Introduction to 3D Game Engine Design Using DirectX 9 and C#

LYNN T. HARRISON

Apress™

Introduction to 3D Game Engine Design Using DirectX 9 and C#
Copyright ©2003 by Lynn T. Harrison

ISBN (pbk): 1-59059-081-3

Printed and bound in the United States of America 12345678910

Technical Reviewer: Patrick DeNardo

Editorial Board: Dan Appleman, Craig Berry, Gary Cornell, Tony Davis, Steven Rycroft, Julian Skinner, Martin Streicher, Jim Sumser, Karen Watterson, Gavin Wray, John Zukowski

Assistant Publisher: Grace Wong

Project Manager: Tracy Brown Collins

Copy Editor: Ami Knox

Production Manager: Kari Brooks

Compositor and Proofreader: Kinetic Publishing Services, LLC

Indexer: Michael Brinkman

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States: fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, email info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

CHAPTER 1

Overview

THIS BOOK IS WRITTEN FOR people who may or may not have experience with 3D graphics and want to look at the bigger picture of game engine design. This first chapter looks at game engines from a broad perspective. The goal is to provide an overview of game engines that will serve as a foundation for the detailed discussion in the rest of the book. Over the course of the book, we will explore each aspect of game engines and examine the corresponding features in the source code for the example game engine provided with the book (downloadable from the Apress Web site at `http://www.apress.com`). Each chapter focuses on a major concept or subset of the game engine. Several chapters are dedicated to different types of graphical rendering, since different techniques are employed based on the complexity of the object being rendered.

What Is a Game Engine?

A *game engine* is the very heart of any computer game. Many times the engine is highly coupled to the game and unique to that game. A properly designed engine, however, is modular, reusable, and flexible enough to be used for multiple games that are similar. A game engine is generally designed and optimized for a particular type of game. These types include first-person shooters, real-time strategy games, and vehicle simulations, to name a few of the most popular.

The commercial game engines described here are each highly optimized for their respective games, which were written by teams of highly experienced professionals who spent many man-years developing them. These commercial games must perform well on a wide range of computer hardware. They are written to use either DirectX or OpenGL rendering so that players are free to choose the one that works best for the video card in their system. The commercial software uses advanced methods such as vertex and pixel shaders with proprietary coding in order to provide their respective companies with an edge in the game market.

First-Person Shooter Game Engines

First-person shooter game engines are often based indoors. This places an emphasis on visual detail and animations. Increased visual detail comes at the cost of more textures and increased polygon count across a given game level. To compensate for this (which is necessary to maintain a reasonable frame rate), most of these engines rely on Quadtree, Portal, or Binary Space Partition (BSP) culling. *Culling* is the process of removing polygons from the scene. (More detail on these culling methods appears in Chapter 3.) Examples of this genre include Doom and Quake (in its many versions) from id Software and Half-Life (including the extremely popular Counter-Strike version) from Sierra.

Real-Time Strategy Game Engines

Until recently, real-time strategy games have been two-dimensional, sprite-based games that use a fixed viewpoint and cleverly designed sprites to give the illusion of three dimensions. Newer games such as Empire Earth from Sierra and Age of Mythology from Ensemble Studios have brought this type of game into the 3D environment. Figure 1-1 shows a scene from Age of Mythology.

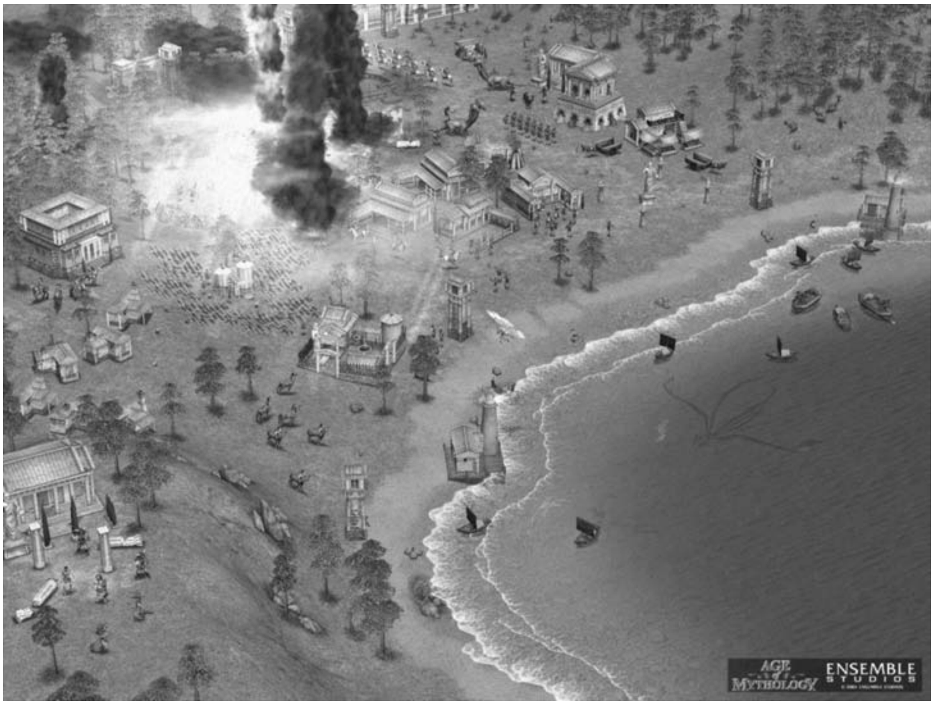

Figure 1-1. A scene from Age of Mythology

This type of game is generally set outdoors and faces the daunting task of displaying a very large number of objects at one time as well as an expansive terrain. The game engines for these games use an aerial view to reduce the number of objects and the amount of terrain that is within the player's view. These games can also use objects that are somewhat lower in resolution than those we would typically find in a first-person shooter. Since the viewpoint is kept at a fixed distance from the ground, the player doesn't typically get close enough to an object to see the details that would require high-resolution modeling.

