wizard is used to create a game project, two source files are generated for your project. One of these is the Program1.cs file, which begins and launches the game application:

```
using System;

namespace WindowsGame1 // namespace is Xbox360Game1 for Xbox 360 game
{
    static class Program
    {
        static void Main(string[] args)
        {   // application entry point
            using (Game1 game = new Game1())
            {
                game.Run();
            }
        }
    }
}
```

The second default file is the Game1.cs file. This file is generated to house the game class that initializes, updates, and closes the game application:

```
// framework references
#region Using Statements
```

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Storage;
#endregion

namespace WindowsGame1
{
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics; // handles drawing
    ContentManager content;         // loads, manages, & disposes gfx media

    public Game1()
    {   // initialize graphics and content objects
        graphics    = new GraphicsDeviceManager(this);
        content     = new ContentManager(Services);
    }

    protected override void Initialize()
    {
        // initialize window, application, starting setup
        base.Initialize();
    }

    // load graphics content
    protected override void LoadGraphicsContent(bool loadAllContent)
    {
        if (loadAllContent)
        { } // managed graphics content from graphics pipeline
        else
        { } // unmanaged graphics content
    }

    protected override void UnloadGraphicsContent(bool unloadAllContent)
    {   // dispose of graphics content
        if (unloadAllContent == true)
        {
```

```
            content.Unload();
        }
    }

    protected override void Update(GameTime gameTime)
    {   // animations, collision checking, event handling

        // allows the default game to exit on Xbox 360 and Windows
        if(GamePad.GetState(PlayerIndex.One).Buttons.Back
           ==ButtonState.Pressed)
            this.Exit();

        base.Update(gameTime);
    }

    protected override void Draw(GameTime gameTime)
    {   // draw to window
        graphics.GraphicsDevice.Clear(Color.CornflowerBlue);