Vehicle Simulation Game Engines

The third type of game engine mentioned is the vehicle simulation category. This group includes first-person military combat simulators (planes, helicopters, tanks, etc.), racing games, and other driving games. One example of this type of game is Comanche 4 from NovaLogic.

Since the setting for these games is outdoors and the view angle is relatively unconstrained, special techniques are required to maintain a playable frame rate. These techniques fall primarily in the areas of culling and level of detail (LOD) to reduce the number of polygons that must be textured and drawn. One of the most common methods in outdoor simulation games is to move the far clipping plane nearer to the point of view. This causes any polygons beyond the plane to be culled from the scene. An adverse effect of this is that the player can see objects appearing and disappearing at the far clipping plane as the viewpoint moves. The solution to this problem is the judicious use of fog. Fog or haze allows polygons to fade to the selected fog color as they approach the fog maximum distance (usually at or just short of the far clipping plane). Both LOD and progressive mesh technologies provide a mechanism to reduce the number of polygons in a given object in a controlled manner as a function of range to the viewpoint. LOD techniques replace highly detailed models with less detailed models as they move further from the eye. Progressive mesh, on the other hand, modifies a single mesh based on its range from the eye.

NOTE *Do not confuse game engine design with game design. The game engine is the enabling technology behind the game. Game design needs to take into account many issues that have nothing to do with the game engine. The game engine supports the game by providing the tools the game designer needs to translate a concept or storyline into a game.*

How This Book's Game Engine Project Differs

The purpose of this book is to illustrate all of the basic components of a game engine while demonstrating how to build an example game engine over the course of the book. You may expand the example game engine to create the game of your choice. I have chosen to concentrate on rendering using DirectX 9. If you prefer OpenGL, I leave it to you as a coding exercise to substitute OpenGL API calls in place of the Direct3D API calls. The same basic rendering capabilities are provided by both APIs.

Discussions of the game engine presented in this book will not delve into all of the great things we can do with vertex and pixel shaders. (As a topic, the new High Level Shader Language included in DirectX 9 could fill most of a book itself.) Nor will we explore all of the potential methods of optimizing terrain rendering or character animation.

In developing an example game engine throughout the course of this book, I will concentrate more on an object-oriented, generic approach to game engine design. This game engine is not meant for use in the next blockbuster game to hit the market. Instead, my hope is that, by providing a fundamental, yet flexible game engine, you will learn something new you can use in your own 3D game engine development. This object-oriented approach will produce a modular engine that can be easily modified and extended as you move into more advanced game engine components.

The Object-Oriented Approach

In order to develop a clean, generic design, we will use object-oriented techniques in the design and development of this game engine. Object-oriented design and programming (when done properly) provides a final system that is simple, straightforward, and easy to maintain. I have designated the C# language as the programming language for our game engine, because it provides all of the object-oriented features we desire for this project.

Why C#?

You may be wondering at the choice of the C# language for this game engine. Almost every game in recent history has been developed in C or C++ due to performance considerations. Microsoft has created the C# language in an attempt to bring the rapid application development (RAD) aspect of Visual Basic to the C++ community. As part of the .NET initiative, Microsoft has also increased the performance and object-oriented nature of Visual Basic. Its stated goal for DirectX 9 is to achieve performance values within a couple percent of C++. This makes the choice of using a RAD language very appealing. The point that tips the scales in favor of C# is the fact that C# includes a feature for self-documentation. Developing a game these days is usually a group effort. It is the rare individual who can develop a complete game on his or her own. This makes sharing well-formatted and documented code a high priority. The clean, object-oriented structure available with C#, combined with good commenting and documentation practices, can make the difference in whether or not you meet your schedule.

Starting at the Beginning—Defining a Few Primitives

Before we venture into the working code of this game engine, we will start with a few of the low-level or primitive data structures that we will be using. The C# language has two ways in which data and associated methods may be defined. The first is a structure (`struct`), which is a value type that is allocated on the stack rather than the managed heap. The structure types do not have the power and flexibility found in the class type, but they are more efficient for small data units such as the primitives that form the foundation of the game engine.

The first primitive is `Vector3`, a simple structure consisting of three single-precision floating-point variables (X, Y, and Z). This simple structure forms the very foundation of much of the 3D rendering and physical dynamics within the game engine. We will use this type to describe everything from each point in three-dimensional space as well as the speed, acceleration, and forces along each axis for an object. The three-dimensional system that we will use in this game engine is defined with X positive moving to the right of the reference point, Y positive moving up from the reference point, and Z positive moving forward from the reference point. The majority of the code found in this book will use the vector class provided with Microsoft with DirectX. There will be an example in Chapter 10 that employs our own implementation of the class for a dynamics library usable by either DirectX or OpenGL. The C# code to represent this structure is shown in Listing 1-1.

Listing 1-1. Vector3 C# Definition

```
public struct Vector3
{
public float X = 0.0;
public float Y = 0.0;
public float Z = 0.0;
}
```

NOTE *The units used in this game engine will all be in the English system. All distances will be measured in feet.*

The `Vector3` values can be referenced to two basic coordinate systems. The first system is called the *world coordinate system*. This system is centered at a point in the world database chosen by the game author. For the purpose of the game engine developed in this book, the origin of the world coordinate system will be at the southwest corner of the terrain model. With this origin, the X coordinate is positive moving east and the Z coordinate is positive moving north. The other coordinate system is referred to as the *local coordinate system*, or *body coordinate system.* For the terrain model, there is no difference between the two systems. For every other object, the local coordinate system is used in the definitions of the vertices that make up the object. As you will see in Chapter 3, both of these coordinate systems will be transformed to screen coordinates (two-dimensional coordinates) in order to draw properly on the screen.

The second primitive, `Attitude`, is a variation on `Vector3` that describes the rotation around each of these axes. You will often see these referred to as *Euler angles.* For the sake of efficiency, all angles (rotations) are stored in radians. This is the format required by trigonometric math functions. If we use degrees in developing our game engine, it might be a bit more understandable to us as programmers, but not worth the computational cost required to convert each angle every time we build a transformation matrix.

There are a few terms related to the Attitude primitive you need to know before we move on. *Pitch* is the rotation around the X-axis with positive values in the clockwise direction when looking from the origin. *Yaw* is the rotation around the Y-axis with positive values in the counterclockwise direction when looking from the origin. *Roll* is the rotation around the Z-axis with positive values in the clockwise direction. It may seem strange that yaw angles are defined a bit differently from the other two. This is done in order to match up with compass heading angles. As you may know, the compass is defined with zero degrees as north, with angles increasing in the clockwise direction. In our system, that would only be true while looking toward the axis origin. Since all of our rotations are defined as looking along the given axis from the origin, we must rotate in the opposite direction for yaw to match compass angles. The C# code to represent this structure is shown in Listing 1-2.

Listing 1-2. Attitude C# Definition

```
public struct Attitude
{
 public float Pitch = 0.0;
 public float Yaw = 0.0;
 public float Roll = 0.0;
}
```

The third and last primitive that I will define for now is `Vertex`. This structure is an extension of the `Vector3` structure that includes information used in texture mapping. A *vertex* defines a point in three-dimensional space that is part of a definition of a three-dimensional object. You will see later that a *mesh* (3D lingo for the data describing a three-dimensional shape) is made up of a list of vertices with additional information about how they are connected and the textures that cover them. The first component in the structure is a `Vector3` component called `Point` that holds the position in three-dimensional space for the vertex. The next two components (TU and TV) are the two-dimensional coordinates within a texture map that corresponds to a particular vertex's position within the texture map. The texture map is simply a bitmap that describes how the pixels at and between vertices will be rendered. If any of this seems confusing, rest assured that we will discuss this in much greater detail in Chapter 3 as we look at the rendering pipeline. We will not need to develop these vertex structures ourselves though. Microsoft has kindly provided all of the basic vertex structures for us in managed DirectX. The C# code to represent this structure is shown in Listing 1-3.

Listing 1-3. Vertex C# Definition

```
public struct Vertex
{
 public Vector3 Point;
 public Vector3 Normal;
 public Color Diffuse;
 public Color Specular;
 public float TU = 0.0;
 public float TV = 0.0;
}
```

This is the first and most basic version of the `Vertex` structure. Later in the book, you will encounter more complex versions of this structure tailored for specific purposes. These other versions will support features such as multitexturing as well as nontextured surfaces.

Interfaces: The Integration Contract

As I said earlier, this will be an object-oriented game engine. Traditionally in C++, multiple inheritance would be used to provide a collection of predefined behaviors for the various classes within the engine.

While this is a powerful technique, it does have its dangers. If a class inherits from two base classes that define attributes or methods with the same name, there would be a problem. The duplicated names would "collide" and produce a compiler error. The C# language addresses this problem by not allowing multiple inheritance. C# classes are allowed to inherit from only a single base class. To provide the benefits of multiple inheritance without the dangers, C# uses a mechanism called an *interface*. If you are familiar with C++, you can think of an interface as a pure virtual or abstract base class. A C# interface may define methods and properties but does not include the implementation of either. A C# class may inherit from as many interfaces as it needs. It is the responsibility of the inheriting class to implement each method and property defined by the interfaces it inherits. If it fails to provide a method or property defined in an inherited interface, the compiler will flag an error.

Class instances may be queried at run time to determine if they support a given interface. This allows us to have a collection of objects of different types, iterate through the collection, and call interface methods of those objects that support the interface. Since each class provides its own implementation for the interface methods, it is free to provide an implementation that is unique and appropriate for that class. As an example, let's look at two classes that implement an interface called `IRenderable` declaring a method called `Render`. This interface will be discussed in detail shortly. For now, accept that a class uses this method in order to draw the object to the screen appropriately for the current view.

For this example, we will assume that one object is a billboard made from two triangles to represent a tree and that the other object is the terrain model, with hundreds of thousands of triangles for an entire outdoors game. It is easy to see how the requirements and implementation of this method must be different for these two classes. The billboard needs only to draw the two triangles oriented toward the point of view and textured to look like a tree. The terrain class, on the other hand, must first determine which triangles are visible (no current hardware can render a world of this size in real time for every frame), transform the vertices for the texture appropriately for the current view, and draw and texture.

Let's look at a view of the more important interfaces that we will use in this game engine. We will get into various implementations of these interfaces as we progress through the book. The code for these interfaces is shown in Listing 1-4, which appears later in this section.

The first interface is the `IRenderable` interface mentioned previously. A class that implements this interface is able to render an image of itself to the screen using the `Render` method. The argument of this method is the camera definition that defines the current view. This is all the information any class implementing this interface requires to render itself.

The second interface, `ICullable`, is implemented by any class that may not always be rendered to the display. This interface defines two properties. The properties manage the cull state of the object (whether the object should be rendered or not). The first property defined is `Culled`, which is responsible for clearing the cull state flag to the not culled state. The second property is defined as a read-only Boolean variable that is read with a `Get` method, `IsCulled`. It is important for game efficiency that any graphical object support this interface. As mentioned earlier when discussing terrain, the number of triangles in an object would overload the video card if not reduced to only the visible subset.

The next interface is the `ICollidable` interface. Any class whose object might physically collide with another object should support this interface. The properties and methods of this interface support the testing for collisions between two objects. The interface specifies several properties that expose the object's geometry in several levels of detail. In order for this interface to work, both objects involved must support the interface. The first property is a `Vector3` property called `CenterOfMass`. This property defines the location of the center of the object in world coordinates. The second property of the object is `BoundingRadius`. This value defines a sphere around the object—the smallest possible sphere centered on the center of mass that completely encloses the object.