        // call shader code here

        base.Draw(gameTime);
    }
}
}
```

That's all of the C# code needed to draw an XNA game window, as shown previously in Figure 3-1. As you can see, creating and displaying a window is fairly simple. This code is generated by the GSE project template and will run on either the Xbox 360 or on your Window PC.

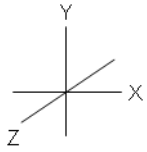
Drawing Graphics in the XNA Game Window

At this point, your XNA game window will only display 2D graphics—you will need a shader to draw 3D graphics. Shaders are explained in more detail in Chapter 4, "Shaders." In a nutshell, shaders receive vertex data from the C# application, apply filters, and then perform other user-defined operations such as texturing, coloring, and lighting. The output from the shader is pixel output in your game window.

DRAWING SHAPES

Graphics start with basic shapes that are created from points, lines, or triangles. These basic elements are referred to as *primitive objects*. Primitive objects are drawn

FIGURE 3-2



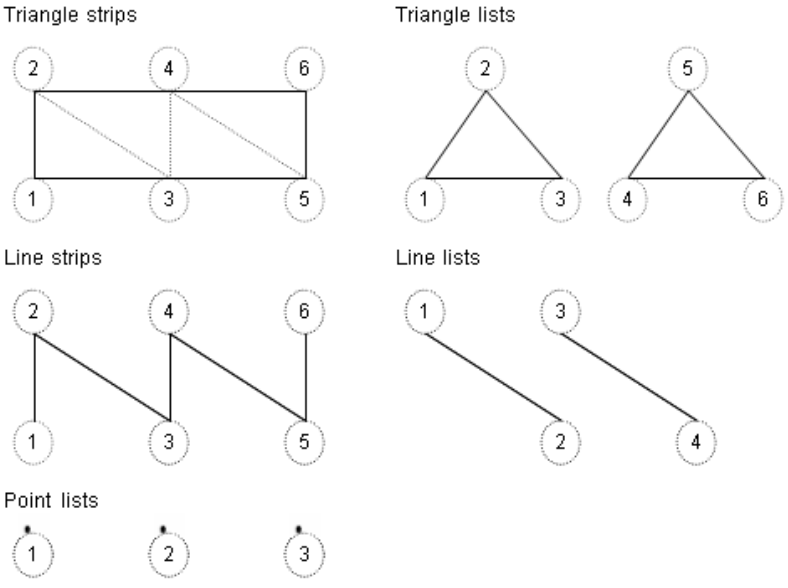
Cartesian coordinate system for drawing in 3D

in 3D space using a Cartesian coordinate system where position is mapped in the X, Y, and Z planes (see Figure 3-2).

Even complex shapes are built with a series of points, lines, or triangles. A static 3D model is basically made from a file containing vertex information that includes X, Y, Z position, color, image coordinates, and possibly other data. The vertices can be rendered by outputting points for each vertex, with a grid of lines that connects the vertices, or as a solid object that is built with a series of triangles—which are linked by the vertices.

Primitive Objects

Complex shapes are created with primitive objects that regulate how the vertices are displayed. The vertex data could be rendered as points, linear grids, or solid triangles.



Drawing Syntax

XNA delivers simple syntax for drawing shapes from primitive objects.

Primitive Object Types

Table 3-1 details the five common primitive object types. You will notice that triangles and lines can be drawn in strips or in lists. Lists are required for drawing separate points, lines, or triangles. Strips, on the other hand, are more efficient where the lines or triangles are combined to create one complex shape like a 3D model.

Strips are also more efficient than lists for saving memory and, as a result, enable faster drawing. When you're drawing a triangle strip, adding one more vertex to the strip generates one more triangle. A strip practically cuts the memory requirements for vertex data in half when compared to a list:

Total triangle list vertices = $N_{triangles}$ * 3 vertices
Total triangle strip vertices = $N_{triangles}$ + 2 vertices

The same logic applies for drawing lines. The line strip is more efficient for complex grids:

Total line list vertices = N_{lines} * 2 vertices
Total line strip vertices = N_{lines} + 1 vertex

Vertex Types

A vertex object stores vertex information, which could include X, Y, and Z positions, image coordinates, a normal vector, and color. The XNA platform offers four predefined vertex formats that are fairly self-explanatory (see Table 3-2).

TABLE 3-1

Primitive Type	*Function*
TriangleStrip	*Enables linking of triangles to create complex solid shapes*
TriangleList	*Enables groups of separate triangles*
LineStrip	*Enables linking of lines to create wire grids*
LineList	*Enables groups of separate lines*
PointList	*Enables groups of separate points*

Common Primitive Types

TABLE 3-2

Vertex Storage Format	Function
VertexPositionColor	Stores X, Y, Z and color coordinates
VertexPositionTexture	Stores X, Y, Z and image coordinates
VertexPositionNormal	Stores X, Y, Z and a normal vector
VertexPositionNormalTexture	Stores X, Y, Z, a normal vector, and image coordinates

Storage Formats for Vertex Buffers

VertexDeclaration

A VertexDeclaration object stores the vertex format for the data contained in each vertex of the shape or model. Before drawing the object, the graphics device must be set to use the correct format to allow for proper retrieval of vertex data from each vertex array. Here is the syntax required to declare and initialize the VertexDeclaration object:

```
VertexDeclaration  vertexDeclaration
= new VertexDeclaration( GraphicsDevice   gfx.GraphicsDevice,
                         VertexElement[]  elements);
```

Before an object is drawn, the graphics device's VertexDeclaration property is assigned so that it can retrieve the vertex data and render it properly:

```
gfx.GraphicsDevice.VertexDeclaration = vertexDeclaration;
```

DrawUserPrimitives

When an object is drawn using primitive types, five items are set just before it is rendered:

1. The vertex type is declared.
2. The primitive type is set so drawings can be rendered using points, lines, or triangles.
3. The vertex array that stores the X, Y, Z, color, texture, and normal data used for drawing is assigned.
4. The starting element in the vertex array is set.
5. The total count for the primitives to be drawn is assigned.

This information is passed to the DrawUserPrimitives() method:

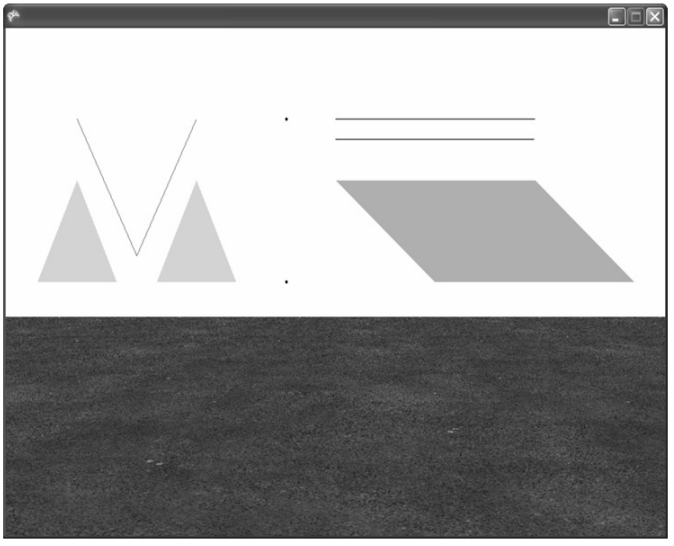
```
gfx.GraphicsDevice.DrawUserPrimitives<struct customVertex>(
                    enum    PrimitiveType,
                    struct  customVertex vertices,
                    int     startingVertex,
                    int     primitiveCount);
```

Drawing Primitive Objects Example

This demonstration shows how to draw the five common primitive shapes with vertex data. When the steps are complete, the game window will show two triangles in a strip, two triangles in a list, two lines in a strip, two lines in a list, and two points in a list (see Figure 3-3). At first glance, the output from this demonstration might seem dull, but keep in mind this is the foundation of any 3D world, so understanding it is worthwhile.

This example begins with either the WinMGHBook project or the Xbox360MGHBook project in the BaseCode directory in the download from the website. The basic code for these projects is identical. The framework differences between the two allow the WinMGHBook project to run on your PC and the Xbox360MGHBook project to run on the Xbox 360.

FIGURE 3-3



Final output for the drawing primitive objects example

With this base code, you can move through the 3D world either by moving the left thumbstick on the game controller up or down or by pressing the UP or DOWN ARROW on the keyboard. Moving the left thumbstick to the left or right allows you to strafe—as do the LEFT and RIGHT ARROW keys on the keyboard. Moving the right thumbstick, or the mouse, allows you to adjust the view. Before you start this example, you may want to run the project and experiment in the basic 3D world.

For this example, the first required addition to the base code is setting up a VertexDeclaration. The VertexDeclaration will later be used to set the vertex format for your GraphicsDevice. The GraphicsDevice must be assigned so it can retrieve data from the vertex array in the correct format and draw primitive shapes with it. In the module level of your game class (in Game1.cs) add this VertexDeclaration:

```
private VertexDeclaration mVertexDeclaration;
```

This vertex type object, mVertexDeclaration, defines the data for each vertex. You should choose the VertexPositionColor format so that you can store the position and color of all the objects that will be drawn in the example. At the end of Initialize(), add this code to define the vertex type:

```
mVertexDeclaration = new VertexDeclaration(gfx.GraphicsDevice,
VertexPositionColor.VertexElements);
```

Triangle Strip

When you work through the next portion of this example, and you run your project, two triangles will appear together in the right side of the game window.

You must declare a vertex array in your game class to store four vertices containing position and color information for the two triangles that will be drawn in the strip. To do so, add this code:

```
private VertexPositionColor[] mVtTriStrip = new VertexPositionColor[4];
```

Next, a method containing code to initialize the positions and colors for each vertex in the triangle strip can be added to the game class:

```
private void init_tri_strip()
{
    mVtTriStrip[0]=new VertexPositionColor(new Vector3(-1.5f, 0.0f, 3.0f),
                   Color.Orange);
    mVtTriStrip[1]=new VertexPositionColor(new Vector3(-1.0f, 0.5f, 3.0f),
                   Color.Orange);
    mVtTriStrip[2]=new VertexPositionColor(new Vector3(-0.5f, 0.0f, 3.0f),
                   Color.Orange);
```

```
    mVtTriStrip[3]=new VertexPositionColor(new Vector3( 0.0f, 0.5f, 3.0f),
                   Color.Orange);
}
```

The method init_tri_strip() should be called at the end of Initialize() to set up the array of vertices for the triangle strip when the program begins:

```
init_tri_strip();
```

Next, you need a method in the game class for drawing the primitive object from the vertex array. For most examples throughout this book, the drawing of primitive shapes is done in five simple steps:

1. Declare transformation matrices for scaling, moving, and rotating your graphics.
2. Initialize the transformation matrices.
3. Build the cumulative transformation by multiplying the matrices.
4. Pass the cumulative transformation to the shader.
5. Select the vertex type, primitive type, and number of vertices, and then draw the object.

The first three steps involve setting up a cumulative matrix to transform the object through scaling, translations, and rotations. Transformations are covered in Chapter 5, "Animation Introduction." More detail is presented in Chapter 4, "Shaders," to explain step 4 (where the shader variables are set). For the purpose of introducing vertices and primitive shapes in this chapter, we'll focus on step 5.

You are going to draw the triangle at the position where the vertices were defined earlier. To do this, you add the draw_objects() method to the game class. draw_objects() uses the vertex data declared earlier to draw two triangles together in a strip.

```
void draw_objects()
{
    // 1: declare matrices
    Matrix matIdentity;

    // 2: initialize matrices
    matIdentity = Matrix.Identity; // always start with identity matrix

    // 3: build cumulative world matrix using I.S.R.O.T. sequence
    // identity, scale, rotate, orbit(translate & rotate), translate
    mMatWorld = matIdentity;
```

```
    // 4: pass wvp matrix to shader
    worldViewProjParam.SetValue(mMatWorld * mMatView * mMatProj);
    mfx.CommitChanges();

    // 5: draw object - select vertex type, primitive type, # of primitives
    gfx.GraphicsDevice.VertexDeclaration = mVertexDeclaration;
    gfx.GraphicsDevice.DrawUserPrimitives<VertexPositionColor>(
        PrimitiveType.TriangleStrip, mVtTriStrip, 0, 2 );
}
```

Note the two last instructions in the draw_objects() method. The GraphicsDevice is assigned the same VertexDeclaration format defined earlier for each vertex. This property allows the GraphicsDevice to retrieve the data in the correct format, which in this case contains color and position data. The DrawUserPrimitives() method is assigned the <VertexPositionColor> format, the primitive type TriangleStrip is selected for output, and the vertex array mVtTriStrip is selected as the source of vertices with color and position data. The last two parameters of the DrawUserPrimitives() method select the offset of the vertex array and the total primitives to be drawn.

draw_objects() must be called while the BasicShader.fx file is referenced in the Draw() method. Inside the Draw() method, the call to draw_objects() must be nested between the Begin() and End() methods for the pass to the BasicShader.fx shader. (As mentioned earlier, more explanation will be provided for this section in Chapter 4, "Shaders.") To help show where draw_objects() needs to be called, some extra code that already exists in the project is included here in italics:

```
// begin shader - BasicShader.fx
// draws objects with color and position
mfx.Begin();
mfx.Techniques[0].Passes[0].Begin();
    draw_objects();
mfx.Techniques[0].Passes[0].End();
mfx.End();
```

Try running this version of the program, and you'll find that the graphics output is displayed in the game window. More specifically, two triangles in a strip will appear in the right side of the window.

Triangle List

When you need to draw separate triangles, the triangle list is handy. To continue with this example, you will display two triangles in a list in the left side of the window.

A vertex array with room for six vertices for two triangles is needed to store the position and color data that will be used to draw the triangles. To set up this array, add the following declaration to the top of the game class:

```
private VertexPositionColor[] mVtTriList = new VertexPositionColor[6];
```

A method for initializing each vertex in the triangle list, init_tri_list(), is needed in the game class:

```
private void init_tri_list()
{
    mVtTriList[0] = new VertexPositionColor(new Vector3( 0.5f, 0.0f, 3.0f),
                    Color.LightGray);
    mVtTriList[1] = new VertexPositionColor(new Vector3( 0.7f, 0.5f, 3.0f),
                    Color.LightGray);
    mVtTriList[2] = new VertexPositionColor(new Vector3( 0.9f, 0.0f, 3.0f),
                    Color.LightGray);
    mVtTriList[3] = new VertexPositionColor(new Vector3( 1.1f, 0.0f, 3.0f),
                    Color.LightGray);
    mVtTriList[4] = new VertexPositionColor(new Vector3( 1.3f, 0.5f, 3.0f),
                    Color.LightGray);
    mVtTriList[5] = new VertexPositionColor(new Vector3( 1.5f, 0.0f, 3.0f),
                    Color.LightGray);
}
```

Call init_tri_list() from Initialize() to fill the vertex array with data that can be used to draw the two triangles in the list:

```
init_tri_list();
```

At the end of draw_objects(), after the triangle strip is drawn, the triangle list can be rendered with an additional DrawUserPrimitives instruction. Drawing more than one primitive object from the same method is possible because both primitive objects use the same vertex format, VertexPositionColor. Notice that the PrimitiveType specified for this new addition is TriangleList. The total number of primitives rendered in the list is two. The data in our vertex array for the triangle list, mVtTriList, is being referenced when drawing the triangle list. The default vertex array offset of 0 is set:

```
gfx.GraphicsDevice.DrawUserPrimitives<VertexPositionColor>(
    PrimitiveType.TriangleList, mVtTriList, 0, 2);
```

When you run the new version of the program, it will show the two triangles in the strip and the two triangles in the list.

Drawing a Line Strip

You have seen how triangles can be created and drawn using strips and lists. The same logic applies for drawing lines. For this next portion of the example, a line strip will be used to draw two joined lines on the right. The line strip might be useful for you if you ever want to show a wire grid between the vertices that make the 3D object. You undoubtedly have seen this effect used when rendering 3D models or terrain with line strips instead of triangle strips.

A vertex array must be declared with the position and color data that build the line strip. For this example, enough room will be given to store two lines in the strip. In other words, three vertices are required. To declare the vertex array, add this code to the module declarations section:

```
private VertexPositionColor[] mVtLineStrip = new VertexPositionColor[3];
```

Next, add a method to store the vertex information for each of the vertices in the line strip. For each vertex, the X, Y, and Z position is specified and the color is assigned.

```
private void init_line_strip()
{
    mVtLineStrip[0]=new VertexPositionColor(new Vector3(1.3f, 0.8f, 3.0f),
    Color.Gray);
    mVtLineStrip[1]=new VertexPositionColor(new Vector3(1.0f, 0.13f, 3.0f),
    Color.Gray);
    mVtLineStrip[2]=new VertexPositionColor(new Vector3(0.7f, 0.8f,  3.0f),
    Color.Gray);
}
```

To initialize the line strip when the program begins, add the call statement for init_line_strip() to the end of the Initialize() method:

```
init_line_strip();
```

Finally, code for drawing our line strip is added as the last line in the draw_objects() method after the setup for the rendering has been completed. This instruction tells the GraphicsDevice to draw two lines in a strip using position and color data and to extract the vertex data from the mVtLineStrip array.

```
gfx.GraphicsDevice.DrawUserPrimitives
<VertexPositionColor>(PrimitiveType.LineStrip, mVtLineStrip, 0, 2);
```

When you run the game application, the output will show the line strip in the left side of the window.

Adding a Line List

Now that drawing lines using strips has been demonstrated, this next section of code will show how to add two lines that are drawn using a list.

Each line in the list requires two separate vertices. This part of the demonstration begins by showing how to draw one line in a list.

The vertex array needed to store each vertex in the line list is declared in the module declarations section of the game class.

```
private VertexPositionColor[] mVtLineList = new VertexPositionColor[4];
```

A method, init_line_list(), for initializing each vertex in the line list with X, Y, Z, and color data is added to the methods section:

```
private void init_line_list()
{
    mVtLineList[0]=new VertexPositionColor(new Vector3( 0.0f, 0.7f, 3.0f),
                   Color.Black);
    mVtLineList[1]=new VertexPositionColor(new Vector3(-1.0f, 0.7f, 3.0f),
                   Color.Black);
    mVtLineList[2]=new VertexPositionColor(new Vector3( 0.0f, 0.8f, 3.0f),
                   Color.Black);
    mVtLineList[3]=new VertexPositionColor(new Vector3(-1.0f, 0.8f, 3.0f),
                   Color.Black);
}
```

init_line_list() is called from Initialize() to set up the line list when the program begins:

```
init_line_list();
```

Finally, a new instruction should be added to the very end of the draw_objects() method to render the line list. The first parameter of the DrawUserPrimitives() method sets the LineList type, the second parameter selects the mVtLineList array as the source of vertex data for the primitive object being drawn, the third parameter sets the default array offset of 0, and the last parameter sets the total number of lines that are rendered.