The first method defined by the interface is `CollideSphere`, which takes an object reference as an argument. This method performs a spherical collision check between the two objects. This is the quickest collision check possible, since it only needs to check the distance between the two objects against the sum of the bounding radii of the two objects. This is a low-fidelity collision check, as it is possible to report a false positive if the two objects are close together without any of the polygonal faces intersecting or coming into contact. If neither of the objects is the player's model, and both are far enough from the viewpoint or otherwise out of view, this might be sufficient. Otherwise, we would normally proceed to using the second method of this interface. This method, `CollidePolygon`, takes three `Vector3` variables as arguments. The method is called for each polygon in one of the models until a collision is detected or all polygons have returned a false Boolean value. As you can see, this is far more computationally expensive. Unfortunately, we must go to this extent if we want 100 percent confidence in the collision test.

The next interface that we will look at is the `IDynamic` interface. This interface supports any object that moves or changes as time progresses. Only one method is defined for this interface: `Update`. The only argument to this method is a floating-point variable containing the number of milliseconds since the object was last updated. This uses the method for integration of the position and attitude of the object, the step to the proper frame of an animation, or both. The properties of the interface are related to the physical dynamics, which I will address in detail in Chapter 10.

The final interface that we will discuss for now is `ITerrainInfo`. Any class that may be queried for information about the terrain implements this interface. This information is vital for any object that moves along or over the surface of the terrain. The first method for this interface is `HeightOfTerrain`, which returns the Y-axis value of the terrain at the supplied location in meters. By preventing an object from moving below this value, the object stays on the surface rather than dropping through. The second method, `HeightAboveTerrain`, is an extension of the first method that returns the difference between the Y-axis value passed in as part of the location and the height of the terrain at that point. This method is important for objects that are in flight above the terrain and striving not to collide with the surface. The next method is `InLineOfSight`, which returns a positive (true) value if there is an unobstructed line of sight between the two points that are supplied as arguments. The final method of the interface, GetSlope, is used by any object (such as a ground vehicle) to match its attitude with that of the slope it is resting upon. As you can see in the code in Listing 1-4, this method accepts the location in question and the heading of the object making the call. The heading allows the method to return an attitude that is rotated to match the object's heading.

Listing 1-4. Interface Definitions

```
public interface IRenderable
{
   void Render(Camera cam);
}

public interface ICullable
{
   bool Culled { set; }
   bool IsCulled { get; }
}

public interface ICollidable
{
   Vector3 CenterOfMass { get; }
   float   BoundingRadius { get; }
   bool CollideSphere ( Object3D other );
   bool CollidePolygon ( Vector3 Point1, Vector3 Point2, Vector3 Point3 );
}

public interface IDynamic
{
   void Update( float DeltaT );
}

public interface ITerrainInfo
{
   float    HeightOfTerrain( Vector3 Position );
   float    HeightAboveTerrain( Vector3 Position );
   bool     InLineOfSight( Vector3 Position1, Vector3 Position2 );
   Attitude GetSlope( Vector3 Position, float Heading );
}
```

Process Flow Overview

Although this is not part of the game engine design, we need to look at the typical game application structure and process flow that will employ the engine to provide a framework for the game engine that we will develop. This process flow is based on observations of numerous commercial computer games. The important thing to remember is that the steps in this flow are not hard-and-set requirements. Do not feel bound to include any steps that do not seem appropriate. The implementation of this process for this book's example game will be a simple state machine.

We will have five states in our game process:

- Developer splash screen
- Game splash screen
- Options
- Game play
- After action review

There will be one or more triggers. Each trigger controls the transition from one state to the next. These triggers will be a keystroke, mouse click, or timer expiration as appropriate for whichever state is current.

Developer Splash Screen

The first thing that a player will see when starting this game will be the developer splash screen. Think of this as analogous to seeing the movie studio logo at the beginning of a movie. It lets the viewer know who created the wonderful media production they are about to enjoy. This splash screen could be either a static image or a short video depending on the preferences (and production capabilities) of the game developer. For the purposes of our sample game engine, we will be treating all game resources as separate files rather than building them into the executable as resources. The game engine has a method called `ShowSplash`, which takes a string argument holding the path to the image to be displayed. The implementation of this method will be described in Chapter 2 as part of the player interface. The method call will look like this:

```
m_Engine.ShowSplash("devsplash.jpg", 8, new CGameEngine.BackgroundTask(LoadOptions));
```

This splash screen will remain displayed until a state change causes it to be replaced by something else. The ShowSplash method does not include the state change logic to transition to the next state. The primary reason for this is that time spent displaying a splash screen can be put to good use. This is a good time to preload other resources the game will need, rather than have the player wait later. My preference is to spend the time during this splash screen to load configuration and option data. This data includes the video resolution to use in the actual playing portion of the game, special-effect filter values to exclude some flashy special effects that the player's computer might not be able to support, and keyboard key mappings that let the player choose how he or she prefers to control the game.

This configuration data will be loaded using another call to an engine method (LoadOptions). Once the time has expired for the splash screen (say 8 seconds), or the player has pressed a key on the keyboard or clicked a mouse button, the game state will be advanced to the game splash screen state.

Game Splash Screen

After reminding players who developed this wonderful game, it is time for a flashy reminder of just what game they are about to play. This splash screen may actually come in two varieties. The very first time the game is played, we want to set the stage for the player so that they understand the premise for the game. A registry entry could be set after the game has run the first time. If this value is not set, we show an extended introduction. Otherwise, we make a short introduction and proceed directly to the game.

Game development studios with large production budgets tend to provide an extended high-quality video (often with live casts) to present the opening story for the game. Other studios with more modest budgets will produce animated videos developed through the same 3D modeling software used to build the 3D models within the game. Garage developers are typically reduced to displaying one or more static images with text as the introduction to their game. Regardless of the source of the media used, the technology to present the media is the same, the ShowSplash method we use when presenting the developer splash screen.

Usually when a longer opening is being presented, the user is not allowed to terminate the presentation before it has completed. A flag stored with the configuration data may be used to record whether this is the first time the game has been played. If this is not the first time, we could program the game to play a shorter opening or allow early termination into the next state, or both.

As with the first splash screen, we have processing time to spare while the player is enjoying the splash screen. This time can be spent loading additional resources, such as texture bitmaps, 3D model files common to all variations of the game, etc., we will need so that once the player hits the Play button there is no significant delay before the fun starts.

Presenting the Options

In all but the simplest games, we are still not ready for the player to start playing the game. The player will likely want to make a few choices before starting to play. What screen resolutions will the game use? What controls and keystrokes will the player use to control the game? What map, level, or scenario will he or she be playing? Will the player be loading a previously saved game? Will he or she be playing a single player game or connecting with a multiplayer game? The number of options that we give the player will determine the screens required to present the options.

If the game requires more than one option screen (our example game does not), we would require a state for each screen. The details on how each screen is presented will be covered in detail in Chapter 2. We will use several different styles of option screens (button oriented and Windows dialog based) to give us the tools we need to develop our own option screens. The code extract in the "Looking at the C#" section at the end of this chapter shows the control structure used for transitioning through the option screen states.

The most important options are the ones that lead into playing the game itself. Once the play game state is entered, the game engine itself takes center stage to control the play.

Playing the Game

Once the primary game state is entered, the game itself starts. The game loop begins executing as soon as the player enters this state. The game loop consists of the following general steps:

- Process player inputs.
- Calculate automated player actions.
- Update the dynamics for all dynamic models.
- If in multiplayer mode, exchange state information with other players (not included in the sample game engine).
- Render the next frame.

These steps that make up the game loop continue to execute as long as the game is in the play state. The first step works with the control preferences defined through one of the option screens. These preferences map mouse, joystick, or keystroke actions to game control functions, which include movement control, weapons control if applicable, and game state control (save game, exit game, activate game console, etc.).

The second step in the game loop provides a similar control function for any computer-controlled models. By controlling the automated models at the same level of fidelity as the player-controlled model, it provides several positive factors. It simplifies the dynamics software by eliminating duplication of creating two separate versions to support both player-controlled and automated models. The other bonus is the degree of fidelity attained by keeping a level playing field between the player and the automated opponents.

Next, we need to update all of the dynamic models. In other words, everything that should move is moved. Objects that support the `IDynamic` interface have their `Update` method called in order to calculate new positions or sequence an animation. Once everything has been updated, we are almost ready to redraw the screen. Before we continue on to the rendering phase, we check to see if this is a multiplayer game. If it is, then we must send the state of any local controller objects out to the other players and accept any updates that have been sent to us.

Now we are finally ready to update the screen. The `Render` method of each object is called. The order in which we make calls is important for some of the objects. As you will see in Chapter 4, the `SkyBox` object (if it exists for this game) should always be rendered first so that it appears in the background. User interface and console objects should always be rendered last so that they appear on top of the scene. If any of this seems a bit confusing right now, no need for you to worry. As we build up the game engine during the course of the book, each of these steps will be dealt with in detail.

After everything has been rendered, it is time to make a decision. Is the game over? If not, we branch back to the process player inputs step and continue to iterate the game loop. If the game is over, it is time to proceed to the next state—player scoring.

After Action Review: Player Scoring

After the game play has completed, it is good to sum things up for the player. Depending on the game, this summary could come in one of several forms. For simple games, this might be no more than a display of high scores that gives players an indication how they fared against other players (or themselves on another attempt). If there were automated opponents in the game, it could show how players ranked against the automated opponents. If the game was multiplayer, it could show how each player ranked in that game. In a scenario-based game, we could present the option to restart the game to play the same scenario again. This step is optional, but most good games provide some form of feedback to the player on completion.

Once players indicate through some form of user input that they are finished looking at the scoring page, it is time to change states again. The normal procedure is to set the game state back to the main (or only) option screen. This allows players the opportunity to set up and play another game or exit the game entirely.

Looking at the C# Code

The code in Listing 1-5 is from the application class for the sample game. As you will see, the application inherits from another class called `CD3DApplication`. This class is based on a class supplied by Microsoft with the DirectX 9 SDK. A handy base class takes care of all the initial bookkeeping required to initialize and terminate DirectX. I have modified the Microsoft version slightly to start in full-screen mode and adapted the processing loop to the one used with the game engine. If you are interested in the details of setting up and tearing down DirectX 9, I encourage you to look at the source code downloadable from the Apress Web site.