```
gfx.GraphicsDevice.DrawUserPrimitives
<VertexPositionColor>(PrimitiveType.LineList, mVtLineList, 0 , 2);
```

When you run the program, two separate lines will appear in the right side of the window.

Adding a Point List

Now for our final primitive object—the point list. In this portion of the demonstration, two points from a list will be added to the window.

First, a class declaration for a vertex array is used to store each point in the list using the position and color format:

```
private VertexPositionColor[] mVtPointList = new VertexPositionColor[2];
```

Next, a method is required to initialize each vertex in the point list with X, Y, Z position data and color information. To do this, add the following method to the game class:

```
private void init_pointList()
{
    mVtPointList[0]=new VertexPositionColor(new Vector3(0.25f, 0.8f, 3.0f),
                    Color.Black);
    mVtPointList[1]=new VertexPositionColor(new Vector3(0.25f, 0.0f, 3.0f),
                    Color.Black);
}
```

The point list should be initialized when the program starts. A call to init_pointList() from the Initialize() method will do this:

```
init_pointList();
```

Now the point list can be drawn. Add the following DrawUserPrimitives() instruction to the draw_objects() method. The parameters indicate that a PointList is being rendered, two points are being drawn, and the vertex data should be read from the mVtPointList vertex array.

```
gfx.GraphicsDevice.DrawUserPrimitives
<VertexPositionColor>(PrimitiveType.PointList, mVtPointList, 0, 2);
```

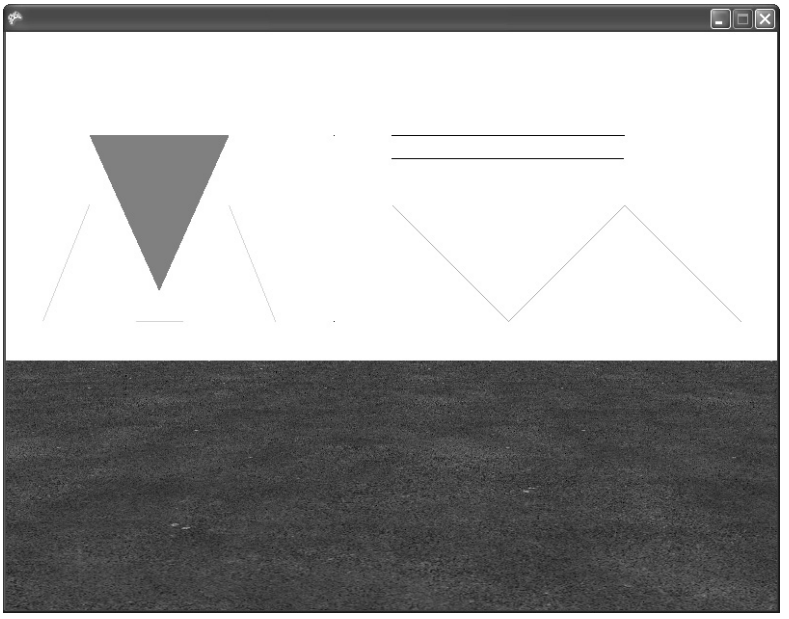
When you run the program, two points will appear in the middle of the window.

This chapter has shown how a typical XNA game window is generated, displayed, and updated with graphics. The vertices and primitive surfaces drawn with them are the foundation for all XNA game graphics. Even fiery effects and 3D models begin with vertices and primitive surfaces.

CHAPTER 3 REVIEW EXERCISES

To get the most from this chapter, try out these chapter review exercises:

1. Implement the step-by-step examples presented in this chapter.
2. After obtaining the completed solution for Exercise 1, which modifications must be made to make the output appear as shown here?



3. Use line and triangle primitives to create a small house with a roof and fence around it. You can use triangle strips, triangle lists, line strips, and line lists.