Listing 1-5. Sample Application

```
//-----------------------------------------------------------------------------
// File: App.cs
//
// Desc: Sample code for Introduction to 3D Game Engine Design
//
//      This sample shows the basic application software that sets up the
//       base application and the process flow.  The application uses a version
//       of the CD3DApplication base class provided with the Microsoft
//       DirectX 9 SDK to perform the standard initialization of DirectX.
//
//       Note: This code uses the D3D Framework helper library.
//
// Copyright (c) 2002 Lynn T. Harrison. All rights reserved.
//-----------------------------------------------------------------------------
using System;
using System.Drawing;
using System.Collections;
using Microsoft.DirectX;
using Microsoft.DirectX.Direct3D;
using Microsoft.DirectX.DirectInput;
using GameEngine;
using GameAI;

namespace SampleGame
{
   /// <summary>
   /// Summary description for GameEngine.
   /// </summary>
   class CGameApplication : GraphicsSample
   {
      #region   // Game State enumeration
      /// <summary>
      /// Each member of this enumeration is one possible state for the
      /// application
      /// </summary>
      ///
      /// <remarks>
      /// DevSplash          - Display the developer splash screen
      /// </remarks>
      /// <remarks>
      /// GameSplash         - Display the game splash screen
      /// </remarks>
      /// <remarks>
      /// OptionsMain        - Display and process the primary options screen
      /// </remarks>
      /// <remarks>
      /// GamePlay           - State to actually play the game
      /// </remarks>
      /// <remarks>
      /// AfterActionReview - Display the results of the game
      /// </remarks>
      public enum GameState
      {
         /// <summary>
         /// Display the developer splash screen
         /// </summary>
         DevSplash,
         /// <summary>
         /// Display the game splash screen
         /// </summary>
         GameSplash,
         /// <summary>
      /// Display and process the primary options screen
      /// </summary>
      OptionsMain,
      /// <summary>
      /// State to actually play the game
      /// </summary>
      GamePlay,
      /// <summary>
      /// Display the results of the game
      /// </summary>
      AfterActionReview,
   }
#endregion

   #region // Application member variables
   /// <summary>
   /// Current state of the application
   /// </summary>
   private GameState          m_State;
   private static CGameEngine m_Engine = new CGameEngine();
   private GraphicsFont       m_pFont = null;
   private GameEngine.Console m_Console;
   private ArrayList          m_opponents = null;
   private OptionScreen       m_OptionScreen = null;
   private bool               m_bShowStatistics = false;
   private bool               m_bScreenCapture = false;
   private bool               m_bUsingJoystick = true;
   private bool               m_bUsingKeyboard = false;
   private bool               m_bUsingMouse = false;
   private Ownship            m_ownship = null;
   private Cloth              m_flag = null;
   private Jukebox            music = null;
   #endregion

   public static CGameEngine Engine { get { return m_Engine; } }

   /// <summary>
   /// Application constructor. Sets attributes for the app.
   /// </summary>
   public CGameApplication()
   {
      // Initialize the game state for the developer splash screen.
      m_State = GameState.DevSplash;
```

```
    m_pFont = new GraphicsFont( "Aerial", System.Drawing.FontStyle.Bold );
    windowed = false;

    m_opponents = new ArrayList();
}

/// <summary>
/// Called during initial app startup, this function performs all the
/// permanent initialization.
/// </summary>
protected override void OneTimeSceneInitialization()
{
    // Initialize the font's internal textures.
    m_pFont.InitializeDeviceObjects( device );

    // Nothing much to do yet - will be used in later chapters.
    m_Engine.Initialize( this, device );

    CGameEngine.Inputs.MapKeyboardAction(Key.Escape,
        new ButtonAction(Terminate), true);
    CGameEngine.Inputs.MapKeyboardAction(Key.A,
        new ButtonAction(MoveCameraXM), false);
    CGameEngine.Inputs.MapKeyboardAction(Key.W,
        new ButtonAction(MoveCameraZP), false);
    CGameEngine.Inputs.MapKeyboardAction(Key.S,
        new ButtonAction(MoveCameraXP), false);
    CGameEngine.Inputs.MapKeyboardAction(Key.Z,
        new ButtonAction(MoveCameraZM), false);
    CGameEngine.Inputs.MapKeyboardAction(Key.P,
        new ButtonAction(ScreenCapture), true);
    CGameEngine.Inputs.MapMouseAxisAction(0,
        new AxisAction(PointCamera));
    CGameEngine.Inputs.MapMouseAxisAction(1,
        new AxisAction(PitchCamera));

    m_Console = new GameEngine.Console( m_pFont, "console.jpg" );

    GameEngine.Console.AddCommand("QUIT", "Terminate the game",
        new CommandFunction(TerminateCommand));
    GameEngine.Console.AddCommand("STATISTICS",
        "Toggle statistics display",
        new CommandFunction(ToggleStatistics));
    m_OptionScreen = new OptionScreen( "Options1.jpg" );
    m_OptionScreen.AddButton( 328, 150, "PlayOff.jpg", "PlayOn.jpg",
        "PlayHover.jpg", new ButtonFunction(Play) );
    m_OptionScreen.AddButton( 328, 300, "QuitOff.jpg", "QuitOn.jpg",
        "QuitHover.jpg", new ButtonFunction(Terminate) );
    m_Engine.SetOptionScreen( m_OptionScreen );

    music = new Jukebox();
    music.AddSong("nadine.mp3");
    music.AddSong("ComeOn.mp3");
    music.AddSong("Rock.mp3");
    music.Volume = 0.75f;
    music.Play();
}

/// <summary>
/// Called once per frame, the call is the entry point for all game
/// processing. This function calls the appropriate part of the
/// game engine based on the
/// engine based on the current state.
/// </summary>
protected override void FrameMove()
{
    try
    {
        SelectControls select_form = null;
        // get any player inputs
        m_Engine.GetPlayerInputs();

        // Clear the viewport.
        device.Clear( ClearFlags.Target | ClearFlags.ZBuffer,
            0x00000000, 1.0f, 0 );

        device.BeginScene();

      // Determine what needs to be rendered based on the current game state.
      switch ( m_State )
      {
          case GameState.DevSplash:
              if ( m_Engine.ShowSplash("devsplash.jpg", 8,
                      new BackgroundTask(LoadOptions)) )
              {
                  m_State = GameState.GameSplash;
              }
```

```
         break;
      case GameState.GameSplash:
         if ( m_Engine.ShowSplash("gamesplash.jpg", 8, null) )
         {
            m_State = GameState.OptionsMain;
            select_form = new SelectControls();
            select_form.ShowDialog(this);
            m_bUsingJoystick = select_form.UseJoystick.Checked;
            m_bUsingKeyboard = select_form.UseKeyboard.Checked;
            m_bUsingMouse = select_form.UseMouse.Checked;
            if ( m_bUsingJoystick )
               GameEngine.Console.AddLine("Using Joystick");
            if ( m_bUsingKeyboard )
               GameEngine.Console.AddLine("Using Keyboard");
            if ( m_bUsingMouse )
               GameEngine.Console.AddLine("Using Mouse");
            m_ownship = (Ownship)Engine.GetObject("car1");
            m_ownship.UseJoystick = m_bUsingJoystick;
            m_ownship.UseKeyboard = m_bUsingKeyboard;
            m_ownship.UseMouse = m_bUsingMouse;
         }
         break;
      case GameState.OptionsMain:
         m_Engine.DoOptions();
         break;
      case GameState.GamePlay:
         m_Engine.GetPlayerInputs();
         m_Engine.DoAI( elapsedTime );
         m_Engine.DoDynamics( elapsedTime );
         m_Engine.DoNetworking( elapsedTime );
         m_Engine.Render();
         break;
      case GameState.AfterActionReview:
         m_Engine.DoAfterActionReview();
         break;
   }

   GameEngine.Console.Render();

   if ( m_ownship != null && m_State == GameState.GamePlay )
   {
      m_pFont.DrawText( 200, 560, Color.FromArgb(255,0,0,0),
               m_ownship.MPH.ToString() );
            m_pFont.DrawText( 200, 580, Color.FromArgb(255,0,0,0),
                      m_ownship.ForwardVelocity.ToString() );
            m_pFont.DrawText( 200, 600, Color.FromArgb(255,0,0,0),
                      m_ownship.SidewaysVelocity.ToString() );
         }

         // Output statistics.
         if ( m_bShowStatistics )
         {
            m_pFont.DrawText( 2, 560, Color.FromArgb(255,255,255,0),
               frameStats );
            m_pFont.DrawText( 2, 580, Color.FromArgb(255,255,255,0),
               deviceStats );
         }

         if ( m_bScreenCapture )
         {
           SurfaceLoader.Save("capture.bmp",ImageFileFormat.Bmp,
                      device.GetBackBuffer(0,0,BackBufferType.Mono));
            m_bScreenCapture = false;
            GameEngine.Console.AddLine("snapshot taken");
         }
      }
      catch (DirectXException d3de)
      {
         System.Diagnostics.Debug.WriteLine(
                      "Error in Sample Game Application FrameMove" );
         System.Diagnostics.Debug.WriteLine(d3de.ErrorString);
      }
      catch ( Exception e )
      {
         System.Diagnostics.Debug.WriteLine(
                      "Error in Sample Game Application FrameMove" );
         System.Diagnostics.Debug.WriteLine(e.Message);
      }
      finally
      {
         device.EndScene();
      }
   }

   /// <summary>
   /// The main entry point for the application
   /// </summary>
```

```
[STAThread]
static void Main(string[] args)
{
   try
   {
      CGameApplication d3dApp = new CGameApplication();
      if (d3dApp.CreateGraphicsSample())
         d3dApp.Run();
   }
   catch (DirectXException d3de)
   {
      System.Diagnostics.Debug.WriteLine(
                   "Error in Sample Game Application" );
      System.Diagnostics.Debug.WriteLine(d3de.ErrorString);
   }
   catch ( Exception e )
   {
      System.Diagnostics.Debug.WriteLine(
                   "Error in Sample Game Application" );
      System.Diagnostics.Debug.WriteLine(e.Message);
   }
}

// Action functions

/// <summary>
/// Action to start playing
/// </summary>
public void Play()
{
   m_State = GameState.GamePlay;
   GameEngine.Console.Reset();
}

/// <summary>
/// Action to terminate the application
/// </summary>
public void Terminate()
{
   m_bTerminate = true;
}
```

```
/// <summary>
/// Screen capture
/// </summary>
public void ScreenCapture()
{
   m_bScreenCapture = true;
}

/// <summary>
/// Version of terminate for use by the console
/// </summary>
/// <param name="sData"></param>
public void TerminateCommand( string sData )
{
   Terminate();
}

/// <summary>
/// Toggle the display of statistics information.
/// </summary>
/// <param name="sData"></param>
public void ToggleStatistics( string sData )
{
   m_bShowStatistics = !m_bShowStatistics;
}

/// <summary>
/// Action to transition to the next game state based on a mapper action
/// </summary>
public void NextState()
{
   if ( m_State < GameState.AfterActionReview )
   {
      m_State++;
   }
   else
   {
      m_State = GameState.OptionsMain;
   }
}
```

```
public void PointCamera( int count )
{
   m_Engine.MoveCamera(0.0f, 0.0f, 0.0f, 0.0f, 0.0f, count);
}

public void PitchCamera( int count )
{
   m_Engine.MoveCamera(0.0f, 0.0f, 0.0f, count * 0.1f, 0.0f, 0.0f);
}

public void MoveCameraXP()
{
   m_Engine.MoveCamera(0.5f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f);
}

public void MoveCameraXM()
{
   m_Engine.MoveCamera(-0.5f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f);
}

public void MoveCameraY()
{
   m_Engine.MoveCamera(0.0f, 0.5f, 0.0f, 0.0f, 0.0f, 0.0f);
}

public void MoveCameraZP()
{
   m_Engine.MoveCamera(0.0f, 0.0f, 0.5f, 0.0f, 0.0f, 0.0f);
}

public void MoveCameraZM()
{
   m_Engine.MoveCamera(0.0f, 0.0f, -0.5f, 0.0f, 0.0f, 0.0f);
}

/// <summary>
///
/// </summary>
protected override void RestoreDeviceObjects(System.Object sender,
   System.EventArgs e)
{
   // Set the transform matrices (view and world are updated per frame).
   Matrix matProj;
```

```
    float fAspect = device.PresentationParameters.BackBufferWidth /
                    (float)device.PresentationParameters.BackBufferHeight;
    matProj = Matrix.PerspectiveFovLH( (float)Math.PI/4, fAspect,
                    1.0f, 100.0f );
    device.Transform.Projection = matProj;

    // Set up the default texture states.
    device.TextureState[0].ColorOperation = TextureOperation.Modulate;
    device.TextureState[0].ColorArgument1 = TextureArgument.TextureColor;
    device.TextureState[0].ColorArgument2 = TextureArgument.Diffuse;
    device.TextureState[0].AlphaOperation = TextureOperation.SelectArg1;
    device.TextureState[0].AlphaArgument1 = TextureArgument.TextureColor;
    device.SamplerState[0].MinFilter = TextureFilter.Linear;
    device.SamplerState[0].MagFilter = TextureFilter.Linear;
    device.SamplerState[0].MipFilter = TextureFilter.Linear;
    device.SamplerState[0].AddressU = TextureAddress.Clamp;
    device.SamplerState[0].AddressV = TextureAddress.Clamp;

    device.RenderState.DitherEnable = true;
}

/// <summary>
/// Called when the app is exiting, or the device is being changed, this
/// function deletes any device-dependent objects.
/// </summary>
protected override void DeleteDeviceObjects(System.Object sender,
    System.EventArgs e)
{
    m_Engine.Dispose();
}

public void LoadOptions()
{
    try
    {
      System.Random rand = new System.Random();
      // Loading of options will happen here.
      m_Engine.SetTerrain(200,200,"heightmap.jpg","sand1.jpg", 10.0f, 0.45f);

       for ( int i=0; i<300; i++ )
       {
          float north = (float)(rand.NextDouble() * 1900.0);
          float east  = (float)(rand.NextDouble() * 1900.0);
```

```
   BillBoard.Add( east, north, 0.0f, "cactus"+i, "cactus.dds",
      1.0f, 1.0f);
}
for ( int i=0; i<300; i++ )
{
   float north = (float)(rand.NextDouble() * 1900.0);
   float east  = (float)(rand.NextDouble() * 1900.0);
   BillBoard.Add( east, north, 0.0f, "tree"+i, "palmtree.dds",
      6.5f, 10.0f);
}
GameEngine.Console.AddLine("all trees loaded");

 m_Engine.AddObject( new ParticleGenerator("Spray1", 2000, 2000,
            Color.Yellow, "Particle.bmp",
            new ParticleUpdate(Gravity)));

double j = 0.0;
double center_x = 1000.0;
double center_z = 1000.0;
double radius = 700.0;
double width = 20.0;

m_flag = new Cloth("flag", "flag.jpg", 2, 2, 0.1, 1.0f);
m_flag.Height = 0.6f;
m_flag.North = 2.0f;
m_flag.East = 0.1f;
Cloth.EastWind = -3.0f;

for ( double i=0.0; i<360.0; i += 1.5 )
{
   float north = (float)(center_z +
            Math.Cos(i/180.0*Math.PI) * radius );
   float east  = (float)(center_x +
            Math.Sin(i/180.0*Math.PI) * radius );
   BillBoard.Add( east, north, 0.0f, "redpost"+
            (int)(i*2), "redpost.dds",0.25f, 1.0f);
   j += 5.0;
   if ( j > 360.0 ) j -= 360.0;
}

j = 0.0;
for ( double i=0.5; i<360.0; i += 1.5 )
{
   float north = (float)(center_z +
            Math.Cos(i/180.0*Math.PI) * (radius+width) );
      float east  = (float)(center_x +
                Math.Sin(i/180.0*Math.PI) * (radius+width) );
      BillBoard.Add( east, north, 0.0f, "bluepost"+
                (int)(i*2), "bluepost.dds",0.25f, 1.0f);
      j += 5.0;
      if ( j >= 360.0 ) j -= 360.0;
   }

   m_ownship = new Ownship(this, "car1", "SprintRacer.x",
                new Vector3(0.0f, 0.8f, 0.0f),
                 new Attitude(0.0f, (float)Math.PI, 0.0f));
   m_ownship.AddChild(m_flag);

   SoundEffect.Volume = 0.25f;

   m_Engine.AddObject( m_ownship );

   m_ownship.North = 298.0f;
   m_ownship.East = 1000.0f;
   m_Engine.Cam.Attach(m_ownship, new Vector3(0.0f, 0.85f,-4.5f));
   m_Engine.Cam.LookAt(m_ownship);
   m_ownship.Heading = (float)Math.PI * 1.5f;
   m_ownship.SetLOD( 10, 3000.0f );

   GameEngine.GameLights headlights =
                GameEngine.GameLights.AddSpotLight(
                new Vector3(0.0f,0.0f,0.0f),
      new Vector3(1.0f,0.0f,1.0f), Color.White, "headlight");
   headlights.EffectiveRange = 200.0f;
   headlights.Attenuation0 = 1.0f;
   headlights.Attenuation1 = 0.0f;
   headlights.InnerConeAngle = 1.0f;
   headlights.OuterConeAngle = 1.5f;
   headlights.PositionOffset = new Vector3(0.0f, 2.0f, 1.0f);
   headlights.DirectionOffset = new Vector3(0.0f, 0.00f, 1.0f);
   m_ownship.AddChild(headlights);
   headlights.Enabled = false;

   CGameEngine.FogColor = Color.Beige;
   CGameEngine.FogDensity = 0.5f;
   CGameEngine.FogEnable = true;
   CGameEngine.FogStart = 100.0f;
   CGameEngine.FogEnd = 900.0f;
   CGameEngine.FogTableMode = FogMode.Linear;
}
```

```
            catch ( Exception e )
            {
               GameEngine.Console.AddLine("Exception");
               GameEngine.Console.AddLine(e.Message);
            }
         }

         public void Gravity( ref Particle Obj, float DeltaT )
         {
            Obj.m_Position   += Obj.m_Velocity * DeltaT;
            Obj.m_Velocity.Y  += -9.8f * DeltaT;
            if ( Obj.m_Position.Y < 0.0f ) Obj.m_bActive = false;
         }


         public void OwnshipUpdate( Object3D Obj, float DeltaT )
         {
         }

         public void OpponentUpdate( Object3D Obj, float DeltaT )
         {

            Obj.Height = CGameEngine.Ground.HeightOfTerrain(Obj.Position) +
                     ((Model)Obj).Offset.Y;
         }

      }
   }
```

Summary

In this chapter, we have looked at what a game engine is in general. Simply restated, a game engine is the underlying technology used in writing a game. By employing a game engine, the game designers are free to concentrate on the game play and any applicable story lines within the game. The chapter has also presented the basic design of this sample game engine at a high level. We have also looked at how the game application interfaces with some of the higher-level game engine functions for controlling the flow of the game.